Compiler Report
DD2488 Compiler Construction

Hugo Sandelius
911203-2892
hugosa@kth.se

May 15, 2014

1 Introduction

This report describes a compiler made in the course DD2488 Compiler Construction at KTH in the spring of 2014. The compiler is for the language *MiniJava*, which is described on the DD2488 course web pages.[1]. The language is almost a subset of Java, and has, with some exceptions, the same semantics as Java. This report will describe how to use the compiler, document the code in detail, and discuss the process of writing the compiler and some of the choices made along the way. There is also a section on some tough bugs and mistakes encountered while coding.

2 Usage

The compiler can be compiled from source and packaged into a jar by running `ant jar`.

Run the compiler by doing `java -jar mjc.jar <MINIJAVA FILE>`. This will generate one or more Jasmin assembly files, which will need to be assembled into JVM `.class` files. If you have Jasmin installed, this can be done with `jasmin <ASM FILE>`. Make sure you have a recent version of Jasmin that handles escaping of Jasmin reserved words. Otherwise you will end up with a file named `'<ASM FILENAME>'.class` (including the single quotes), which will not work.

[1] http://www.csc.kth.se/utbildning/kth/kurser/DD2488/komp14/

3 Code documentation

3.1 se.kth.hugosa.compiler

Contains a few general-purpose classes.

3.1.1 Indenter

Class used for indenting the output of AST and symbol table `toString` methods to make them more readable. Only used for debugging purposes.

3.1.2 CompilationException

Superclass to all exceptions that can be thrown during compilation.

3.1.3 Main

The main class of the compiler. It puts all the different passs together to compile MiniJava source code into Jasmin assembly.

public static void main(String[] args)
The main method defined in the `jar`, and thus the entry point of the program. Compiles the MiniJava source code file given in `args[0]` to Jasmin assembly files in its' own directory. Prints an error message on `stderr` and leaves return value 1 if something goes wrong, otherwise silently produces assembly files and leaves return value 0.

3.2 se.kth.hugosa.compiler.ast

This package contains the hand-written abstract syntax tree. Each class in this package corresponds to a type of node in the syntax tree. The nodes are deliberately kept simple, including only their children, the line and column they appear at in the source code, and in some cases any information they contain (e.g. the value of an int literal). They also have methods accepting visits from two types of visitors, and a `toString` method mainly used for debugging purposes. Since most classes in this package are very simple syntax tree nodes, only the more interesting classes will be described here.

3.2.1 Visitor

An interface implementing visits to all the different node types in the syntax tree, to be implemented by varying Visitors.

3.2.2 Exp

Superclass to all expressions in the syntax tree.

3.2.3 Stmt

Superclass to all statements in the syntax tree.

3.2.4 Type

Superclass to all types in the syntax tree.

3.3 se.kth.hugosa.compiler.parser

This package contains the JavaCC parser-generating file, and all files generated by JavaCC. Apart from the files mentioned below, it contains a number of auto-generated support files to MiniJavaParser which is not very interesting and not described.

3.3.1 MiniJavaParser.jj

This JavaCC file contains the lexer tokens, the grammar and the generation of the abstract syntax tree as the grammar is processed. When run through JavaCC, it will generate the MiniJavaParser class.

3.3.2 MiniJavaParser

Autogenerated from MiniJavaParser.jj. Parses an input stream and outputs an abstract syntax tree in the shape of a `Program` object. Will throw a `ParseException` if it encounters invalid syntax.

public MiniJavaParser(InputStream inStream)
Initializes the parser with the `inStream` `InputStream`.

public MiniJavaParser(InputStream inStream, String encoding)
Initializes the parser with the `inStream` `InputStream` and the specified encoding.

public Program parse() throws ParseException
Parses the parser's `InputStream` and returns an abstract syntax tree in the form of a `Program` object. Throws a `ParseException` if it encounters invalid syntax.

3.4 se.kth.hugosa.compiler.symboltable

This package contains classes for generating a symbol table of all the classes and methods in an input file.

3.4.1 SymbolTableCreator

An AST-visitor which creates a symbol table.

public Map<String, ClassTable>

createSymbolTable(Program program) throws RedefinitionException
Creates a symbol table from an abstract syntax tree. Returns a `Map` from class names to `ClassTable`s. Throws a `RedefinitionException` if a name is illegally reused within the same scope.

3.4.2 ClassTable

Description of the symbol table contents of a single class, i.e. its' name, methods and fields. Contains a number of getters and setters for these.

3.4.3 MethodTable

Description of the symbol table for a single method, i.e. its' name, parameters, local variables and return type. Contains a number of getters and setters and convenience methods for accessing and changing these.

3.4.4 RedefinitionException

Thrown when a name is redefined within the same scope.

3.5 se.kth.hugosa.compiler.typechecking

This package contains classes used for typechecking of the abstract syntax tree.

3.5.1 TypeChecker

Class for typechecking an AST and making sure there are no type errors. Apart from the `typeCheck` method, which typechecks an entire AST, individual `visit` methods can be called to typecheck individual expressions.

public TypeChecker(Map<String, ClassTable> classes)
Initializes the `TypeChecker` with a specified symboltable.

public void typeCheck(Program program)
throws CompilationException
Type checks the AST given in `Program` and throws an `UndefinedNameException`, `WrongArgumentAmountException` or `WrongTypeException` if a type error is detected. If no type errors are detected, the method does nothing.

3.5.2 TypeVisitor

A visitor which returns the type of the visited syntax tree node (or `null` for untyped things like statements).

3.5.3 UndefinedNameException

Thrown if some name which is never declared is used.

3.5.4 WrongArgumentAmountException

Thrown if the amount of arguments a method is called with does not match the amount of arguments the method is defined with.

3.5.5 WrongTypException

Thrown if a different type than the expected one is encountered.

3.6 se.kth.hugosa.compiler.codegen

This package contains classes for generating Jasmin assembly code from MiniJava source.

3.6.1 LabelGenerator

A simple class for generating unique labels to be used in the generated source.

3.6.2 JasminAssembler

A simple class for appending individual instructions into one or several output files. Also contains a number of static methods for creating method and type descriptor strings of the kind requires by Jasmin.

3.6.3 StackDepthCalculator

A visitor which calculates the maximum possible required stack depth for a method.

public int calcMaxStackDepth(MethodDecl methodDecl)
public int calcMaxStackDepth(StmtList statements)
These methods returns the maximum possible required stack depth for a method, or list of statements, respectively.

3.6.4 CodeGenerator

A visitor which visits an abstract syntax tree and generates Jasmin assembler code based on it.

public CodeGenerator(String sourceFile, Program program,
Map<String, ClassTable> symbolTable, String outDirectory)
throws IOException
Initializes the CodeGenerator with `program` and `symbolTable` and sets the assembler files generated based on them to be put in `outDirectory`, and marks `sourceFile` as the source file in the generated assembly. Throws `IOException` if any IO operation goes wrong.

public void generateCode()
Actually generate the code based on the constructor parameters.

4 Discussion

Generally in the design of the compiler, I tried dividing it in individual packages based on the different compiler passes to make the compiler as modular as possible. I divided it into four packages for the four compiler passes - `parser` for the parsing, which generated an AST, `symboltable` which generates a symbol table from the AST, `typechecking` which makes sure the program has no type errors, and `codegen` which generates the assembly code.

As suggested by the book I use a visitor pattern, designing different visitors for visiting the AST. This made sure I could visit an expression and have it do the correct thing based on the runtime type of the expression without using `instanceof`. I am very pleased with this choice, as it allowed me a lot of modularity and clean code.

It doesn't entirely just execute each package in one pass, as the CodeGenerator uses the TypeChecker to find out the type of expressions, as it needs type information of expressions to know what code to generate in some cases. This means that the same expression might be typechecked twice - first in the typechecking pass to see if it is valid, and then again in the code generation pass to find out its' type. It would probably be more optimized to save the type information of each expression during the type-checking pass and then use this information in the code-generation pass, but I could not find an easy and clean way to do this. As optimization is not a goal with this compiler, I decided to instead take the simpler approach of doing typechecking twice.

Finding out the right stack size for each method (which needs to be specified as `.limit stack`) proved a challenge. Initially I had just set this to 100, which works for pretty much every non-contrived program, but of course wastes a lot of memory. I solved this by adding a StackDepthCalculator visitor which goes through the syntax tree, keeps track of how much the generated code could possibly add to the stack, and find out the maximum possible stack size during program execution. This is then chosen as the stack size.

4.1 Tools

JavaCC was used for the lexer and parser generation. JavaCC was chosen because it was an LL-parser that seemed simple and well-used. I wanted to use an LL-parser because it seemed simpler and less "magic". How an LL-parser works is easy to understand and is relatively easy, but time consuming, to implement. It required a rewriting of the grammar to make it unambiguous, but this posed no major problem. I was not entirely pleased with JavaCC, it was easy

to use and didn't cause me any major problems, but I feel like the code became somewhat unwieldy and hard to understand, and it was hard to avoid that. A clearer distinction between grammar and AST generation would have been nice.

Tools exist to automatically create syntax trees from JavaCC grammars, but I decided against using that, and instead create my tree manually. Creating the AST manually meant it became more separated from the syntactic details, and I could use more descriptive names for children of nodes. The downside was the menial and boring work of creating about 50 AST classes, although my IDE helped a bit with that. I also had to write code to generate the AST while parsing the grammar in my JavaCC file, which uglified things quite a bit.

For my development I decided to start using IntelliJ IDEA Community Edition as IDE, having heard a lot of good things about it. This is the first project I have done using IntelliJ, previously using Eclipse for Java development, and I am very pleased with it and will certainly use it for Java projects in the future.

For version control I used git, which I have a lot of experience with and personally like a lot.

5 Bugs & Mistakes

There were a few bugs I was stuck on longer than usual.

Multidimensional arrays
My main problem here was understanding what was wrong. Kattis told me the noncompile *multidim.java* didn't pass the tests, but I thought multidimensional arrays was already not allowed by the grammar. Eventually I realized that `new int[5][2]` is valid syntax, since it accesses index 5 on a newly created int array of size 5. After that I made the parser throw an exception if you try to make an index access on a newly created array, but this failed another multidimensional test on Kattis. This took me even longer to understand, but after being on the brink of reporting an erroneous test in Kattis, I realized that (new int[5])[2] should be allowed, and wasn't by my parser, since I didn't include parenthesis in my parse tree. After including parenthesis, it finally worked.

Object initialization
Here the problem was that I didn't read the JVM bytecode specification close enough. I didn't realize that even if an object only has an empty constructor (as all constructors are in MiniJava), it still has to be called for the object to be properly initialized. After looking up the (very good) Oracle JVM bytecode specification and reading it carefully, I found the error. Perhaps the lesson to be learned here is to make sure you implement the correct thing before bug searching your implementation.