Java to Scala

Cédric Lavanchy

January 11, 2009

Contents

Chapter 1

Goal of the project

The Scala programming language coexists with the Java language. Here, "coexist" means that a Scala project can contain both Scala and Java source files. The Scala compiler can compile Scala files that access elements defined in some Java files. Unfortunately, the Scala compiler does not produce the class file for these Java files. The way to compile a mixed project is:

1. Submit all the Java/Scala sources to the Scala compiler.[1]
2. Compile the Java sources (with the java compiler) using the Scala generated .class files.[1]
3. Compile again the Scala sources using the Java generated .class files.[1]

In fact, the Scala compiler only parses the definitions of the classes, the interfaces, the enumerations, the fields and the methods. That is the reason why we must compile the Java sources with the Java compiler.

The goal of this project is to extend the Java source parser of the Scala compiler to parse the Java statements and expressions. The extended parser would build the abstract syntax tree that corresponds to the Java code it has read. We expect two applications of the implementation of this extension of the Java parser:

1. Translate some Java sources into the Scala language.
2. Compile the Java sources with the Scala compiler in the case of mixed projects.

The translator works as follows. The improved Java source parser builds the abstract syntax trees that correspond to the Java sources and then prints these trees with the tree printer already present in the Scala compiler. The tree printer prints the tree in the corresponding Scala code. Therefore this works as a translator from Java to Scala.

To compile the Java sources with the Scala compiler, the parser produces the abstract syntax trees and give them to the next phases of the compiler (namer, type checker, generator) that checks that it is correct (according to the Scala

[1] Source: http://www.scala-lang.org/node/348

typing rules!) and generate the .class files. The advantage of this application is to simplify the way to compile a mixed project. The procedure presented above would be replaced by only one step: Submit all the Java/Scala sources to the Scala compiler that generates the .class files.

Chapter 2

Previous work

Previously, we explained that the way to compile a mixed Java/Scala project and that the Scala compiler has a Java source parser. This parser only parses the declarations (classes, interfaces, enumerations, fields and methods) but skips all the statements and the expressions. The declarations are necessary to compile the Scala sources of the mixed project because the type checker must ensure that the dependencies between the Java and the Scala sources are correct. Because the Scala compiler does not compile the Java sources (this is the Java compiler's job), it does not need to have the method bodies and the initialisation expression of the fields. This is the reason why the Java parser skips all these parts of the code. Therefore it can not generate any .class files for the Java sources.

The work of translating the Java sources into Scala falls in two categories. First some people have defined a set of correspondances between the Java elements and the Scala ones. This is the case of Emir Burak[1] and A. Sundararajan[2]. Unfortunately all the work must be done by hand because they only give the way to translate this or that element from Java to Scala. Then some other people have implemented a translator from Java to Scala: Scalafy[3] and Jatran[4]. These two translators build their own abstract syntax tree from the Java source files. That means they have defined and optimised the tree to translate the Java code into Scala. This is a main difference with our project because our implementation of the parser builds an abstract syntax tree that is optimised to compile the Scala code.

[1] http://lamp.epfl.ch/~emir/bqbase/2005/01/21/java2scala.html

[2] http://blogs.sun.com/sundararajan/entry/scala_for_java_programmers

[3] Author: Paul Philips
http://wiki.jvmlangsummit.com/Scalify and http://github.com/paulp/scalify/tree/master

[4] http://code.google.com/p/jatran/ and
http://eokyere.blogspot.com/2007/07/jatran-java-to-scala-and-actionscript.html

Chapter 3

Grammar and correspondences

3.1 The Java grammar

The first step of this project is to get the Java grammar and analyse it. We use the grammar provided by Sun in the Java specification third edition [1]. Because we just have to add the parsing of the expressions and the statements, we only treat these parts of the grammar (and the ones pointed from these parts). The first part of the current chapter presents the elements of the grammar that have been modified to simplify the implementation of a parser. The second part explains which are the correspondences between the transformed grammar and the Scala abstract syntax tree.

We can see the original grammar in appendix A and the grammar after the transformations in appendix B.

3.1.1 Short if

The first element that can be changed is the "...If" and "...ShortIf" separate rules. A copy of the relevant part of the grammar is given in the figure 3.1.1. They are different for only one reason: In Java it is allowed to write:

```
if (condition1)
  if (condition2)
    statement1
  else
    statement2
```

In this case, the *else* corresponds to the inner *if*. The Java specification defines that the *else* is related to the innermost *if*. The grammar is designed to take care of this problem. The solution is two have two kind of nodes. The first one can contain some short *if* (if-else without braces) in its substatements while the second that can not. In the implementation of a parser, the fact that the *else* is related to the innermost *if* is trivial to handle: when we see an *if* token, we get the condition, the *then* statement and if we see an *else* we recover it and link it to the current *if*. Hence if there is another *if* in the *then* statement that

is not surrounded by braces, it consumes the *else* part automatically. Therefore we simply can remove the "...ShortIf" rules from the grammar.

```
Statement:
    StatementWithoutTrailingSubstatement
    LabeledStatement
    IfThenStatement
    IfThenElseStatement
    WhileStatement
    ForStatement

StatementNoShortIf:
    StatementWithoutTrailingSubstatement
    LabeledStatementNoShortIf
    IfThenElseStatementNoShortIf
    WhileStatementNoShortIf
    ForStatementNoShortIf
    IfThenStatement:
    if ( Expression ) Statement
```

```
StatementWithoutTrailingSubstatement:
    Block
    EmptyStatement
    ExpressionStatement
    AssertStatement
    SwitchStatement
    DoStatement
    BreakStatement
    ContinueStatement
    ReturnStatement
    SynchronizedStatement
    ThrowStatement
    TryStatement
```

Figure 3.1: Relevant part of the grammar about the short if rules

3.1.2 Infinite loop in the Primary rule

Another problem that appears in this grammar concerns the Primary, FieldAccess, MethodInvocation and ArrayAccess rules (see A. There is an infinite loop with these rules that does not consume any token. These loops are used to chain field accesses, method invocations and array accesses (ex: a.b[3].c(1, 2).d). To make them parsable we must modify the grammar.

The PrimaryNoNewArray rule (without FieldAccess, MethodInvocation and ArrayAccess alternatives) contains the set of elements that can start an expression. Thus, we remove the FieldAccess, MethodInvocation and ArrayAccess alternatives from the PrimaryNoNewArray rule. We adapt the Primary rule to start with this updated PrimaryNoNewArray rule and continue with a new rule (PrimaryRest) that consumes (and create the corresponding AST) the FieldAccess, the MethodInvocation and the ArrayAccess rules.

```
Primary:
    PrimaryNoNewArray
    ...

PrimaryNoNewArray:
    ... (other alternatives
         not relevant here)
    FieldAccess
    MethodInvocation
    ArrayAccess
```

```
FieldAccess:
    Primary . Identifier
    ...

MethodInvocation:
    ...
    Primary . NonWildTypeArgumentsopt
    ...

ArrayAccess:
    ...
    PrimaryNoNewArray [ Expression ]
```

3.2 Correspondences with the Scala AST

In the next step the correspondences between the Java grammar rules (statement and expressions in our case) and the Scala abstract syntax tree must be

formalised. Most of the grammar rules can can be translated directly but some other are more tricky to translate. I presents these special cases in the next subsections.

3.2.1 While and do-while statements

In the Scala abstract syntax tree, there is no single node that represents the *while* statement and the *do-while* statement. They are represented in a more complex manner. To build the *while* (resp. *do-while*) node, we use the labels. The advantage of the labels is that we can jump to them. When we jump to a label, the body of the label is executed. Thus a while is transformed into a label. Its body is an *if* node that checks the condition of the while. If the condition is true then we can execute the body of the original *while* (resp. *do-while*) statement and, at the end, jump to the label to loop.

```
while ( Expression ) Statement
```

is transformed by the parser in

```
while$1():   //Label.
{
  if (Expression) {
    Statement
    while$1()   //Jump to the label while$1.
  }
}
```

3.2.2 For loops

There exist two kind of *for* statement in Java:

1. The basic *for* statement:
   'for' '(' ForInit$_{opt}$ ';' Expression$_{opt}$ ';' ForUpdate$_{opt}$ ')' Statement
2. The enhanced *for* statement:
   'for' '(' VariableModifiers$_{opt}$ Type Identifier ':' Expression ')' Statement

Scala does not have *for* loops. Thus the Java *for* loops are translated into *while* loops. The first *for* statement (1) is transformed into:

```
ForInit
while (Expression) {
  Statement
  ForUpdate
}
```

The translation of the second *for* statement (see 2) needs a some adjustments. This syntax is a syntactic sugar to go through an iterable structure. Thus we remove the sugar when building the abstract syntax tree. The result of the translation is:

```
Java.util.Iterator<Type> iter$ = Expression.iterator();
while (iter$.hasNext()) {
```

```
  Type Identifier = iter$.next();
  Statement
}
```

where "Type", "Identifier", "Expression" and "Statement" are those present in the grammar presented in the enumeration above (see 2).

3.2.3 Break and continue statements

The break statement and the continue statement do not exist in Scala. The limitations inherent to these two statements are presented in Section 5.2.3. But even if they do not exist in the Scala language, we can build an abstract syntax tree for them because they are only jumps. The break statement jumps to the end of the enclosing loop or switch while the continue statement jumps to the next iteration of the directly enclosing loop. For the both statements we have to be careful when we create the labels because we can only jump to labels that enclose the jump. That means that a break that jumps to the end of the enclosing loop or switch can not jump to an empty label placed immediately after the loop.

The continue statement As explained in Section 3.2.1 a label is created for the *while*, *do-while* and *for* (because it is translated into a while) statements. Thus the continue statement can simply jump to this label in order to go to the next iteration. This is correct for the while and the do-while statements because they do not have a specific piece of code that update the variables. In the *for* statement, before going to the next iteration we must execute the update instructions. Therefore we can not simply jump to the first enclosing defined label because the update would not be executed. Then we must place these instructions before checking the condition of the loop. But it must not be executed at the first execution of the loop. To solve the problem a boolean variable is added and set to false as initial value. Then the update statements are executed only if the previously defined variable is true. After the first iteration, the variable is set to true and will not change until the end of the loop. An example of such a transformation is provided in the following listing.

```
for (init; cond; update)
  body
```

is transformed into

```
bool for$1$var = false;
init
for$1() :  //Label.
{
  if (for$1$var) {
    update
  }
  for$1$var = true;
  if (cond) {
    body
    for$1()  //Jump to the label for$1.
  }
```

}

But this additional code is only necessary if there is one (or more) *continue* statement inside the body of the loop. Thus one easy optimisation, that consist not generating the non necessary code if there is no continue in the loop's body, can be implemented.

The break statement The *break* statement is a statement that jumps to the end of the enclosing loop (*while*, *do-while* or *for*) or *switch*. That means that the loop does not take any further iteration. The way to do that is to jump after the loop. Unfortunately, it is not so simple because we can not jump to a label that is defined later in the code. For this reason we must create a abstract syntax tree that is a little bit more complicated. We have to defined a boolean variable that conditions the execution of the loop and is set to true. Then we create a label and in the body of the label we check whether the variable is true or not. If it is true we execute the loop else not. The *break* statement sets this variable to false and then jump to the previously defined label. An example of such a transformation is provided in the listing below.

```
while (cond) {
  ...
  break;
  ...
}
```

is transformed into

```
bool while$1$var = true;
while$1() :  //Label.
{
  if (while$1$var) {
    if (cond) {
      ...
      //These lines are the translation
      //of the break instruction.
      while$1$var = false;
      while$1();  //Jump to the label while$1.
      ...
    }
  }
}
```

As for the *continue* statement, all of the added code around the loop is necessary only if there is a break in the loop's body. If there is no break, we can easily not generate all these elements.

Chapter 4

Implementation

After these transformations the grammar is parsable and we have defined the correspondences between the rules of the grammar and the Scala compiler's abstract syntax tree. It is time to implement it. Most of the parser was easy to implement but some small elements have creating major problems. They are presented in this chapter and the solution we have found for them.

4.1 Comparison and cast

One major issue that we encounter when we want to write a Java parser is: How to differentiate between a cast and a comparison. Lets just present an example:

(A<B>)a

If the parser produces the left-most derivation of the input, it thinks that it is a parenthesised expression. This expression starts with a comparison (less than) between A and B that produces a node of the abstract syntax tree BinOp(A, "<", B). Then the parser sees the greater than operator and produces an error because there is no right hand side in the comparison. Therefore the parser must produce the right-most derivation (LR) of the input, i.e. the derivation that consumes the longest sequence of tokens. Writing an LR parser is not so easy because we have to see a non-determined sequence of tokens before being able to make a decision.

There are two possibilities to write a parser that handles this case:

1. Not really consume the tokens and then we have the possibility to return back in the input to try another rule.

2. Build the two possible trees (cast and comparison) until the moment we can make the decision.

The first solution consists in having a list of tokens. When we are in this special case where we have to make a difference between a comparison and a cast, we add the read tokens in the list instead of forgetting them. If the current checked rule (cast for example) is correct, we clear the list, else we try the other case with as next token, the first token of the built list. The second solution is much easier to implement because we do not have to build a list and adapt

the nextToken method. It requires only to build two trees and throw away the wrong solution. This is the solution we have implemented in our Java parser.

4.2 Field access, method invocation and types

The FieldAccess, the MethodInvocation and the Type rules in the grammar are very similar. We must read a sequence of tokens (instead of one) to make the difference between these three rules. There are many locations in the grammar where we have to make this difference. Instead of copy-pasting the code, we put the treatment in a dedicated method. It returns the built tree and some booleans (three) that indicates if the element is:

1. A field
2. A method
3. A type
4. Do not know

There is some case where the parser does not know which element we have.

A . b . c

Is it a type, or a field access? The programmer knows what he has wrote but the parser can not make a difference by only looking at this five tokens. The difference is made further in the calling rule, *i.e.* the rule that calls this dedicated method for the treatment of these three rules (FieldAccess, MethodInvocation and Type). The calling rule knows when reading the next tokens if it was a type of a field access. Thus the method that parse the code "A.b.c" returns that it does not know what it returns and let the calling rule choose in what case it is.

4.3 Break and continue statement

In Section 3.2.3 we explained how we translate the *break* and the *continue* statements in an abstract syntax tree node. We also explained in Sections 3.2.1 and 3.2.2 that the *for*, *while* and *do-while* statements are transformed in a label node. Thus we simply have to jump to the first enclosing loop. To do that we must build a stack of the loops in which we are. Each time we see a loop we push on the stack the name of the corresponding label. Thus, when we parse a *break* or a *continue* statement, we can get the element on top of the stack and jump to it. The last element that must be taken into account is that a *break* can be present in a *switch* statement but not a *continue*. Therefore, we add a label when we see a *switch* as for the other loops. During parsing the rest of the code, if the current statement is a *continue* statement, we check whether the last added label corresponds to a switch or not. If so, we get the first previously added label.

We can do a optimisation that makes the abstract syntax tree more simpler. As explained in Section 3.2.3, some elements must be added around the loops to be able to treat correctly the *break* and the *continue* statements. But all of this is not necessary if we do not see any of these two statements. Thus we

add two other stacks of booleans, one for the *break* and one for the *continue*. When we enter a new loop we push "false" on the top of the two stacks. These elements are set to true if and only if there is one (or more) *break* and one (or more) *continue* in the direct body of the loop and not in a sub-loop. At the end of the parsing of the loop, the tree is created. At that time the top element of the two stacks are popped and if they are both "false", the additional elements needed to handle the *break* and the *continue* statement are not added to the abstract syntax tree.

4.4 Binary expressions

The treatment of the binary expressions is the same than the one in the Scala parser of this compiler. It keeps a stack of the previous binary expressions. When it sees a new one, it pushes it on the stack and try to reduce the stack. The reduction takes care of the precedence of the operators. That means that the reduction collapse only the previous binary operations that have a precedence bigger than the current one. A list of the precedences is given in the following table.

Operators	Precedence
assignment operator	0
text	1
"||"	2
"&&"	3
"|"	4
"^"	5
"&"	6
"==", "!"	7
"<", ">", "<=", ">="	8
"<<", ">>", ">>>"	9
"+", "−"	10
"∗, "/", "%"	11
others	12

Table 4.1: Table of the precedences of the different operators

Chapter 5

Limitations

Unfortunately we are not able to translate in Scala or compile with the Scala compiler every Java program. This comes from two elements. First we are building a Scala abstract syntax tree from the Java code. This tree is designed to compile the code and not to translate it. That means that some elements do not have their exact corresponding node in the tree. These elements are transformed into something more complex. Then there is some differences between the Java and the Scala languages which lead to Java programs with elements that can not (in a reasonable manner) be translated in Scala.

5.1 Limitations for both translation and compilation

There is some problems that limit both the translation from Java to Scala and the compilation of the Java sources by the Scala compiler.

5.1.1 Override

In Scala, when a method in class A (that extends B) overrides a method defined in class B, the keyword "override" must be present in front of the method definition. This keyword indicates to the compiler that this method overrides a method from a superclass. In Java, this keyword does not exist. We can override methods without indicating anything to the compiler.

If we want to compile any Java code that contains some overriding methods, we need to set the "OVERRIDE" flag of the method node in the tree. In the Scala compiler, this is the namer phase which binds the classes, the interfaces, the methods etc. to each other. Therefore, during the parser phase, we are not able to define if a method is overriding another or not. Thus we can not add the flag at this point. The problem is also present if we want to translate the Java code into the Scala language. If we want to produce a fully correct Scala code we need to write the keyword "override" but at this phase (parser) we do not know if it must be present or not.

One way to solve this problem (for compiling the Java files) would be to modify the type checker phase. First, during the parser phase, we build an abstract syntax tree of the methods that contains a *Modifier* field. This field is

composed with the explicit modifiers of the method in the file (public, private, static, etc.) but if it comes from a Java file, it also has the flag "JAVA". Then the element "CompilationUnit" that contains the tree for one file of code (Java or Scala) knows if it comes from a Java or a Scala file. That means that when we type check this CompilationUnit and therefore the overriding method, the compiler can know whether it comes from a Java file. If so, and if the method is overriding another one in a superclass, it could add the flag "OVERRIDE" or simply not checking if it is present or not.

5.1.2 Constructors

The Scala language has some constraints about the constructors.

- There can be only one primary constructor.
- Only the primary constructor can call a constructor from the superclass.
- All additional constructors must call as their first statement a previously defined constructor.

This is very different with the Java constructors. In Java, as the first statement we can: call another constructor of the same class (that can be declared later), call a superclass's constructor or call nothing (call to the default constructor of the superclass is inserted). The best way to illustrate the problem is a concrete example.

```
class A {
  public A() { ... }
  public A(int b) { ... }
}
class B extends A {
  public B() { super(); ... }
  public B(int b) { super(b); ... }
}
```

Figure 5.1: Source: http://www.nabble.com/-scala–the-winding-road-to-java-constructor-semantics-td18658789.html

In this example we can easily see the problem if we want to translate this code into the Scala language: Which of the constructors of A (resp. B) is the primary constructor? If we want to compile this code, the problem is the same: How build the tree such that only one constructor calls a constructor of the superclass and all other constructor (that must be defined later) call this primary constructor?

This is a major problem in the context of this project that has not only one possible solution. One solution could be to add an artificial constructor that takes one argument for each field of the class. This constructor calls the primary constructor of the superclass (that could either be built by the same way or already written by the programmer) and initialise all the fields of this class. Then insert a call to the previously created constructor as the first statement of each additional constructor. The problem (as for the override problem) is that

we need to analyse the tree to do such constructions. We need to know the list of all fields of the class, which class is the superclass (link to its tree node) and which constructor of the superclass is the primary. All these elements are not available in the parser phase. Once again we would need an additional phase immediately after the Java parser phase that would handle this problem.

The verification of the rules defined above is the Scala parser's job. Thus, the further phases do not check it any more. Therefore if the Java parser build an abstract syntax tree from a class that contains more than one constructor and some of them call a superclass's constructor, then it can give it to the further phases. Unfortunately, these further phases do not expect to receive such a tree and we can not ensure that the generated code is correct or not.

5.2 Translate Java programs into Scala language

As explained in the introduction of this chapter, we are constrained by the way we choose to translate the Java code into the Scala language. We use a tree to represent the program that is built and optimised to compile (and produce class files of) Scala source files. That means that much syntactic sugar is removed at the parser phase of the compiler. Thus if we want to write the program represented by the tree using a pretty printer, we have the code with some elements that are transformed in a much more complex block and without the syntactic sugars.

5.2.1 While, do-while statements

A good example for the current problem is the *while* statement. As explained in Section 3.2.1, the *while* statement is transformed in a complex node of the abstract syntax tree. The limitation of a *do-while* is exactly the same. We recall the transformation for a *while*:

```
while (cond) body
```

is transformed in

```
while$1(): //Label.
{
  if (cond) {
    body
    while$1() //Jump to the label while$1.
  }
}
```

The second code is not a valid Scala code because the labels (¨while$1¨ in the example) are not allowed in Scala. We use the same argument for the *do-while* statement. One solution would be to adapt the pretty printer present in the compiler to handle these cases. But they are complex cases and they could become even more complex if there is some *break* and *continue* statements (see Section 3.2.3) in their bodies. Another solution is to create some intermediate tree nodes for the abstract syntax tree. These nodes (called While and DoWhile) would be valid only in the Java parser phase, the pretty printer and an additional phase that would remove them. Because the pretty-printer would handle them,

it could print correctly any *while* and *do-while* statements in Scala. However, with this solution, we create temporarily an invalid abstract syntax tree. It is invalid because it contains some nodes that are not accepted by the other phases of the compiler.

5.2.2 For statement

Another example is the *for* statement. It is transformed (explained in Section 3.2.2) in a *while* that is itself transformed in a complex node (explained in Section 3.2.1). That means that the abstract syntax tree (built during the parser phase) is completely different from the original code (but behaves exactly the same). If the tree is printed using the tree-printer present in the compiler, all the complex nodes are printed even if some elements do not exists in Scala (see Section 5.2.1. We could use the solution presented in Section 5.2.1 for the *while* and the *do-while* statements: Adding a temporary while node that could be printed correctly by the tree-printer, the *for* loop would be transformed into a *while* loop.

5.2.3 Break and continue statements

The *break* and the *continue* statements do not exist in Scala. The solution to compile these two statements is to transform them into a jump to a label placed around the loop we want to *continue* (resp. *break*). Unfortunately the labels are not allowed in the Scala files. Therefore the resulting code of the translation is not correct in Scala.

5.3 Compiling Java source files

5.3.1 Type checker

The type checker present in the Scala compiler makes some assumptions about the abstract syntax tree it receives. These assumptions are correct for a Scala AST because all the previous phases execute all their steps when compiling a Scala source file. This does not hold in the case of the compilation of a Java source file. Indeed the namer phase skips some steps for the Java code and not for a Scala code. It implies that some elements are not added in the AST but the type checker needs them. If they are not present, it generates an error and prevents the rest of the compilation phases from executing. One example of this problem is a field declaration in a class. In Java, the getter and the setter for this field are not necessary. Therefore the namer phase does not add them automatically for Java code but does it for Scala code. Thus the type checker assumes that all the fields have a getter and a setter but it assumes this in all cases and not only in the case of a Scala code. When it receives a Java code, it checks whether the AST contains the getters and the setters for all fields and it finds that they have not. If we want to be able to compile every Java source file into Scala, the namer and the type checker need to be modified to be able to compile the Java code.

Chapter 6

Future work

One way to simplify the Java parser is to add some new temporary nodes in the Scala abstract syntax tree. These nodes would only be present in the Java parser and printed by the pretty printer. In the Java parser, they would represent some elements that have a special meaning in Java but not in Scala as array access for example. If we also add nodes for while and do-while statements, the pretty printer would be able to produce more correct Scala code. These nodes could be removed by an additional phase in the compiler that would be added immediately after the Java parser phase.

As presented in Sections 5.1.1 and 5.1.2 some problems need an additional phase to be solved. Adding these two phases for the problem of override and the one for the constructors would give the possibility to the Scala compiler to translate a larger set of Java source files. Without these phases we are very limited about the elements we can translate because the constructors and the overriding of methods are two notions very important in object oriented programming.

The last element we suggest to do is to adapt the namer and the type checker phase in order to being able to compile correctly a Java code. They are defined to name and type check the Scala code but if we want to add the possibility to mix some Java and Scala source files in the same project and compile all of them with the same compiler, it must be able to do it correctly for both languages.

The only elements of the Java grammar that are not handled by this parser are the annotations. They are not necessary to parse and compile the Java code but they could be useful. For example: if the programmer add the @override annotation for a method, this would permit to set the OVERRIDE flag in the modifiers of the method node. Thus the code could be compiled without any modifications of the namer or the type checker. But it would be useful only if the programmer writes correctly and every time it is necessary.

Bibliography

[1] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

Appendix A

Original Java grammar

A.1 Expressions

```
ConstantExpression:
    Expression

Expression:
    AssignmentExpression

AssignmentExpression:
    ConditionalExpression
    Assignment

Assignment:
    LeftHandSide AssignmentOperator
        AssignmentExpression

LeftHandSide:
    ExpressionName
    FieldAccess
    ArrayAccess

AssignmentOperator: one of
    = *= /= %= += -= <<= >>= >>>= &= ^= |=

Primary:
    PrimaryNoNewArray
    ArrayCreationExpression

PrimaryNoNewArray:
    Literal
    Type . class
    void . class
    this
    ClassName.this
    ( Expression )
    ClassInstanceCreationExpression
    FieldAccess
    MethodInvocation
    ArrayAccess

Literal:
    IntegerLiteral
    FloatingPointLiteral
    BooleanLiteral
    CharacterLiteral
    StringLiteral
    NullLiteral

ClassInstanceCreationExpression:
    new TypeArgumentsopt ClassOrInterfaceType
        ( ArgumentListopt ) ClassBodyopt
    Primary. new TypeArgumentsopt Identifier
        TypeArgumentsopt ( ArgumentListopt )
         ClassBodyopt

ArgumentList:
    Expression
    ArgumentList , Expression

ArrayCreationExpression:
    new PrimitiveType DimExprs Dimsopt
    new ClassOrInterfaceType DimExprs Dimsopt
    new PrimitiveType Dims ArrayInitializer
    new ClassOrInterfaceType Dims ArrayInitializer

DimExprs:
    DimExpr
    DimExprs DimExpr

DimExpr:
    [ Expression ]

Dims:
    [ ]
    Dims [ ]

FieldAccess:
    Primary . Identifier
    super . Identifier
    ClassName .super . Identifier

MethodInvocation:
    MethodName ( ArgumentListopt )
    Primary . NonWildTypeArgumentsopt
        Identifier ( ArgumentListopt )
    super . NonWildTypeArgumentsopt
        Identifier ( ArgumentListopt )
    ClassName . super . NonWildTypeArgumentsopt
        Identifier ( ArgumentListopt )
    TypeName . NonWildTypeArguments
        Identifier ( ArgumentListopt )

ArgumentList:
    Expression
    ArgumentList , Expression

ArrayAccess:
    ExpressionName [ Expression ]
    PrimaryNoNewArray [ Expression ]
```

```
PostfixExpression:
    Primary
    ExpressionName
    PostIncrementExpression
    PostDecrementExpression

PostIncrementExpression:
    PostfixExpression ++

PostDecrementExpression:
    PostfixExpression --

UnaryExpression:
    PreIncrementExpression
    PreDecrementExpression
    + UnaryExpression
    - UnaryExpression
UnaryExpressionNotPlusMinus

PreIncrementExpression:
    ++ UnaryExpression

PreDecrementExpression:
    -- UnaryExpression

UnaryExpressionNotPlusMinus:
    PostfixExpression
    ~ UnaryExpression
    ! UnaryExpression
    CastExpression

CastExpression:
    ( PrimitiveType Dimsopt )
        UnaryExpression
    ( ReferenceType )
        UnaryExpressionNotPlusMinus

MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression +
        MultiplicativeExpression
    AdditiveExpression -
        MultiplicativeExpression

ShiftExpression:
    AdditiveExpression
    ShiftExpression << AdditiveExpression
    ShiftExpression >> AdditiveExpression
    ShiftExpression >>> AdditiveExpression


RelationalExpression:
    ShiftExpression
    RelationalExpression < ShiftExpression
    RelationalExpression > ShiftExpression
    RelationalExpression <= ShiftExpression
    RelationalExpression >= ShiftExpression
    RelationalExpression instanceof ReferenceType


EqualityExpression:
    RelationalExpression
    EqualityExpression == RelationalExpression
    EqualityExpression != RelationalExpression


AndExpression:
    EqualityExpression
    AndExpression & EqualityExpression


ExclusiveOrExpression:
    AndExpression
    ExclusiveOrExpression ^ AndExpression


InclusiveOrExpression:
    ExclusiveOrExpression
    InclusiveOrExpression | ExclusiveOrExpression


ConditionalAndExpression:
    InclusiveOrExpression
    ConditionalAndExpression && InclusiveOrExpression


ConditionalOrExpression:
    ConditionalAndExpression
    ConditionalOrExpression || ConditionalAndExpression


ConditionalExpression:
    ConditionalOrExpression
    ConditionalOrExpression ? Expression :
        ConditionalExpression
```

A.2 Statements

```
Block:
    { BlockStatementsopt }

BlockStatements:
    BlockStatement
    BlockStatements BlockStatement

BlockStatement:
    LocalVariableDeclarationStatement
    ClassDeclaration
    Statement

LocalVariableDeclarationStatement:
    LocalVariableDeclaration ;

LocalVariableDeclaration:
    VariableModifiers Type VariableDeclarators

VariableDeclarators:
    VariableDeclarator
    VariableDeclarators , VariableDeclarator

VariableDeclarator:
    VariableDeclaratorId
    VariableDeclaratorId = VariableInitializer


VariableDeclaratorId:
    Identifier
    VariableDeclaratorId [ ]


VariableInitializer:
    Expression
    ArrayInitializer


Statement:
    StatementWithoutTrailingSubstatement
    LabeledStatement
    IfThenStatement
    IfThenElseStatement
    WhileStatement
    ForStatement
```

```
StatementWithoutTrailingSubstatement:
    Block
    EmptyStatement
    ExpressionStatement
    AssertStatement
    SwitchStatement
    DoStatement
    BreakStatement
    ContinueStatement
    ReturnStatement
    SynchronizedStatement
    ThrowStatement
    TryStatement

StatementNoShortIf:
    StatementWithoutTrailingSubstatement
    LabeledStatementNoShortIf
    IfThenElseStatementNoShortIf
    WhileStatementNoShortIf
    ForStatementNoShortIf
    IfThenStatement:
    if ( Expression ) Statement

IfThenElseStatement:
    if ( Expression ) StatementNoShortIf
        else Statement

IfThenElseStatementNoShortIf:
    if ( Expression ) StatementNoShortIf
        else StatementNoShortIf

EmptyStatement:
    ;

LabeledStatement:
    Identifier : Statement

LabeledStatementNoShortIf:
    Identifier : StatementNoShortIf

ExpressionStatement:
    StatementExpression ;

StatementExpression:
    Assignment
    PreIncrementExpression
    PreDecrementExpression
    PostIncrementExpression
    PostDecrementExpression
    MethodInvocation
    ClassInstanceCreationExpression

AssertStatement:
    assert Expression1 ;
    assert Expression1 : Expression2 ;

SwitchStatement:
    switch ( Expression ) SwitchBlock

SwitchBlock:
    { SwitchBlockStatementGroupsopt
        SwitchLabelsopt }

SwitchBlockStatementGroups:
    SwitchBlockStatementGroup
    SwitchBlockStatementGroups
        SwitchBlockStatementGroup

SwitchBlockStatementGroup:
    SwitchLabels BlockStatements

SwitchLabels:
    SwitchLabel
    SwitchLabels SwitchLabel

SwitchLabel:
    case ConstantExpression :
    case EnumConstantName :
    default :

EnumConstantName:
    Identifier

WhileStatement:
    while ( Expression ) Statement

WhileStatementNoShortIf:
    while ( Expression ) StatementNoShortIf

DoStatement:
    do Statement while ( Expression ) ;

ForStatement:
    BasicForStatement
    EnhancedForStatement

BasicForStatement:
    for ( ForInitopt ;
          Expressionopt ;
          ForUpdateopt ) Statement

ForStatementNoShortIf:
    for ( ForInitopt ;
          Expressionopt ;
          ForUpdateopt ) StatementNoShortIf

ForInit:
    StatementExpressionList
    LocalVariableDeclaration

ForUpdate:
    StatementExpressionList

StatementExpressionList:
    StatementExpression
    StatementExpressionList , StatementExpression

EnhancedForStatement:
    for ( VariableModifiersopt Type Identifier:
        Expression) Statement

BreakStatement:
    break Identifieropt ;

ContinueStatement:
    continue Identifieropt ;

ReturnStatement:
    return Expressionopt ;

ThrowStatement:
    throw Expression ;

SynchronizedStatement:
    synchronized ( Expression ) Block

TryStatement:
    try Block Catches
    try Block Catches1opt Finally
%

Catches:
    CatchClause
    Catches CatchClause

CatchClause:
    catch ( FormalParameter ) Block

Finally:
    finally Block

FormalParameter:
    VariableModifiers Type VariableDeclaratorId

VariableDeclaratorId:
    Identifier
    VariableDeclaratorId [ ]
```

Appendix B

Transformed Java grammar

To simplify the grammar some new symbols are used:

- {} is the repetition. At least zero time.
- [] is the option. Zero or one time.
- () is the alternative. Exactly one of the alternatives.

The transformations are present to remove the infinite loops present in the grammar (see Section 3.1.2) and to simplify the work to write the parser. But we did not transform the fact that some elements have to consume a sequence of tokens before taking a decision. We explain it in Sections 4.1 and 4.2.

B.1 Expressions

```
Primary:
    PrimaryNoNewArray [ PrimaryRest ]
    ArrayCreationExpression
        { '.' Identifier
            ['(' [ArgumentList] ')'] }
    ClassInstanceCreationExpression
        [ PrimaryRest ]

PrimaryRest:
    { ( '.' Identifier ['(' [ArgumentList] ')']
        | '[' Expression ']' ) }

PrimaryNoNewArray:
    Literal
    'void' '.' 'class'
    'this'
    '(' Expression ')'
    'super' . Identifier
        ['(' [ArgumentList] ')']
    Identifier {'.' Identifier} '.' 'this'
    Identifier {'.' Identifier}
        ['(' [ArgumentList] ')']
    Identifier {'.' Identifier} '.' 'super'
        '.' Ident ['(' [ArgumentList] ')']
    Identifier {'.' Identifier} {'[' ']'}
        '.' 'class'

ArgumentList:
    Expression { ',' Expression }
```

```
PostfixExpression:
    Primary
    PostfixExpression '++'
    PostfixExpression '--'

UnaryExpression:
    '++' UnaryExpression
    '--' UnaryExpression
    '+' UnaryExpression
    '-' UnaryExpression
    UnaryExpressionNotPlusMinus

UnaryExpressionNotPlusMinus:
    PostfixExpression
    '~' UnaryExpression
    '!' UnaryExpression
    CastExpression

CastExpression:
    '(' PrimitiveType '['Dims']' ')'
        UnaryExpression
    '(' ReferenceType ')'
        UnaryExpressionNotPlusMinus

MultiplicativeExpression:
    UnaryExpression
        { ('*' | '/' | '%') UnaryExpression }

AdditiveExpression:
    MultiplicativeExpression
        { ('+' | '-') MultiplicativeExpression }
```

```
ShiftExpression:
   AdditiveExpression
        { ('<<' | '>>' | '>>>')
            AdditiveExpression }

RelationalExpression:
    ShiftExpression
        { ('<' | '>' | '<=' | '>=')
            ShiftExpression }
    ShiftExpression 'instanceOf' ReferenceType

EqualityExpression:
    RelationalExpression
        { ('==' | '!=')
            RelationalExpression }

AndExpression:
    EqualityExpression
        { '&' EqualityExpression }

ExclusiveOrExpression:
    AndExpression { '^' AndExpression }

InclusiveOrExpression:
    ExclusiveOrExpression
        { '|' ExclusiveOrExpression }

ConditionalAndExpression:
    InclusiveOrExpression
        { '&&' InclusiveOrExpression }

ConditionalOrExpression:
    ConditionalAndExpression
        { '||' ConditionalAndExpression }

ConditionalExpression
    ConditionalOrExpression
        [ '?' Expression ':' ConditionalOrExpression ]
```

B.2 Statements

No transformations have been done in this part of the grammar.