Corretto: A CUP of Java with Grappa

A Tool for Parser Generation

Laura Apostoloiu

York University, Department of Computer Science
4700 Keele Street, Toronto, Canada M3J 1P3

November 26, 2002

Abstract

CUP is a parser generator. The generated parser recognizes valid sequences of tokens. Java snippets, known as action code, can be included in a CUP specification such that the generated parser also builds parse trees (also known as concrete syntax trees). However, writing this low level code is tedious and error prone.

In Java, the nodes of a syntax tree are represented by objects. Typically, for each construct a class is introduced. To represent a language like Java, a few hundred such classes are needed. Writing these classes by hand is tedious. Tools, like Grappa, have been developed that mechanically generate such classes from a specification.

Because a CUP specification and a Grappa specification have a lot in common and since the CUP action code for building concrete syntax trees can be generated from the Grappa specification, we have merged the two specifications into one Corretto specification. The tool Corretto extracts from a Corretto specification the corresponding CUP and Grappa specifications.

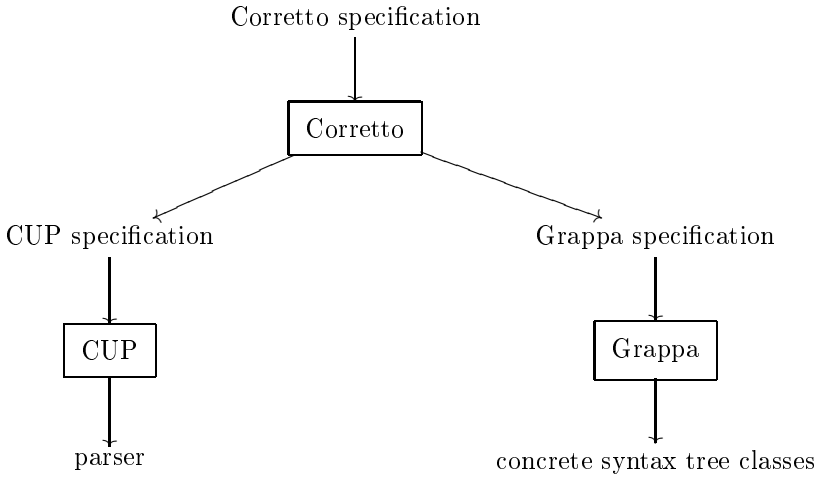
1 Introduction

CUP (Constructor of Useful Parsers) [6] serves the same role as the widely used program YACC [7]: given a specification, CUP generates an LALR parser. A CUP specification consists of a decorated grammar. A parser takes as input a sequence of tokens, typically produced by a lexer, and checks if this sequence is part of the language specified by the grammar. While YACC is entirely written in C, CUP is developed using Java and produces parsers that are implemented in Java.

Besides recognizing valid sequences of tokens, parsers often build parse trees, also known as concrete syntax trees. The productions of a CUP specification can be augmented with snippets of Java code, known as action, for this purpose. Adding action code is tedious and error prone. In Java, the nodes of a syntax tree are represented by objects. Typically, we introduce a class for each construct of the language. For a language like Java, a few hundred such classes need to be introduced. Several tools have been developed to mechanically generate these classes from a specification. In this report, we discuss one of them, namely Grappa [1].

Given a CUP and a Grappa specification, using the tools CUP and Grappa, we can construct a parser and a collection of classes to represent concrete syntax trees. The generated parser not only recognizes valid sequences of tokens, but also builds a parse tree for those sequences. A CUP specification and a Grappa specification contain similar information. In particular, the productions have a lot in common. As we will see, the action code of the CUP specification can be extracted from the Grappa specification. Therefore, we propose to combine the CUP and Grappa specification into one specification. This saves us from creating two specifications and keeping them consistent. Furthermore, we do not have to write the action code by hand as it can be generated mechanically.

Our tool, which we call Corretto[1], extracts a CUP specification and a Grappa specification from the common specification. These extracted specifications can subsequently be used to produce a parser and a hierarchy of classes. The parser uses these classes to build concrete syntax trees.



Our main contributions are two fold. First of all, we have implemented Grappa in Java. The tool was originally implemented in Perl. In the interest of widespread utilization of the tool, we decided that Grappa should be capable of running on all major platforms. We chose Java as it is platform independent. Furthermore, the current popularity of Java makes the inner workings of our tool accessible to a large audience. Secondly, we have designed a syntax for Corretto specifications and we have implemented Corretto in Java.

The rest of this report is organized as follows. In Section 2, we describe the syntax of CUP. The tools Grappa and Corretto are discussed in Section 3 and Section 4, respectively. An example Corretto specification and the corresponding CUP and Grappa specifications are given in Section 5. In Section 6, we conclude and discuss future work.

1.1 Acknowledgements

I would like to express my gratitude to Professor Franck van Breugel, my supervisor, whose expertise, understanding and patience have added considerably to my graduate experience. I strongly appreciate his knowledge, skills and the invaluable assistance in writing this report.

I would also like to thank Professor Richard Paige, whose valuable input and comments did improve the quality of this report.

Department of Computer Science of York University and the entire academic and support staff deserves my appreciation for the valuable experience I gained through my years as graduate student here.

I wish to acknowledge my very good friend, Razvan Dumitrescu, for encouraging me to pursue and finish a graduate degree.

Finally, I want to thank my family for the support they provided me through my entire life and in particular, I must acknowledge my husband and best friend, Mihai, without whose love and encouragement I would not have finished this thesis.

2 CUP specification

A CUP specification consists of five sections. Below, we discuss four of them. The fifth one allows for inclusion of snippets of Java code into the generated parser. Since the CUP specifications generated by our tool Corretto do not need such a section, we will not discuss the fifth section here, but we refer the interested

[1] In Italian, corretto means corrected. It is used for coffee, in particular for espresso. In that case, the coffee is corrected with a dash of an alcoholic beverage like grappa.

reader to the CUP user's manual [6] for details. Below, we present the four remaining sections of a CUP specification. As we will see, some of them are optional. However, if present, the sections should appear in the order they are presented below.

As a running example, we consider simple arithmetic expressions over integers and lists of these expressions. The expressions are defined by the production

$$
\begin{array}{lcl}
\textit{Expression} & \rightarrow & \textit{Expression} \texttt{ + } \textit{Expression} \\
& | & \textit{Expression} \texttt{ - } \textit{Expression} \\
& | & \textit{Expression} \texttt{ * } \textit{Expression} \\
& | & \textit{Expression} \texttt{ / } \textit{Expression} \\
& | & \texttt{(}\textit{Expression}\texttt{)} \\
& | & \textit{Number} \\
& | & \infty
\end{array}
$$

where *Number* can be an arbitrary integer. The lists of expressions are captured by the production

$$
\begin{array}{lcl}
\textit{List} & \rightarrow & \\
& | & \textit{Expression List}
\end{array}
$$

We assume that we already have a lexer that, given an arithmetic expression or a list of expressions, produces a sequence of tokens. Let us assume that the symbols `+`, `-`, `*`, `/`, `(`, `)` and ∞ are represented by the tokens `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `LPAREN`, `RPAREN` and `INFINITY` and that an integer is represented by the token `NUMBER`. Furthermore, we assume that the lexer associates an `Integer` object with each `NUMBER` token. This object represents the actual integer value. (Such a lexer can be generated by, for example, JFlex [8].) Next, we present the different sections of a CUP specification to generate a parser to recognize sequences of these tokens.

2.1 Package and import specifications

This first section is optional. It may contain a package declaration and one or more import declarations. These declarations have the same syntax and role as the package and import declarations in an ordinary Java program.

The package declaration is used to indicate which package the generated parser is part of. For example, to express that the parser for arithmetic expressions is part of the package `expression`, we write

```
package expression;
```

Any import declaration given in the CUP specification will also appear in the generated parser. This allows us to make direct use of the imported classes in the action code (which we will discuss in Section 3.5). For example,

```
import java.util.*;
```

imports all the classes of the package `java.util.*`.

2.2 Symbols

This second part is required. It consists of the declaration of the terminal and nonterminal symbols that appear in the grammar. In our example, we have two nonterminals, *Expression* and *List*, and eight nonterminals represented by the tokens `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `LPAREN`, `RPAREN`, `INFINITY` and `NUMBER`. These symbols can be declared as follows.

```
nonterminal List;
nonterminal Expression;
terminal ADD, SUBTRACT, MULTIPLY, DIVIDE, LPAREN, RPAREN, INFINITY;
terminal Integer NUMBER;
```

The declaration of `NUMBER` specifies that an `Integer` object is associated with a `NUMBER` token. How to exploit these `Integer` objects we will discuss in Section 3.5.

2.3 Precedence and associativity specifications

This third section is optional as well. It specifies the precedence and associativity of the terminals. For example, to express that `*` and `/` have precedence over `+` and `-`, and that all are left associative, we write

```
precedence left ADD, SUBTRACT;
precedence left MULTIPLY, DIVIDE;
```

The order of precedence, from highest to lowest, is from bottom to top. Any terminal whose precedence is not declared has lowest precedence. Besides specifying that a terminal is `left` associative, we can also specify that it is `right` associative or `nonassociative`. There is no default for associativity.

2.4 Productions

The fourth section is required. It contains the productions. For our example, this section amounts to

```
List ::=
       | Expression List
       ;

Expression ::= Expression ADD Expression
             | Expression SUBTRACT Expression
             | Expression MULTIPLY Expression
             | Expression DIVIDE Expression
             | LPAREN Expression RPAREN
             | NUMBER
             | INFINITY
             ;
```

The start nonterminal is the left hand side of the first production. In our example, `List` is the start nonterminal. As we already mentioned in the introduction, fragments of Java code can be added to the productions. We will discuss the addition of action code to the above production in Section 3.5. A more elaborate CUP specification is given in Section 5.2.

For more details about CUP, we refer the reader to the CUP user's manual [6] and to Appel's textbook [2, pages 70–84].

3 Grappa

Grappa is a tool for automatically generating Java classes to represent syntax trees. The tool was designed by Antia [1]. It was originally implemented in Perl. In this section we discuss the tool and describe our implementation of the tool in Java. Before presenting the tool, we first discuss how to represent syntax trees in Java. For more details about Grappa, including a discussion of related tools, we refer the reader to Antia's thesis [1].

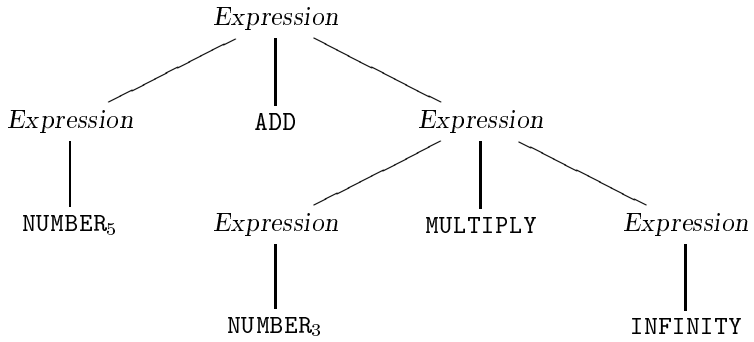
3.1 Syntax tree representation in Java

To represent syntax trees in Java we use a rather standard approach. For a detailed discussion of this approach we refer the reader to the textbooks of Appel [2, pages 96–105] and of Watt and Brown [13, Section 4.4] and the thesis of Antia [1]. Here, we only present the approach by means of an example.

Let us consider the arithmetic expressions over integers we introduced in the previous section. The sequence of tokens

```
NUMBER ADD NUMBER MULTIPLY INFINITY
```

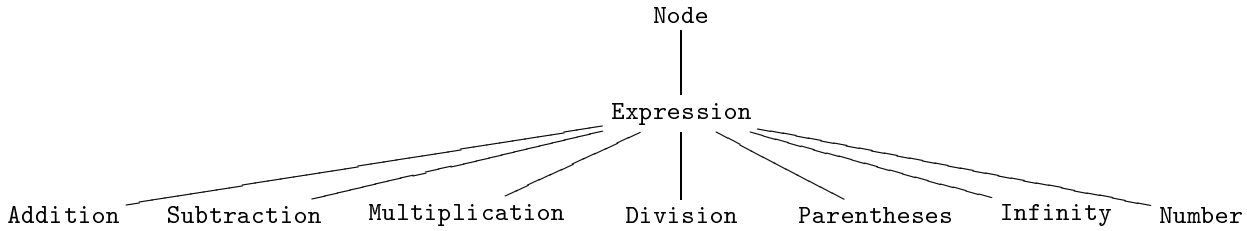
is successfully recognized by the parser generated by CUP. Let us assume that the two `NUMBER` tokens contain the integer values 5 and 3, respectively. The corresponding parse tree, also known as concrete syntax tree, looks as follows.



We will represent the above parse tree by the Java object

```
new Addition(new Number(new Integer(5)),
             new Multiplication(new Number(new Integer(3)),
                                new Infinity()))
```

The classes `Addition`, `Multiplication` and `Number` are part of the following inheritance hierarchy.

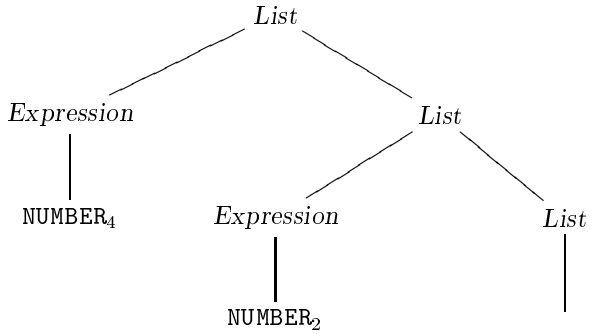


The classes `Node` and `Expression` are abstract whereas all other classes are concrete. By making these classes abstract we disallow the creation of concrete syntax trees that correspond to incomplete parse trees. The class `Node` is a superclass of all the classes. This class contains information general to all classes like the position within the original source file of the characters from which this `Node` of the syntax tree is derived. The classes `Addition`, `Subtraction`, `Multiplication` and `Division` have two instance variables of type `Expression`. These represent the first and second operand. The class `Parentheses` has an instance variable of type `Expression`. The class `Number` has an instance variable of type `Integer`, which holds the actual integer value. The class `Infinity` does not have any instance variables.

Let us look at another example. This time we consider the list of expressions represented by the sequence of tokens

$\texttt{NUMBER}_4\ \texttt{NUMBER}_2$

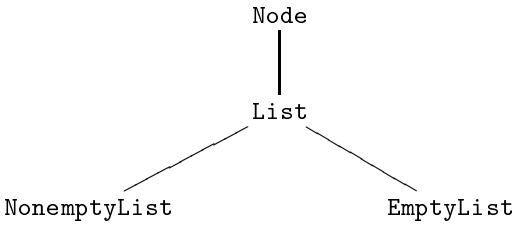
The corresponding concrete syntax tree looks as follows.



This tree is represented by the Java object

```
new NonemptyList(new Number(new Integer(4)),
                 new NonemptyList(new Number(new Integer(2)),
                                  new EmptyList()))
```

The classes `List`, `NonemptyList` and `EmptyList` form the following inheritance hierarchy.



The class `List` is abstract and the classes `NonemptyList` and `EmptyList` are concrete. `NonemptyList` has instance variables of type `Expression` and `List` and `EmptyList` does not have any instance variables.

3.2 Grappa specification

Given a specification, the Grappa tool generates a collection of Java classes to represent syntax trees. Below, we present a specification that gives rise to the classes to represent concrete syntax trees for the arithmetic expressions and lists of these expressions.

```
package expression;

% List of expressions %
List : % Empty list of expressions %
       EmptyList
       empty

     | % Nonempty list of expressions %
       NonemptyList
       % head : first expression of the list,
         tail : rest of the list %
       Expression List
     ;

% Arithmetic expression over integers %
Expression : % Addition %
             Addition
             % first : first expression of the addition,
               second : second expression of the addition %
             Expression Expression

           | % Subtraction %
             Subtraction
             % first : first expression of the subtraction,
               second : second expression of the subtraction %
             Expression Expression

           | % Multiplication %
             Multiplication
             % first : first expression of the multiplication,
               second : second expression of the multiplication %
             Expression Expression
```

```
  | % Division %
    Division
    % first : first expression of the division,
      second : second expression of the division %
    Expression Expression

  | % Parenthesized expression %
    Parentheses
    % exp : expression %
    Expression

  | % Infinity %
    Infinity

  | % Number %
    Number
    % value : integer value %
    Integer
  ;
```

A specification consists of two parts. These two parts will be discussed below.

3.2.1 Package and import specifications

The first part is optional. It may contain a package declaration and one or more import declarations. These declarations have the same syntax and role as the package and import declarations in an ordinary Java program.

The package declaration is used to indicate which package the generated classes are part of. The classes generated from the above specification are all part of the package `expression`.

Any import declaration given in the Grappa specification will also appear in all the generated classes. This allows us to make direct use of the imported classes in the generated classes. For example,

```
import java.util.*;
```

imports all the classes of the package `java.util`. Hence, we can use, for example, `Vector` rather than `java.util.Vector` in the rest of the specification.

3.2.2 Productions

The second part is required. It consists of the productions. There are two different types of productions.

The first type of production is of the form

% comment for class C %
C : % comment for class S_1 %
  S_1
  % $v_{1,1}$: comment for instance variable $v_{1,1}$,
    ⋮
    v_{1,n_1} : comment for instance variable v_{1,n_1} %
  $T_{1,1} \cdots T_{1,n_1}$
⋮
| % comment for class S_m %
  S_m
  % $v_{m,1}$: comment for instance variable $v_{m,1}$,

$\vdots$
v_{m,n_m} : comment for instance variable v_{m,n_m} %
$T_{m,1} \ldots T_{m,n_m}$
;

where $m \geq 2$ and $n_i \geq 0$ for $i = 1, \ldots, m$. From such a production, classes named C and $S_1, \ldots, S_m$ are generated. The abstract class C has subclasses $S_1, \ldots, S_m$ and the concrete class S_i has instance variables named $v_{i,1}, \ldots, v_{i,n_i}$ of type $T_{i,1}, \ldots, T_{i,n_i}$. The production in the above Grappa specification is of this first type of production. If $n_i = 0$ then the right hand side is of the form

```
...
| % comment for class S_i %
  S_i
| ...
```

That is, the block of instance variable definitions is absent. Hence, the class S_i has no instance variables. The fragment of one of the S_j's may have the following form

```
...
| % comment for class S_j %
  S_j
  empty
| ...
```

where `empty` is a Grappa keyword. Also in this case, the class S_j has no instance variables. The main difference between S_i and S_j is that S_i is associated with a token, like, for example, the class `Infinity` is associated with the token `INFINITY`, whereas S_j is not. We will come back to this difference in Section 3.4.

The second type of production is of the form

% comment for class C %
C : % v_1 : comment for instance variable v_1 ,
$\vdots$
v_n : comment for instance variable v_n %
$T_1 \ldots T_n$
;

where $n \geq 0$. From this production a class named C is generated. The concrete class C has instance variables named $v_1, \ldots, v_n$ of type $T_1, \ldots, T_n$. An example of a production of this second type is presented below.

```
% Function declaration %
FunctionDeclaration : % name : name of the function,
                        parameters : parameters of the function,
                        type : return type of the function,
                        body : body of the function %
                      Identifier FieldList Type Expression
                    ;
```

The specifications of the classes `Identifier`, `FieldList`, `Type` and `Expression` are provided in Section 5.1.

Given a set of productions, let $\mathcal{C}$ be the collection of class names that occur as C or S_i in any of the productions, and let $\mathcal{T}$ be the collection of class names that occur as $T_{i,j}$ or T_j in any of the productions, and let $\mathcal{V}$ be the collection of instance variable names that occur in any of the productions. A set of productions should satisfy the following conditions.

1. $\mathcal{C}$ does not contain any duplicates. This prevents us from assigning the same name to different classes.

2. Each class name in $\mathcal{C}$ and each instance variable name in $\mathcal{V}$ consists of a sequence of Java letters and Java digits, the first of which must be a Java letter, and the name is different from a Java keyword (see [5, § 3.9]), `true`, `false`, `null` and the Grappa keyword `empty`. A character is a Java letter if the method `Character.isJavaIdentifierStart` returns true. A character is a Java letter or a Java digit if the method `Character.isJavaIdentifierPart` returns true. Note that the characters `:`, `%`, | and `;`, which play a special role in Grappa specifications, are neither Java letters nor Java digits. This second condition ensures that the names of the generated classes and the names of the instance variables of the generated classes are valid according to the Java language specification [5].

3. Each class name in $\mathcal{C}$ is different from the name of any class in the package `java.lang` or in any of the imported packages. This avoids confusion between generated classes and imported classes.

4. The collections of instance variables $v_{i,1}, \ldots, v_{i,n_i}$ and $v_1, \ldots, v_n$ do not contain any duplicates. This ensures that the names of the instance variables of a generated class are all different.

5. Each class name in $\mathcal{T}$ is either in $\mathcal{C}$ or is part of the package `java.lang` or any of the imported packages. This ensures that all the instance variables of the generated classes have a well-defined type.

The productions of the above Grappa specification satisfies all the above conditions. The current implementation of Grappa does not check these well-formedness constraints.

3.3 Grappa grammar

Next, we present the grammar that defines the syntax of Grappa specifications. This syntax has been designed such that it can easily be integrated with a CUP specification into one specification. This integration is discussed in the next section. Furthermore, in the design we aimed for specifications that are easy to read. Finally, the syntax has been designed such that the Grappa tool is easy to implement.

In the grammar we use $[x]$ to denote zero or one occurrences of x and $\{x\}$ to denote zero or more occurrences of x.

Below, we use *Identifier* for names of classes, instance variables and packages. As we already mentioned before, such a name consists of a sequence of Java letters and Java digits, the first of which must be a Java letter, and the name is different from a Java keyword, `true`, `false`, `null` and the Grappa keyword `empty`. A *ClassComment* consists of a sequence of characters different from `%` and an *InstanceVariableComment* consists of a sequence of characters different from `%`, `:` and `,`.

A Grappa specification has two parts: declarations and productions.

Specification → *Declarations Productions*

The declaration part may contain a package declaration and one or more import declarations.

Declarations → [*PackageDeclaration*] { *ImportDeclaration* }
PackageDeclaration → `package` *Identifier* { `.` *Identifier* } `;`
ImportDeclaration → `import` *Identifier* { `.` *Identifier* } [`.*`] `;`

There are two types of productions.

Productions → *Production* { *Production* }
Production → `%` *ClassComment* `%` *Identifier* `:` *Definitions* `;`
Definitions → *SubClassDefinitions*
| *ClassDefinition*

The first type of production defines a class and its subclasses.

SubClassDefinitions → *SubClassDefinition* | *SubClassDefinition* { | *SubClassDefinition* }
SubClassDefinition → `%` *ClassComment* `%` *Identifier*
| `%` *ClassComment* `%` *Identifier* `empty`
| `%` *ClassComment* `%` *Identifier* `%` *InstanceVariableDefinitions*

The second type of production defines a single class.

ClassDefinition → [% *InstanceVariableDefinitions*]

For each instance variable, we specify its name and a description of the instance variable in the form of a comment and its type.

InstanceVariableDefinitions → *Identifier* : *InstanceVariableComment* % *Identifier*
| *Identifier* : *InstanceVariableComment* , *InstanceVariableDefinitions Identifier*

The productions should satisfy the five conditions discussed in Section 3.2.2.

3.4 Generated classes

Next, we present some of the classes generated from the above Grappa specification. We start with the class `Node`. This class is the root of the generated inheritance hierarchy of classes. In a `Node`, we only keep track of the position within the original source file of the token represented by the `Node` (see, for example, [2, page 101]).

```
package expression;

/**
 * Node of a syntax tree.
 */
abstract class Node
{
    private int left;
    private int right;

    /**
     * Node of a syntax tree.
     *
     * @param left position in input stream of left side of token represented by the node.
     * @param right position in input stream of right side of token represented by the node.
     */
    Node(int left, int right)
    {
        this.left = left;
        this.right = right;
    }
}
```

This class is extended by the class `Expression`.

```
package expression;

/**
 * Arithmetic expression over integers.
 */
abstract class Expression extends Node
{
    /**
     * Expression.
     *
     * @param left position in input stream of left side of token represented by the node.
     * @param right position in input stream of right side of token represented by the node.
```

```
     */
    Expression(int left, int right)
    {
        super(left, right);
    }
}
```

This class is extended by a number of classes including `Addition`

```
package expression;

/**
 * Addition.
 */
class Addition extends Expression
{
    private Expression first;
    private Expression second;

    /**
     * Addition.
     *
     * @param left position in input stream of left side of token represented by the node.
     * @param right position in input stream of right side of token represented by the node.
     * @param first first expression of the addition.
     * @param second second expression of the addition.
     */
    Addition(int left, int right, Expression first, Expression second)
    {
        super(left, right);
        this.first = first;
        this.second = second;
    }
}
```

and `Infinity`.

```
package expression;

/**
 * Infinity.
 */
class Infinity extends Expression
{
    /**
     * Infinity.
     *
     * @param left position in input stream of left side of token represented by the node.
     * @param right position in input stream of right side of token represented by the node.
     */
    Infinity(int left, int right)
    {
        super(left, right);
    }
}
```

The expression `5 + 3 * ∞` is represented by the sequence of tokens $\texttt{NUMBER}_5$ `ADD` $\texttt{NUMBER}_3$ `MULTIPLY INFINITY` and by the `Expression` object

```
new Addition(0, 8, new Number(0, 0, new Integer(5)),
                   new Multiplication(4, 8, new Number(4, 4, new Integer(3)),
                                            new Infinity(8, 8)))
```

Since the class `Infinity` represents the token `INFINITY`, we keep track of the position of this token within the original source file in the instance variables `left` and `right` of the corresponding `Infinity` object. The class `EmptyList` does not represent a token and, therefore, no positional information is needed in an `EmptyList` object.

```
package expression;

/**
 * Empty list of expressions.
 */
class EmptyList extends List
{
    /**
     * Empty list of expressions.
     */
    EmptyList() {}
}
```

We introduced the keyword `empty` to distinguish between classes like `Infinity` and `EmptyList`.

3.5 Building syntax trees with CUP

As we already mentioned in the introduction, we can add fragments of Java code to the productions of a CUP specification. Such a Java snippet, which is also known as action code, is executed at the point when the right hand side of the production has been recognized. For example,

```
Expression ::= ...
             | NUMBER
               {: System.out.println("number"); :}
             ;
```

prints `number` whenever a `NUMBER` token is recognized.

In the example CUP specification presented in Section 2, we associated an `Integer` object with the terminal `NUMBER`. This object can be accessed using a label. For example,

```
Expression ::= ...
             | NUMBER:value
               {: System.out.println(value); :}
             ;
```

labels the object associated with `NUMBER` as `value`. This label can be used in the action code. As a result, whenever a `NUMBER` token is recognized, its `Integer` value is printed.

CUP allows us to associate objects to nonterminals as well. We associate an `Expression` object with the nonterminal `Expression`. This object represents the parse tree rooted at the nonterminal. This association is specified as follows.

```
nonterminal Expression Expression;
```

In the action code we can associate a `Number` object, which is an `Expression` object according to the inheritance hierarchy, with the `Expression` nonterminal as follows.

```
Expression ::= ...
             | NUMBER:value
               {: RESULT = new Number(..., ..., value); :}
             ;
```

Whenever a `NUMBER` token is recognized, a `Number` object, containing the `Integer` object `value`, is created and this `Number` object is associated with the `Expression` nonterminal. Hence, the nonterminal at the left hand side of a production is always implicitly labelled as `RESULT`.

Besides an `Integer` object that represents the actual integer value, a `Number` object also contains two integers that represent the position of the `left` and `right` most character of the token in the original input stream. We assume that the lexer keeps track of these two integers for each token. (A lexer generated by, for example, JFlex supports this feature.) These integers can be accessed in the action code as follows.

```
Expression ::= ...
             | NUMBER:value
               {: RESULT = new Number(valueleft, valueright, value); :}
             ;
```

Even if we do not associate an object with a token, we can still obtain its position by labelling the token and postfixing the label with `left` and `right`, like in

```
Expression ::= ...
             | LPAREN:LPARENfirst Expression:exp RPAREN:RPARENlast
               {: RESULT = new Parentheses(LPARENfirstleft, RPARENlastright, exp); :}
               ...
```

We can add action code to the production presented in Section 2.4 resulting in the following decorated production.

```
List ::=
         {: RESULT = new EmptyList(); :}

       | Expression:head List:tail
         {: RESULT = new NonemptyList(headleft, tailright, head, tail); :}
       ;

Expression ::= Expression:first ADD Expression:second
               {: RESULT = new Addition(firstleft, secondright, first, second); :}

             | Expression:first SUBTRACT Expression:second
               {: RESULT = new Subtraction(firstleft, secondright, first, second); :}

             | Expression:first MULTIPLY Expression:second
               {: RESULT = new Multiplication(firstleft, secondright, first, second); :}

             | Expression:first DIVIDE Expression:second
               {: RESULT = new Division(firstleft, secondright, first, second); :}

             | LPAREN:LPARENfirst Expression:exp RPAREN:RPARENlast
               {: RESULT = new Parentheses(LPARENfirstleft, RPARENlastright, exp); :}

             | NUMBER:value
               {: RESULT = new Number(valueleft, valueright, value); :}

             | INFINITY:INFINITYlast
               {: RESULT = new Infinity(INFINITYlastleft, INFINITYlastright); :}
             ;
```

The parser generated from this specification not only recognizes valid sequences of tokens. It also builds a `List` or `Expression` object that represents the parse tree for the sequence of tokens. This object can subsequently be used in the semantic analysis phase of a compiler.

3.6 Grappa implementation

Grappa has originally been developed by Antia in Perl. The current implementation, written in Java, uses CUP to parse the Grappa specification. Action code added to each production is exploited to generate the desired results. In order to keep the CUP specification clean and easy to maintain, the code snippets only contain calls to the methods of an imported class named `Grappa`. For example, the CUP production for the grammar rule

Specification → *Declarations Productions*

would look like

```
Specification ::= Declarations:decls Productions:prods
                  {: RESULT = Grappa.specification(decls, prods); :}
```

where the labels `decls` and `prods` are associated with strings. Those strings represent the declarations and the productions. The method `Grappa.specification` combines the two strings into a string representing the specification. This string contains all the information for the classes to be generated. The classes can easily be extracted from the string.

4 Corretto

As we have seen in the foregoing sections, we can exploit the tools CUP and Grappa to automatically generate from a CUP and a Grappa specification a parser that not only recognizes a sequence of tokens but that also builds a concrete syntax tree. If we compare the CUP and Grappa specification for our simple example of (lists of) arithmetic expressions, we note that the productions contain similar information. If we leave out the terminals and the action code from the CUP specification and we remove the comments, the subclasses and the instance variables from the Grappa productions, then we end up with roughly the same productions. As we will show, the action code in the CUP specification can be automatically generated from the Grappa specification. Therefore, we propose to combine the CUP and Grappa specification into one specification. This saves us from creating two specifications and keeping them consistent. Furthermore, we do not have to write the action code by hand as it can be generated automatically.

The tool Corretto extracts from the common specification, which from now on is called the Corretto specification, a CUP and a Grappa specification. These two specifications can subsequently be used to produce a parser and a hierarchy of classes to represent concrete syntax trees.

The rest of this section is organized as follows. First, we discuss the syntax of a Corretto specification. Second, we describe how the CUP and Grappa specification are extracted from a Corretto specification. Next, we give an overview of our implementation of the tool Corretto. Finally, we discuss related work.

4.1 Corretto specification

A Corretto specification should contain enough information so that we can extract a CUP and a Grappa specification from it. While designing the syntax to capture this information, we focused on the following three objectives. First of all, we want a Corretto specification to be easy to read and easy to write. Secondly, the Corretto tool should be easy to implement. Thirdly, we do not want a Corretto specification to contain a lot of redundant information. These three objectives are conflicting. For example, unlike a CUP specification, the nonterminals need not be declared in a Corretto specification since they can be inferred from the productions. However, removing this redundant information from the specification will increase the complexity of the Corretto tool and may lead to a less readable specification, but may make it easier to write the specification. Therefore, trade-offs need to be made.

In the design of the syntax of Corretto specifications we have tried to stay close to the CUP syntax. This has several advantages. First of all, many prospect users of Corretto will probably be familiar with CUP. Having a similar syntax will make it easier for those users to read and write Corretto specifications. Furthermore, for many languages, like, for example, Java and XML, a CUP specification has been developed. These CUP specifications can be extended fairly easily to Corretto specifications. Finally, CUP and Corretto having a similar syntax will make it easy to extract a CUP specification from a Corretto specification.

Like a CUP specification, a Corretto specification has four sections. Next, we present these sections in the order in which they should appear in a Corretto specification. Again, we will use the (lists of) arithmetic expressions over integers as our running example.

4.1.1 Package and import specifications

The first section is optional. This section plays the same role and has the same syntax as the corresponding section in a CUP or Grappa specification. Since this section only contains (at most) one package declaration, the extracted CUP and Grappa specification both inherit this declaration. Therefore, the generated parser and the generated syntax tree classes are part of the same package. For our example, this first section amounts to

```
package expression;
```

4.1.2 Terminals

The second part is required. It contains the declaration of the terminal symbols. Each terminal consists of a sequence of Java letter and Java digits, the first of which must be a Java letter, and it is different from a Java keyword, a CUP keyword (see [6]), `true`, `false`, `null` and the Grappa keyword `empty`. Each typed terminal should be different from the name of any class in the package `java.lang` or in any of the imported packages. Its type should be part of the package `java.lang` or any of the imported packages.

In contrast to a CUP specification, a Corretto specification does not contain a list of the nonterminal symbols. Every symbol used in the productions that is not declared in this section as a terminal is assumed to be a nonterminal. Such a nonterminal should appear exactly once on the left hand side of a production.

For the arithmetic expressions, the terminals can be declared as follows.

```
terminal ADD, SUBTRACT, MULTIPLY, DIVIDE, LPAREN, RPAREN, INFINITY;
terminal Integer NUMBER;
```

4.1.3 Precedence and associativity specifications

This section is optional as well. It is identical to the third section of a CUP specification. For our example, this section amounts to

```
precedence left ADD, SUBTRACT;
precedence left MULTIPLY, DIVIDE;
```

4.1.4 Productions

The final section is required. It consists of the productions. For the (lists of) arithmetic expressions, we have the following productions.

```
% List of expressions %
List : % Empty list of expressions %
       EmptyList

     | % Nonempty list of expressions %
       NonemptyList
       % head : first expression of the list,
         tail : rest of the list %
```

```
        Expression List
      ;

  % Arithmetric expression over integers %
  Expression : % Addition %
               Addition
               % first : first expression of the addition,
                 second : second expression of the addition %
               Expression ADD Expression

             | % Subtraction %
               Subtraction
               % first : first expression of the subtraction,
                 second : second expression of the subtraction %
               Expression SUBTRACT Expression

             | % Multiplication %
               Multiplication
               % first : first expression of the multiplication,
                 second : second expression of the multiplication %
               Expression MULTIPLY Expression

             | % Division %
               Division
               % first : first expression of the division,
                 second : second expression of the division %
               Expression DIVIDE Expression

             | % Parenthesized expression %
               Parentheses
               % exp : expression %
               LPAREN Expression RPAREN

             | % Infinity %
               Infinity
               INFINITY

             | % Number %
               Number
               % value : integer value %
               NUMBER
             ;
```

Recall that there are two types of Grappa productions. The first type defines a class and its subclasses, whereas the second type defines a class (without any subclasses). The above Corretto production corresponds to a Grappa production of the first type. An example of a Corretto production corresponding to a Grappa production of the second type is given below.

```
  % Function declaration %
  FunctionDeclaration : % name : name of the function,
                          parameters : parameters of the function,
                          type : return type of the function,
                          body : body of the function %
                        FUNCTION Identifier LPAREN FieldList RPAREN Type EQ Expression
                      ;
```

where `FUNCTION`, `LPAREN`, `RPAREN` and `EQ` are tokens representing `function`, `(`, `)` and `=`, respectively. The productions for the nonterminals `FieldList`, `Type` and `Expression` and the specification of the terminal `Identifier` can be found in Section 5.1.

Next, we have a more detailed look at the two types of productions. In general, a production of the first type is of the form

```
% comment for C %
C : % comment for S_1 %
      S_1
      % v_{1,1} : comment for v_{1,1} ,
        ⋮
        v_{1,n_1} : comment for v_{1,n_1} %
      X_{1,1} ... X_{1,k_1}
    ⋮
    | % comment for S_m %
      S_m
      % v_{m,1} : comment for v_{m,1} ,
        ⋮
        v_{m,n_m} : comment for v_{m,n_m} %
      X_{m,1} ... X_{m,k_m}
    ;
```

where $m \geq 2$, $n_i \geq 0$ and $k_i \geq 0$ for $i = 1, \ldots, m$. In the above production, C is a nonterminal and each $X_{i,j}$ is either a nonterminal or a terminal. It is required that, for $i = 1, \ldots, m$, n_i equals the number of $X_{i,j}$'s that are either a nonterminal or a typed terminal. If all $X_{i,j}$'s are untyped terminals, that is, $n_i = 0$, then the right hand side is of the form

```
...
| % comment for S_i %
  S_i
  X_{i,1} ... X_{i,k_i}
| ...
```

For an empty production we have that $k_i = 0$, and hence $n_i = 0$. In that case the right hand side is of the form

```
...
| % comment for S_i %
  S_i
| ...
```

Since the terminals are part of the Corretto production, the productions for `Infinity` and `EmptyList` are structurally different. Therefore, we do not have to introduce the keyword `empty` as we did in the Grappa specification.

Let us have a look at the different ingredients of the above Corretto production and the role these ingredients will play in the corresponding CUP and Grappa productions. Here, we just give a brief overview. More details will be provided in Section 4.3. The C of the Corretto production will be the nonterminal on the left hand side of the CUP production and the class name on the left hand side of the Grappa production. The S_i's are the names of the subclasses of the class named C. These do not play a role in the CUP production. The $v_{i,j}$'s are the labels of the nonterminals and typed terminals of the right hand side of the CUP production and the names of the instance variables of the class S_i generated by Grappa. The $X_{i,j}$'s are the nonterminals and terminals of the right hand side of the CUP production. Only the $X_{i,j}$'s that correspond to a nonterminal or a typed terminal play the role of type of some instance variable $v_{i',j'}$ in Grappa. For example, consider the following part of a Corretto production.

```
% name : name of the variable,
  type : type of the variable,
  exp : expression representing the initial value of the variable %
VAR Identifier Type ASSIGN Expression
```

Assume that `Type` and `Expression` are nonterminals and that the terminal `Identifier` has type `String`. Then the instance variables named `name`, `type` and `exp` have types `String`, `Type` and `Expression`, respectively.

A production of the second type is of the form

% comment for C %
C : % v_1 : comment for instance variable v_1 ,
$\vdots$
v_n : comment for instance variable v_n %
$X_1 \ldots X_k$
;

where $n \geq 0$ and $k \geq 0$. Again, C is a nonterminal and each X_j is either a nonterminal or a terminal. Also for this type of production it is required that n equals the number of X_j's that are either a nonterminal or a typed terminal.

Given a set of productions, let $\mathcal{C}$ be the collection of nonterminals/class names that occur as C in any of the productions, let $\mathcal{S}$ be the collection of class names that occur as S_i in any of the productions, let $\mathcal{X}$ be the collection of nonterminals and terminals/class names that occur as $X_{i,j}$ or X_j in any of the productions, and let $\mathcal{V}$ be the collection of labels/instance variable names that occur in any of the productions. A set of productions should satisfy the following conditions.

1. $\mathcal{C}$ does not contain any duplicates, $\mathcal{S}$ does not contain any duplicates, and $\mathcal{C}$ and $\mathcal{S}$ do not have any name in common. This prevents us from assigning the same name to different classes.

2. Each nonterminal/class name in $\mathcal{C}$ consists of a sequence of Java letters and Java digits, the first of which must be a Java letter, and the name is different from a Java keyword, a CUP keyword, `true`, `false`, `null` and the Grappa keyword `empty`. This condition ensures that the names of the generated classes are valid according to the Java language specification and that the nonterminals of the generated CUP specification are not CUP keywords.

3. Each class name in $\mathcal{S}$ consists of a sequence of Java letters and Java digits, the first of which must be a Java letter, and the name is different from a Java keyword, `true`, `false`, `null` and the Grappa keyword `empty`. This condition ensures that the names of the generated subclasses are valid according to the Java language specification.

4. Each class name in $\mathcal{C}$ or $\mathcal{S}$ is different from the name of any class in the package `java.lang` or in any of the imported packages. This avoids confusion between generated classes and imported classes.

5. Each label/instance variable name in $\mathcal{V}$ consists of a sequence of Java letters and Java digits, the first of which must be a Java letter, and the name is different from a Java keyword, a CUP keyword, `true`, `false`, `null` and the Grappa keyword `empty`. This condition ensures that the names of the instance variables of the generated classes are valid according to the Java language specification and that the labels of the generated CUP specification are not CUP keywords.

6. The collections of labels/instance variables $v_{i,1}, \ldots, v_{i,n_i}$ and $v_1, \ldots, v_n$ do not contain any duplicates. This ensures that the names of the CUP labels/instance variables of a generated class are all different.

7. For each X in $\mathcal{X}$, either X occurs in $\mathcal{C}$ or X is declared as a terminal. That is, X is either a nonterminal or a terminal.

The current implementation of Corretto does not check these conditions.

4.2 Corretto grammar

Below, we present the grammar defining the syntax of Corretto specifications. In the grammar, we use *Identifier* for names of nonterminals, terminals, labels, classes, instance variables and packages. Such a name consists of a sequence of Java letters and Java digits, the first of which must be a Java letter, and the name is different from a Java keyword (see [5, § 3.9]), a CUP keyword (see [6]), `true`, `false`, `null` and the Grappa keyword `empty`. A *ClassComment* consists of a sequence of characters different from `%` and an *InstanceVariableComment* consists of a sequence of characters different from `%`, `:` and `,`.

A Corretto specification has four parts: package and import declarations, terminals, precedence and associativity declarations and productions. The first and the third part are optional.

Specification →
[*PackageAndImportDeclarations*] *Terminals* [*PrecedenceAndAssociativityDeclarations*] *Productions*

The first part may contain a package declaration and one or more import declarations.

PackageAndImportDeclarations	→	[*PackageDeclaration*] { *ImportDeclaration* }
PackageDeclaration	→	`package` *Identifier* { `.` *Identifier* } `;`
ImportDeclaration	→	`import` *Identifier* { `.` *Identifier* } [`.*`] `;`

The second part consists of one of more declarations of terminals.

Terminals	→	*TerminalDeclaration* { *TerminalDeclaration* }
TerminalDeclaration	→	`terminal` [*Identifier*] *Identifier* { `,` *Identifier* } `;`

The third part may contain one or more precedence and associativity declarations.

PrecedenceAndAssociativityDeclarations	→	{ *PrecedenceAndAssociativityDeclaration* }
PrecedenceAndAssociativityDeclaration	→	`precedence left` *Identifier* { `,` *Identifier* } `;`
	|	`precedence right` *Identifier* { `,` *Identifier* } `;`
	|	`precedence nonassoc` *Identifier* { `,` *Identifier* } `;`

The fourth and final part consists of the productions. There are two types of productions. They only differ in their right hand sides.

Productions	→	*Production* { *Production* }
Production	→	`%` *ClassComment* `%` *Identifier* `:` *RightHandSides* `;`
RightHandSides	→	$RightHandSides_1$
	|	$RightHandSide_2$

The right hand side of the first type of production is of the form

$RightHandSides_1$	→	$RightHandSide_1$ | $RightHandSide_1$ { | $RightHandSide_1$ }
$RightHandSide_1$	→	`%` *ClassComment* `%` *Identifier* [[`%` *Labels* `%`] *Identifier* { *Identifier* }]

The right hand side of the second type of production is of the form

$RightHandSide_2$	→	[[`%` *Labels* `%`] *Identifier* { *Identifier* }]

For each label/instance variable, we specify its name and a description of the instance variable in the form of a comment.

Labels	→	*Identifier* `:` *InstanceVariableComment*
	|	*Identifier* `:` *InstanceVariableComment* `,` *Labels*

4.3 Generated specifications

Below we sketch how a CUP and a Grappa specification are extracted from a Corretto specification.

4.3.1 CUP specification

As we described in Section 2, a CUP specification consists of four sections. The first section, consisting of package and import declarations, is copied verbatim from the Corretto specification. The second section contains the declaration of the nonterminals and terminals. The terminal declarations of the Corretto specification are copied verbatim to the CUP specification. For each Corretto production with left hand side C, we introduce the nonterminal declaration

`nonterminal` C C;

The third section consists of precedence and associativity declarations. These are copied verbatim from the Corretto specification. The fourth and final section contains the productions. Each Corretto production gives rise to a CUP production in the following way. A Corretto production of the form

% comment for C %
C : % comment for S_1 %
S_1
% $v_{1,1}$: comment for $v_{1,1}$,
$\vdots$
v_{1,n_1} : comment for v_{1,n_1} %
$X_{1,1} \ldots X_{1,k_1}$
$\vdots$
| % comment for S_m %
S_m
% $v_{m,1}$: comment for $v_{m,1}$,
$\vdots$
v_{m,n_m} : comment for v_{m,n_m} %
$X_{m,1} \ldots X_{m,k_m}$
;

gives rise to a CUP production

C ::= $X_{1,1} \ldots X_{1,k_1}$
$\vdots$
| $X_{m,1} \ldots X_{m,k_m}$
;

decorated with labels and action code. How to add the labels and action code is discussed below. A Corretto production of the form

% comment for C %
C : % v_1 : comment for instance variable v_1 ,
$\vdots$
v_n : comment for instance variable v_n %
$X_1 \ldots X_k$
;

is transformed into a CUP production

C ::= $X_1 \ldots X_k$
;

decorated with labels and action code.

Labels Of the symbols on the right hand side of the generated CUP production, the first and the last symbol, all the nonterminals and all the typed terminals are labelled. Let us first look at a special case. If the first (last) symbol is the untyped terminal X, then the first (last) symbol is labelled with X`first` (X`last`). If the right hand side consists of a single untyped terminal X then the terminal is labelled X`last`.

Given

% $v_{i,1}$: comment for $v_{i,1}$,
$\vdots$
v_{i,n_i} : comment for v_{i,n_i} %
$X_{i,1} \ldots X_{i,k_i}$

the nonterminals and typed terminals of $X_{i,1}, \ldots, X_{i,k_i}$ are labelled with $v_{i,1}, \ldots, v_{i,n_i}$. For example,

```
% name : name of the variable,
  type : type of the variable,
  exp : expression representing the initial value of the variable %
VAR Identifier COLON Type ASSIGN Expression
```

gives rise to the labelling

```
VAR:VARfirst Identifier:name COLON Type:type ASSIGN Expression:exp
```

Action code From

% comment for S_i %
S_i
% $v_{i,1}$: comment for $v_{i,1}$,
$\vdots$
v_{i,n_i} : comment for v_{i,n_i} %
$X_{i,1} \ldots X_{i,k_i}$

we extract the action code

{: RESULT = new S_i(*f*`left`, ℓ`right`, $v_{i,1}$, ..., v_{i,n_i}); :}

where f is the label of $X_{i,1}$ and ℓ is the label of X_{i,k_i}. Note that *f*`left` (ℓ`right`) is the position of the left (right) most character of $X_{i,1}$ (X_{i,k_i}) in the original input stream, and hence the left (right) most character of $X_{i,1} \ldots X_{i,k_i}$. For example,

```
% Variable declaration %
VariableDeclaration
% name : name of the variable,
  type : type of the variable,
  exp : expression representing the initial value of the variable %
VAR Identifier COLON Type ASSIGN Expression
```

amounts to

```
{: RESULT = new VariableDeclaration(VARfirstleft, expright, name, type, exp); :}
```

In the special case that $k_i = 0$ we produce the action code

{: RESULT = new S_i(); :}

A Corretto production

% comment for C %
C : % v_1 : comment for instance variable v_1 ,
$\vdots$
v_n : comment for instance variable v_n %
$X_1 \ldots X_k$
;

gives rise to the action code

{: RESULT = new C(`f`left, `ℓ`right, v_1, ..., v_n); :}

where f is the label of X_1 and ℓ is the label of X_k.

4.3.2 Grappa specification

A Grappa specification consists of package and import declarations and productions. The package and import declaration section is copied verbatim from the Corretto specification. Each Corretto production, as described in Section 4.1.4, is turned into a Grappa production in the following way. From each sequence of nonterminals and terminals $X_{i,1} \ldots X_{i,k_i}$ the untyped terminals are removed and the typed terminals are replaced with their type. For example, the sequence

```
VAR Identifier COLON Type ASSIGN Expression
```

is turned into the sequence

```
String Type Expression
```

(recall that the type of `Identifier` is `String`). If all $X_{i,j}$'s are untyped terminals then an empty sequence is the result. In the special case that the sequence of $X_{i,j}$'s is empty the keyword `empty` is used.

4.4 Corretto implementation

Similar to Grappa, Corretto is implemented in Java and uses CUP to parse its specification. The action code attached to each CUP production follows the same pattern as well. It only contains calls to the methods of two external classes, named `CUP` and `Grappa`. For Corretto, the CUP production for

PrecedenceAndAssociativityDeclaration → `precedence left` *Identifier* { , *Identifier* } ;
| `precedence right` *Identifier* { , *Identifier* } ;
| `precedence nonassoc` *Identifier* { , *Identifier* } ;

would look like

```
PrecedenceAndAssociativityDeclaration ::=
  PRECEDENCE LEFT IdentifierList:list SEMI
  {: RESULT = pair(CUP.precedenceAndAssociativityDeclarationLeft(list[0]),
                   Grappa.precedenceAndAssociativityDeclarationLeft(list[1])); :}

| PRECEDENCE RIGHT IdentifierList:list SEMI
  {: RESULT = pair(CUP.precedenceAndAssociativityDeclarationRight(list[0]),
                   Grappa.precedenceAndAssociativityDeclarationRight(list[1])); :}

| PRECEDENCE NONASSOC IdentifierList:list SEMI
  {: RESULT = pair(CUP.precedenceAndAssociativityDeclarationNonassoc(list[0]),
                   Grappa.precedenceAndAssociativityDeclarationNonassoc(list[1])); :}
```

A pair of strings is associated with the label `list`. The first element of this pair, `list[0]`, is the string representation of the list of identifiers for the CUP specification to be generated. Similarly, the second element, `list[1]`, is used to generate the Grappa specification. The classes `CUP` and `Grappa` contain methods, like `precedenceAndAssociativityDeclarationLeft`, to generate the resulting CUP and Grappa specifications. While parsing the Corretto specification, the untyped terminals, the typed terminals and the nonterminals are stored in suitable data structures. These are needed in the process of generating the CUP specification.

4.5 Related tools

Many parser generators that produce parsers written in Java have been developed. We already discussed CUP in Section 2. As far as we know, the only other parser generators that produce a parser which also builds a concrete syntax tree are ANTLR [9], JavaCC [10] in combination with either JJTree [11] or JTB [12], and SableCC [3].

Specifications for our simple arithmetic expressions for these other tools are very similar to our Corretto specification. For example, the SableCC specification roughly looks as follows.

```
expression = {addition} addition
           | {subtraction} subtraction
           | {multiplication} multiplication
           | {division} division
           | {parentheses} parentheses
           | {number} number;

addition       = [first]:expression add [second]:expression;
subtraction    = [first]:expression subtract [second]:expression;
multiplication = [first]:expression multiply [second]:expression;
division       = [first]:expression divide [second]:expression;
parentheses    = lparen expression rparen;
```

Note that our Corretto specification is more verbose than the above SableCC specification. Also the ANTLR specification is less verbose than ours. However, the additional annotations allow us to generate code with Javadoc comments.

One of the advantages of ANTLR and SableCC over Corretto is that they automatically generate tree walkers. These tree walkers provide ways to systematically traverse syntax trees. In order the traverse a syntax tree using such a tree walker, the internal data of the nodes of the tree is exposed. Hence, data encapsulation is violated. The tree walkers of SableCC often manipulate global instance variables and hence again violate the object oriented encapsulation law. Furthermore, the tree walkers use downcasting and hence have a risk of runtime failure. JavaCC in combination with JJTree or JTB provide tree nodes that implement the visitor design pattern (see, for example, [4, pages 331–350] for a detailed discussion of this design pattern). These visitors can be used to traverse syntax trees. This design pattern also exposes the internal data of the nodes of the syntax tree. Furthermore, downcasting is used. The classes generated by Grappa to represent syntax trees do not expose the internal data and traversal code can be added to these classes that does not use downcasting. For more details we refer the reader to [1].

The Corretto syntax is closer to the CUP syntax than the SableCC syntax is. Hence, a CUP specification can be adapted to a Corretto specification more easily than a SableCC specification can. Since for many languages a CUP specification has been developed, we believe that this is can be an advantage of using Corretto over SableCC. ANTLR specifications have been designed for a number of languages including Java and HTML (but not, for example, XML, as far as we know). A large variety of languages, including Java, HTML and XML, have been specified in JavaCC.

5 Example

In Section 4.1, we already presented an example of a Corretto specification for arithmetic expressions over integers. The corresponding CUP and Grappa specifications were given in Section 2 and Section 3.2, respec-

tively. In this section, we present a more elaborate Corretto specification and the corresponding CUP and Grappa specifications for a language very similar to the one studied in [2]. For a detailed discussion of the language we refer the reader to [2, Appendix A].

5.1 Corretto specification

```
package tiger;

terminal LPAREN, RPAREN, LBRACK, RBRACK, LBRACE, RBRACE;
terminal ADD, SUBTRACT, MULTIPLY, DIVIDE;
terminal EQ, NEQ, LT, LE, GT, GE;
terminal AND, OR;
terminal ASSIGN, IF, THEN, ELSE, WHILE, DO, FOR, TO, LET, IN, END, BREAK, NIL;
terminal FUNCTION, TYPE, VAR;
terminal ARRAY, OF;
terminal COMMA, COLON, SEMICOLON, DOT;

terminal String Identifier;
terminal String StringExp;
terminal Integer IntegerExp;

precedence nonassoc ASSIGN;
precedence left OR;
precedence left AND;
precedence nonassoc EQ, NEQ, LT, LE, GT, GE;
precedence left ADD, SUBTRACT;
precedence left MULTIPLY, DIVIDE;

% Expression %
Expression : % L-value expression %
             LValueExpression
             % lValue : l-value %
             TYPE LValue TYPE

           | % Nil expression %
             Nil
             NIL

           | % Integer expression %
             IntegerExpression
             % value : integer value %
             IntegerExp

           | % String expression %
             StringExpression
             % value : string value %
             StringExp

           | % Function call expression %
             FunctionCall
             % name : name of the function,
               arguments : arguments of the function %
             Identifier LPAREN ExpressionList RPAREN
```

```
| % Conjunction %
  Conjunction
  % first : first expression of the conjunction,
    second : second expression of the conjunction %
  Expression AND Expression

| % Disjunction %
  Disjunction
  % first : first expression of the disjunction,
    second : second expression of the disjunction %
  Expression OR Expression

| % Addition %
  Addition
  % first : first expression of the addition,
    second : second expression of the addition %
  Expression ADD Expression

| % Subtraction %
  Subtraction
  % first : first expression of the subtraction,
    second : second expression of the subtraction %
  Expression SUBTRACT Expression

| % Multiplication %
  Multiplication
  % first : first expression of the multiplication,
    second : second expression of the multiplication %
  Expression MULTIPLY Expression

| % Division %
  Division
  % first : first expression of the division,
    second : second expression of the division %
  Expression DIVIDE Expression

| % Equality %
  Equality
  % first : first expression of the equality expression,
    second : second expression of the equality expression %
  Expression EQ Expression

| % Inequality %
  Inequality
  % first : first expression of the inequality expression,
    second : second expression of the inequality expression %
  Expression NEQ Expression

| % Less than %
  LessThan
  % first : first expression of the comparison,
    second : second expression of the comparison %
```

```
  Expression LT Expression

| % Less than or equal %
  LessThanOrEqual
  % first : first expression of the comparison,
    second : second expression of the comparison %
  Expression LE Expression

| % Greater than %
  GreaterThan
  % first : first expression of the comparison,
    second : second expression of the comparison %
  Expression GT Expression

| % Greater or equal %
  GreaterThanOrEqual
  % first : first expression of the comparison,
    second : second expression of the comparison %
  Expression GE Expression

| % Record creation %
  RecordCreation
  % type : type of the record,
    fields : list of field names and expressions %
  Identifier LBRACE FieldExpressionList RBRACE

| % Assignment expression %
  Assignment
  % lValue : l-value of the assignment,
    exp : expression of the assignment %
  LValue ASSIGN Expression

| % If then else expression %
  IfThenElse
  % condition : condition of the if then else expression,
    thenClause : then clause of the if then else expression,
    elseClause : else clause of the if then else expression %
  IF Expression THEN Expression ELSE Expression

| % If then expression %
  IfThen
  % condition : condition of the if then else expression,
    thenClause : then clause of the if then else expression %
  IF Expression THEN Expression

| % While loop expression %
  WhileLoop
  % condition : condition of the while loop expression,
    body : body of the while loop expression %
  WHILE Expression DO Expression

| % For loop expression %
  ForLoop
```

```
            % variable : loop variable,
              initial : initial value of variable,
              final : final value ,
              body : body of the for loop expression %
            FOR Identifier ASSIGN Expression TO Expression DO Expression

          | % Break expression %
            Break
            BREAK

          | % Let statement expression  %
            Let
            % decs : list of declarations,
              exps : sequence of expressions %
            LET DeclarationList IN ExpressionSequence END

          | % Array creation %
            ArrayCreation
            % type : type of the array,
              size : size of the array,
              value: initial value for the array %
            Identifier LBRACK Expression RBRACK OF Expression

          | % Sequencing of expressions %
            Sequencing
            % exps : sequence of expressions %
            LPAREN ExpressionSequence RPAREN
          ;

% Location whose value can be read or assigned %
LValue : % Variable or parameter %
         Variable
         % name : name of the variable %
         Identifier

       | % Record field %
         RecordField
         % record : record value,
           field : field name %
         LValue DOT Identifier

       | % Array Subscript %
         ArraySubscript
         % name : name of array,
           exp : expression representing the index of the array %
         LValue LBRACK Expression RBRACK
       ;

% Nonempty list of expressions, separated by colons %
ExpressionList : % List consisting of one expression %
                 SingleExpressionList
                 % exp : single expression of the list %
                 Expression
```

```
              | % List consisting of more than one expression %
                MultipleExpressionList
                % head : first expression of the list,
                  tail : rest if the list %
                Expression COMMA ExpressionList
              ;

% Nonempty list of field names and expressions %
FieldExpressionList : % List consisting of one field name and expression %
                      SingleFieldExpressionList
                      % name : single field name of the list,
                        exp : single expression of the list %
                      Identifier EQ Expression

                    | % List consisting of more than one field name and expression %
                      MultipleFieldExpressionList
                      % name : name of the first field,
                        exp : expression of the first field,
                        tail : rest of the list of field names and expressions %
                      Identifier EQ Expression COMMA FieldExpressionList
                    ;

% List of type, variable, and function declarations %
DeclarationList : % Empty list of declarations %
                  EmptyDeclarationList

                | % Nonempty list of declarations %
                  NonemptyDeclarationList
                  % head : first declaration of the list,
                    tail : rest of the list %
                  Declaration DeclarationList
                ;

% Declaration of a type, a variable, or a function %
Declaration : % Type declaration %
              TypeDeclaration
              % name : name of the declared type,
                type : actual type of the declared type %
              TYPE Identifier EQ Type

            | % Variable declaration %
              VariableDeclaration
              % name : name of the variable,
                type : type of the variable,
                exp : expression representing the initial value of the variable %
              VAR Identifier COLON Type ASSIGN Expression

            | % Function declaration %
              FunctionDeclaration
              % name : name of the function,
                parameters : parameters of the function,
                type : return type of the function,
```

```
              body : body of the function %
            FUNCTION Identifier LPAREN FieldList RPAREN COLON Type EQ Expression
          ;

% Type %
Type : % Predefined type %
       PredefinedType
       % name : name of the predefined type %
       Identifier

     | % Record type %
       RecordType
       % fields : list of fields of the record type %
       LBRACE FieldList RBRACE

     | % Array type %
       ArrayType
       % type : type of the array %
       ARRAY OF Identifier
     ;

% List of fields of a record type %
FieldList : % Empty list of fields %
            EmptyFieldList

          | % Nonempty list of fields %
            NonemptyFieldList
            % name : name of the field,
              type : type of the field,
              tail : rest of the list %
            Identifier COLON Identifier FieldList
          ;

% Sequence of expressions, separated by semicolons %
ExpressionSequence : % Sequence consisting of a single expression %
                     SingleExpressionSequence
                     % exp : single expression of the sequence %
                     Expression

                   | % Sequence consisting of more than one expression %
                     MultipleExpressionSequence
                     % head : first expression of the sequence,
                       tail : rest of the sequence %
                     Expression SEMICOLON ExpressionSequence
                   ;
```

5.2 CUP specification

```
package tiger;

terminal LPAREN, RPAREN, LBRACK, RBRACK, LBRACE, RBRACE;
terminal ADD, SUBTRACT, MULTIPLY, DIVIDE;
terminal EQ, NEQ, LT, LE, GT, GE;
```

```
terminal AND, OR;
terminal ASSIGN, IF, THEN, ELSE, WHILE, DO, FOR, TO, LET, IN, END, BREAK, NIL;
terminal FUNCTION, TYPE, VAR;
terminal ARRAY, OF;
terminal COMMA, COLON, SEMICOLON, DOT;
terminal String Identifier;
terminal String StringExp;
terminal Integer IntegerExp;

nonterminal Expression Expression;
nonterminal LValue LValue;
nonterminal ExpressionList ExpressionList;
nonterminal NonemptyExpressionList NonemptyExpressionList;
nonterminal FieldExpressionList FieldExpressionList;
nonterminal NonemptyFieldExpressionList NonemptyFieldExpressionList;
nonterminal DeclarationList DeclarationList;
nonterminal Declaration Declaration;
nonterminal Type Type;
nonterminal FieldList FieldList;
nonterminal ExpressionSequence ExpressionSequence;
nonterminal NonemptyExpressionSequence NonemptyExpressionSequence;

precedence nonassign ASSIGN;
precedence left OR;
precedence left AND;
precedence nonassign EQ, NEQ, LT, LE, GT, GE;
precedence left ADD, SUBTRACT;
precedence left MULTIPLY, DIVIDE;

Expression ::=
  TYPE:TYPEfirst LValue:lValue TYPE:TYPElast
  {: RESULT = new LValueExpression(TYPEfirstleft, TYPElastright, lValue); :}

| NIL:NILlast
  {: RESULT = new Nil(NILlastleft, NILlastright); :}

| IntegerExp:value
  {: RESULT = new IntegerExpression(valueleft, valueright, value); :}

| StringExp:value
  {: RESULT = new StringExpression(valueleft, valueright, value); :}

| Identifier:name LPAREN ExpressionList:arguments RPAREN:RPARENlast
  {: RESULT = new FunctionCall(nameleft, RPARENlastright, name, arguments); :}

| Expression:first AND Expression:second
  {: RESULT = new Conjunction(firstleft, secondright, first, second); :}

| Expression:first OR Expression:second
  {: RESULT = new Disjunction(firstleft, secondright, first, second); :}

| Expression:first ADD Expression:second
  {: RESULT = new Addition(firstleft, secondright, first, second); :}
```

```
| Expression:first SUBTRACT Expression:second
  {: RESULT = new Subtraction(firstleft, secondright, first, second); :}

| Expression:first MULTIPLY Expression:second
  {: RESULT = new Multiplication(firstleft, secondright, first, second); :}

| Expression:first DIVIDE Expression:second
  {: RESULT = new Division(firstleft, secondright, first, second); :}

| Expression:first EQ Expression:second
  {: RESULT = new Equality(firstleft, secondright, first, second); :}

| Expression:first NEQ Expression:second
  {: RESULT = new Inequality(firstleft, secondright, first, second); :}

| Expression:first LT Expression:second
  {: RESULT = new LessThan(firstleft, secondright, first, second); :}

| Expression:first LE Expression:second
  {: RESULT = new LessThanOrEqual(firstleft, secondright, first, second); :}

| Expression:first GT Expression:second
  {: RESULT = new GreaterThan(firstleft, secondright, first, second); :}

| Expression:first GE Expression:second
  {: RESULT = new GreaterThanOrEqual(firstleft, secondright, first, second); :}

| Identifier:type LBRACE FieldExpressionList:fields RBRACE:RBRACElast
  {: RESULT = new RecordCreation(typeleft, RBRACElastright, type, fields); :}

| LValue:lValue ASSIGN Expression:exp
  {: RESULT = new Assignment(lValueleft, expright, lValue, exp); :}

| IF:IFfirst Expression:condition THEN Expression:thenClause ELSE Expression:elseClause
  {: RESULT = new IfThenElse(IFfirstleft, elseClauseright, condition, thenClause, elseClause); :}

| IF:IFfirst Expression:condition THEN Expression:thenClause
  {: RESULT = new IfThen(IFfirstleft, thenClauseright, condition, thenClause); :}

| WHILE:WHILEfirst Expression:condition DO Expression:body
  {: RESULT = new WhileLoop(WHILEfirstleft, bodyright, condition, body); :}

| FOR:FORfirst Identifier:variable ASSIGN Expression:initial TO Expression:final DO Expression:body
  {: RESULT = new ForLoop(FORfirstleft, bodyright, variable, initial, final, body); :}

| BREAK:BREAKlast
  {: RESULT = new Break(BREAKlastleft, BREAKlastright); :}

| LET:LETfirst DeclarationList:decs IN ExpressionSequence:exps END:ENDlast
  {: RESULT = new Let(LETfirstleft, ENDlastright, decs, exps); :}

| Identifier:type LBRACK Expression:size RBRACK OF Expression:value
```

```
  {: RESULT = new ArrayCreation(typeleft, valueright, type, size, value); :}

| LPAREN:LPARENfirst ExpressionSequence:exps RPAREN:RPARENlast
  {: RESULT = new Sequencing(LPARENfirstleft, RPARENlastright, exps); :}
;

LValue ::=
  Identifier:name
  {: RESULT = new Variable(nameleft, nameright, name); :}

| LValue:record DOT Identifier:field
  {: RESULT = new RecordField(recordleft, fieldright, record, field); :}

| LValue:name LBRACK Expression:exp RBRACK:RBRACKlast
  {: RESULT = new ArraySubscript(nameleft, RBRACKlastright, name, exp); :}
;

ExpressionList ::=
  Expression:exp
  {: RESULT = new SingleExpressionList(expleft, expright, exp); :}

| Expression:head COMMA ExpressionList:tail
  {: RESULT = new MultipleExpressionList(headleft, tailright, head, tail); :}
;

FieldExpressionList ::=
  Identifier:name EQ Expression:exp
  {: RESULT = new SingleFieldExpressionList(nameleft, expright, name, exp); :}

| Identifier:name EQ Expression:exp COMMA FieldExpressionList:tail
  {: RESULT = new MultipleFieldExpressionList(nameleft, tailright, name, exp, tail); :}
;

DeclarationList ::=

  {: RESULT = new EmptyDeclarationList(); :}

| Declaration:head DeclarationList:tail
  {: RESULT = new NonemptyDeclarationList(headleft, tailright, head, tail); :}
;

Declaration ::=
  TYPE:TYPEfirst Identifier:name EQ Type:type
  {: RESULT = new TypeDeclaration(TYPEfirstleft, typeright, name, type); :}

| VAR:VARfirst Identifier:name COLON Type:type ASSIGN Expression:exp
  {: RESULT = new VariableDeclaration(VARfirstleft, expright, name, type, exp); :}

| FUNCTION:FUNCTIONfirst Identifier:name LPAREN FieldList:parameters RPAREN COLON
  Type:type EQ Expression:body
  {: RESULT = new FunctionDeclaration(FUNCTIONfirstleft, bodyright, name, parameters, type, body); :}
;
```

```
Type ::=
  Identifier:name
  {: RESULT = new PredefinedType(nameleft, nameright, name); :}

| LBRACE:LBRACEfirst FieldList:fields RBRACE:RBRACElast
  {: RESULT = new RecordType(LBRACEfirstleft, RBRACElastright, fields); :}

| ARRAY:ARRAYfirst OF Identifier:type
  {: RESULT = new ArrayType(ARRAYfirstleft, typeright, type); :}
;

FieldList ::=

  {: RESULT = new EmptyFieldList(); :}

| Identifier:name COLON Identifier:type FieldList:tail
  {: RESULT = new NonemptyFieldList(nameleft, tailright, name, type, tail); :}
;

ExpressionSequence ::=
  Expression:exp
  {: RESULT = new SingleExpressionSequence(expleft, expright, exp); :}

| Expression:head SEMICOLON ExpressionSequence:tail
  {: RESULT = new MultipleExpressionSequence(headleft, tailright, head, tail); :}
;
```

5.3 Grappa specification

```
package tiger;

% Expression %
Expression :
  % L-value expression %
  LValueExpression
  % lValue : l-value %
  LValue

| % Nil expression %
  Nil

| % Integer expression %
  IntegerExpression
  % value : integer value %
  Integer

| % String expression %
  StringExpression
  % value : string value %
  String

| % Function call expression %
  FunctionCall
```

```
  % name : name of the function,
    arguments : arguments of the function %
  String ExpressionList

| % Conjunction %
  Conjunction
  % first : first expression of the conjunction,
    second : second expression of the conjunction %
  Expression Expression

| % Disjunction %
  Disjunction
  % first : first expression of the disjunction,
    second : second expression of the disjunction %
  Expression Expression

| % Addition %
  Addition
  % first : first expression of the addition,
    second : second expression of the addition %
  Expression Expression

| % Subtraction %
  Subtraction
  % first : first expression of the subtraction,
    second : second expression of the subtraction %
  Expression Expression

| % Multiplication %
  Multiplication
  % first : first expression of the multiplication,
    second : second expression of the multiplication %
  Expression Expression

| % Division %
  Division
  % first : first expression of the division,
    second : second expression of the division %
  Expression Expression

| % Equality %
  Equality
  % first : first expression of the equality expression,
    second : second expression of the equality expression %
  Expression Expression

| % Inequality %
  Inequality
  % first : first expression of the inequality expression,
    second : second expression of the inequality expression %
  Expression Expression

| % Less than %
```

```
  LessThan
  % first : first expression of the comparison,
    second : second expression of the comparison %
  Expression Expression

| % Less than or equal %
  LessThanOrEqual
  % first : first expression of the comparison,
    second : second expression of the comparison %
  Expression Expression

| % Greater than %
  GreaterThan
  % first : first expression of the comparison,
    second : second expression of the comparison %
  Expression Expression

| % Greater or equal %
  GreaterThanOrEqual
  % first : first expression of the comparison,
    second : second expression of the comparison %
  Expression Expression

| % Record creation %
  RecordCreation
  % type : type of the record,
    fields : list of field names and expressions %
  String FieldExpressionList

| % Assignment expression %
  Assignment
  % lValue : l-value of the assignment,
    exp : expression of the assignment %
  LValue Expression

| % If then else expression %
  IfThenElse
  % condition : condition of the if then else expression,
    thenClause : then clause of the if then else expression,
    elseClause : else clause of the if then else expression %
  Expression Expression Expression

| % If then expression %
  IfThen
  % condition : condition of the if then else expression,
    thenClause : then clause of the if then else expression %
  Expression Expression

| % While loop expression %
  WhileLoop
  % condition : condition of the while loop expression,
    body : body of the while loop expression %
  Expression Expression
```

```
| % For loop expression %
  ForLoop
  % variable : loop variable,
    initial : initial value of variable,
    final : final value,
    body : body of the for loop expression %
  String Expression Expression Expression

| % Break expression %
  Break

| % Let statement expression %
  Let
  % decs : list of declarations,
    exps : sequence of expressions %
  DeclarationList ExpressionSequence

| % Array creation %
  ArrayCreation
  % type : type of the array,
    size : size of the array,
    value : initial value for the array %
  String Expression Expression

| % Sequencing of expressions %
  Sequencing
  % exps : sequence of expressions %
  ExpressionSequence
;

% Location whose value can be read or assigned %
LValue :
  % Variable or parameter %
  Variable
  % name : name of the variable %
  String

| % Record field %
  RecordField
  % record : record value,
    field : field name %
  LValue String

| % Array Subscript %
  ArraySubscript
  % name : name of array,
    exp : expression representing the index of the array %
  LValue Expression
;

% Nonempty list of expressions, separated by colons %
ExpressionList :
```

```
  % List consisting of one expression %
  SingleExpressionList
  % exp : single expression of the list %
  Expression

| % List consisting of more than one expression %
  MultipleExpressionList
  % head : first expression of the list,
    tail : rest if the list %
  Expression ExpressionList
;

% Nonempty list of field names and expressions %
FieldExpressionList :
  % List consisting of one field name and expression %
  SingleFieldExpressionList
  % name : single field name of the list,
    exp : single expression of the list %
  String Expression

| % List consisting of more than one field name and expression %
  MultipleFieldExpressionList
  % name : name of the first field,
    exp : expression of the first field,
    tail : rest of the list of field names and expressions %
  String Expression FieldExpressionList
;

% List of type, variable, and function declarations %
DeclarationList :
  % Empty list of declarations %
  EmptyDeclarationList
  empty

| % Nonempty list of declarations %
  NonemptyDeclarationList
  % head : first declaration of the list,
    tail : rest of the list %
  Declaration DeclarationList
;

% Declaration of a type, a variable, or a function %
Declaration :
  % Type declaration %
  TypeDeclaration
  % name : name of the declared type,
    type : actual type of the declared type %
  String Type

| % Variable declaration %
  VariableDeclaration
  % name : name of the variable,
    type : type of the variable,
```

```
    exp : expression representing the initial value of the variable %
  String Type Expression

| % Function declaration %
  FunctionDeclaration
  % name : name of the function,
    parameters : parameters of the function,
    type : return type of the function,
    body : body of the function %
  String FieldList Type Expression
;

% Type %
Type :
  % Predefined type %
  PredefinedType
  % name : name of the predefined type %
  String

| % Record type %
  RecordType
  % fields : list of fields of the record type %
  FieldList

| % Array type %
  ArrayType
  % type : type of the array %
  String
;

% List of fields of a record type %
FieldList :
  % Empty list of fields %
  EmptyFieldList
  empty

| % Nonempty list of fields %
  NonemptyFieldList
  % name : name of the field,
    type : type of the field,
    tail : rest of the list %
  String String FieldList
;

% Sequence of expressions, separated by semicolons %
ExpressionSequence :
  % Sequence consisting of a single expression %
  SingleExpressionSequence
  % exp : single expression of the sequence %
  Expression

| % Sequence consisting of more than one expression %
  MultipleExpressionSequence
```

```
  % head : first expression of the sequence,
    tail : rest of the sequence %
  Expression ExpressionSequence
;
```

6 Conclusion

Grappa is a tool to generate Java classes for representing syntax trees. This tool was originally developed by Antia. In Section 3, we revisited the design and implementation of Grappa. We came up with a new syntax for Grappa specifications. We believe that this new syntax is more readable than the old one. Furthermore, we realized that a keyword, like `empty`, is needed to distinguish classes that represent no token from classes that represent a token. We made a detailed analysis of conditions, like the ones presented on page 8, that need to be satisfied for a Grappa specification to be valid. Also, we implemented Grappa in Java.

Motivated by the fact that CUP and Grappa specifications have a lot in common, we introduced Corretto. We developed the tool Corretto from scratch. In Section 4, we discussed its design, its syntax and its implementation in Java.

Although we made precise which conditions a Grappa/Corretto specification should satisfy to be valid, the current version of Grappa/Corretto does not check these conditions. This is left as future work. The next version of Grappa/Corretto should also provide meaningful error messages if a Grappa/Corretto specification does not correspond to the Grappa/Corretto grammar.

Providing editor support to make it easier to write Grappa and Corretto specifications is another project that may be considered in the future. For example, an emacs mode for editing Grappa and Corretto specifications might be useful.

Allowing to combine and reuse Grappa and Corretto specifications may be desirable. This could be achieved by adding some sort of import command to Grappa and Corretto specifications.

Although we have tested our tools Grappa and Corretto rigorously, all our test cases consisted of relatively small specifications (see, for example, the specification in Section 5). Developing a Corretto specification for the Java language may be a very good exercise. Such a Corretto specification will be considerably larger than any specification we have considered so far.

The current implementation of Corretto is available at

`http://www.cs.yorku.ca/~franck/research/corretto/`

and the current implementation of Grappa can be found out

`http://www.cs.yorku.ca/~franck/research/grappa/`

References

[1] D. Antia. *Semantic Analysis of Pict in Java.* Master's thesis, York University, Toronto, in preparation.

[2] A. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[3] E. Gagnon. *SableCC, An Object-Oriented Compiler Framework.* Master's thesis, McGill University, Montreal, 1998.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[5] J. Gosling, B. Joy, G.L. Steele and G. Bracha. *The Java Language Specification.* Addison Wesley, 2000.

[6] S. Hudson. CUP, LALR Parser Generator for Java. `www.cs.princeton.edu/~appel/modern/java/CUP`.

[7] S.C. Johnson. YACC: Yet Another Compiler Compiler. CS Technical Report 32, Bell Laboratories, Murray Hill, 1975.

[8] G. Klein. JFlex, The Fast Scanner Generator for Java. `www.jflex.de`.

[9] MageLang Institute. ANTLR. `www.antlr.org`.

[10] Metamata and Sun Microsystems. JavaCC. `www.webgain.com/products/java_cc`.

[11] Metamata and Sun Microsystems. JJTree. `www.webgain.com/products/java_cc/jjtree.html`.

[12] W. Wang, K. Tao and J. Palsberg. JTB: Java Tree Builder. `www.cs.purdue.edu/jtb`.

[13] D.A. Watt and D.F. Brown. *Programming Language Processors in Java.* Prentice-Hall, 1999.