MiniJava Compiler

DD2488 - Compiler Construction

Gustaf Lindstedt
glindste@kth.se
910301-2135

Jonathan Murray
jmu@kth.se
900808-1532

May 19, 2014

1 Introduction

Compilers have been around almost since the birth of programming. Although the likelihood of having to write a compiler from scratch today is quite small, it can still be useful to understand how one works. The process of translating source code to machine code instructions or java bytecode is quite complex and the different steps involve many varying techniques that can also be applied in other areas. For instance the task of parsing text according to a grammar is very central in the field of Natural language processing, whereas the task of selecting the final assembly instructions is also needed when developing integrated systems. In the case of making a compiler that compiles to java bytecode, one can learn a lot about how the JVM works, something that can be needed for writing efficient java code.

2 Choice of Tools

For defining the grammar and creating the abstract syntax tree we chose to use the tool JavaCC. This was mostly because it seemed to be a common choice with a satisfying amount of online documentation. We have been mostly happy with this choice. We gained the needed knowledge through a combination of following online tutorials and studying example grammars included in the software. The most common problems that we faced when writing the grammar were handling LOOKAHEAD correctly, avoiding recursive productions and, later on in the process, handling operator precedence and the order of nodes in the resulting syntax tree. To generate the abstract syntax tree we used JJTree which is included in JavaCC.

3 Code Overview

3.1 Important classes

The code for the parser and for building the abstract syntax tree (AST) are generated using JJTree and JavaCC with the defined minijava-grammar located in MiniJava.jjt. To use the generated parser and AST generator we use the ParseTree wrapper class, which takes an input stream and returns an AST. The remaining work after creating the AST up until generating the java bytecode, is mainly handled by the following three tree-visitor classes.

- SymbolTableVisitor.java
- TypeCheckVisitor.java
- JVMVisitor.java

All of them traverse the AST top-to-bottom doing some work for each node visited. An important utility class is VisitorUtil.java, which conain a set of static methods which are used internally by the visitors.

3.2 Helpers

Context

In order to solve the problem of keeping track of the context of a current node in the AST, we use an object called Context. This object is sent along as a node visits its children and augmented along the way as the context changes. It uses three fields to keep track of the context, `className`, `methodName` and `varName`. This Context object is further extended to a JVMContext in the JVMVisitor class, in order to keep track of stack size and local variables.

ClassData and MethodData

In the symbol table we use two objects called ClassData and MethodData. These objects contain important information about the class or method such as return type (for a method) and super class (for a class). They also store internal symbol tables for fields and methods for classes and parameters and local variables for methods.

3.3 SymbolTableVisitor

The task of this class is to visit the nodes that correspond to declarations in the source code and construct a symbol table that can be referenced in later stages of the compilation. These declarations include classes, instance fields, methods, method parameters and local variables. The resulting symbol table is implemented roughly as a nested hashtable, where the outermost keys are the class names, and method names of specific classes map to another layer. The symbol table contains information

about all declarations such as types, names and, in the case of method parameters, order. While creating this symbol table the class also makes sure that there are no cyclic inheritance relations and no duplicate declarations. Other issues such as making sure that no declarations refer to missing types are handled in later stages when the symboltable is complete.

Important exceptions explained:

- CyclicInheritance
  - example: $A \rightarrow B \rightarrow C \rightarrow A$, where $\rightarrow$ denotes inheritance.
- DuplicateDeclaration
  - example: `int getSum(int a, int b){ int a ... }`

3.4 TypeCheckVisitor

The task of this class is to look for all kind of errors in the code relating to type-checking. In general, this is done by examining all expressions (and sub-expressions) in the code and making sure that their actual type after evaluation equals the type that is expected by the context they appear in. By default each visit method for expressions returns the type of the evaluated expression, so the parent can check the evaluated type if needed, but often the type-checking is done by the child by checking its type against the expected type. An extremely simple example for demonstrating how the type-checker works is the assignment `a = true` where a is an int-variable. The code is processed inwards (downwards in the AST) so in this case there is one node that represents the action of assignment while its child nodes represent the variable and the right-hand side expression. From the symbol table it can be seen that `a` is of type `int`, and therefore the the right-hand side expression is expected to be the same. When visiting this expression we use an object called ExprInput, which stores a Context object and the expected type which the expression should evaluate to. When `true` is evaluated and seen to be of type `boolean`, which does not match the expected type of `int`, an informative exception is thrown and the compilation stops.

Important exceptions explained:

- *ReferencedMissingType*
  - example: declaring `Calculator calc;` when the class `Calculator` has not been defined.
- *ReferencedMissingVariable*
  - example: `a = 5;` where `a` has never been declared.
- *ReferencedMissingMethod*

  - example: `a = this.getHamburger();` where `getHamburger()` is never defined in this class.

- *WrongType*
  - example: `a = true && this.getHamburger();` where `getHamburger()` doesn't return a boolean.

- *WrongNumberArgs*
  - example: `h = this.makeSandwich(cheese);` where `makeSandwich()` expects 2 parameters.

3.5 JVMVisitor

This class produces the final bytecode for the JVM, or rather jasmin-code that is then translated to actual class files. If this stage has been reached the source code is known to be error-free so all focus lies on generating jasmin rather than looking for errors. No optimization is implemented at the time of writing, mostly due to time constraints. For each AST-node a specific set of jasmin instructions is used. For instance a negative expression such as `!true` generates the instruction for putting the value `true` on the stack by visiting the child-node and then a set of instructions to negate the current value on the stack. This approach results in quite simple and manageable code.

Branching

In order to have unique labels for branching throughout the code we used a global counter for number of branches encountered and appended the current count to the end of the labels for the current branching.

Stack Size

To calculate the appropriate stack size for a method we used the JVMContext object. It provides two methods for updating the current stack size, `addStack(int n)` and `subStack(int n)`, which are called successively as stack pushing and stack popping JVM instructions are appended to the code. Internally it keeps track of the largest encountered stack size and updates it accordingly.

4 Testing

Testing is a fundamental step in the development process, and a compiler is fairly straightforward to test, since the concept of input and output is very well-defined. The class Test.java was created to handle this task. We started writing test-cases early in the development process and made it a habit to run them when making

changes to the code and especially before pushing to our version control system. We had two types of local tests, those that should compile without error and those that shouldn't. Something we noticed quickly was that there is a high risk of introducing errors in the test programs. In the case of compile-tests this is not a big deal since you know about the, hopefully small, number of tests that fail. When there is a bug in a non-compile test however, the situation is much worse. Take the following test that is supposed to check that a return statement is not allowed in the main method:

```
class FalseMainReturnTest{
    public static void main(String[] args){
        int a
        a = 3;
        return a;
    }
}
```

The problem is that the compiler doesn't reject the test because of the return statement, but rather because of the missing semi-colon on the variable-declaration line. Therefore the compiler may very well allow return statements in the main-method although it passed the test. In other words, it is not enough that a negative test is rejected. It has to be rejected for the right reasons.

We solved this problem by creating a number of specific Exceptions mentioned in Section 3 - Code Overview. All the non-compile tests were placed inside directories whose names denoted which exception the tests were expected to throw. When catching an exception from a non-compile test, a comparison between the exception type and the expected type is made.

To make testing simple and quick, we added instructions to our ant-file for both building the project and running all the tests. When tests have been run, a summary of the results is shown in the command line along with specifics about failed tests such as stack traces. The output of Test.java is highly configurable through boolean flags to fit the needs of the current situation. To further improve the workflow the output is color-coded with red text when some tests fail and green text when everything is working.

5 Conclusion

Overall we are quite happy with the outcome of the project although we had to give up the idea of making a CPU-backend and instead go for the JVM-version. This was in part because we felt that it would take more time than we had left, but also because we didn't feel that the course book gave us enough help regarding some of the steps involved which further increased the time it would take to implement.

6 Appendix A - Bugs and mistakes

6.1 Parsing Booleans

A rather trivial bug which we encountered which nonetheless puzzled us in the beginning was that we had defined the identifier token before the literal tokens in the grammar definition. This led to boolean literals being matched as identifiers, so we got identifiers with values of "true" and "false". We were unaware that JavaCC tries to match tokens from top to bottom, giving precedence to tokens defined higher up in the grammar file. We probably assumed it did some sort of "best fit" where it tried to match the most explicit definition it found. This was trivially solved by defining identifiers after literals in the grammar definition.

6.2 Multidimensional Arrays

Handling, or rather not handling, multidimensional arrays was a bit tricky, due to how the java syntax uses square brackets. The example below demonstrates some of the issues.

Assume that a and i are variables of types `int[]` and int.

```
a = new int[0];
i = a[2];
```

The above code is syntactically correct although it would crash due to an out-of-bounds error if executed. The code below however is not correct. Intuitively it looks like a shorthand of the example above, but the problem is that in the full version of java this would rather be interpreted as the creation of a two-dimensional array.

```
i = new int[0][2];
```

And finally, using parentheses to separate the brackets with different purposes from each other, one arrives at the code below, which is syntactically correct.

```
i = (new int[0])[2];
```

To differentiate between these examples, TypeCheckVisitor must be able to differentiate between a, `new int[0]` and `(new int[0])`, which seems unintuitive since they can all be said to be of type `int[]`.

Our solution was to introduce a new artifical type called `new int[]` that indicates that the array is initialized in this very statement and might not be accessible the same way as an old array. However in the case of the third example the parentheses enclosing the expression of type `new int[]` makes it possible to treat it just like any

`int[]`. Therefore, when visiting the node corresponding to a parenthesis expression, an inner expression of type `new int[]` is converted to `int[]`. When visiting a node corresponding to array access, only expressions of type `int[]` are allowed.

6.3 Calculating Stack Size

One interesting bug we had when attempting to calculate the stack size for the JVM was that the stack was incorrectly calculated for some tests. We looked at one of the tests and found the menacing expression to be:

```
ret = num*this.calc(num-1);
```

However, when changing the expression to the following, the stack was correctly calculated:

```
val = num-1;
ret = num*this.calc(val);
```

This lead us to believe that the problem was in the visitor for a Call node in the AST. And sure enough, there we evaluated first the node containing the parameters, and then the node referencing the object which the method belonged to. When changing the order of evaluation of these nodes the bug was solved. This was due to the fact that the evaluation of the parameter node was using 2 stack slots internally to evaluate the expression, however it was only leaving 1 value on the stack after evaluation (in this case since there was only one parameter). The size of the current stack and the largest encountered stack were as follows for the two orders of evaluation:

Out of order:

```
curstack before call: 1 maxstack: 3
curstack after loading params: 2 maxstack: 3
curstack after loading object: 3 maxstack: 3
```

Correct order:

```
curstack before call: 1 maxstack: 3
curstack after loading object: 2 maxstack: 3
curstack after loading params: 3 maxstack: 4
```

Here we see that when generating the code for loading the parameters onto the stack before loading the object, the stack "ceiling" is high enough for the evaluation of `num-1` (which requires 2 slots) to not increase the maxstack value. However, when the loading of the object is generated beforehand the stack count is one more, enough to break the ceiling and increase the maxstack value.