MiniJava Compiler

A compilator construction project

CHRISTOPHER TELJSTEDT & CARL ERIKSSON

Stockholm 2014

Compilator Construction
School of Computer Science and Engineering
Kungliga Tekniska Högskolan

1 Introduction

This is an documentation report for a MiniJava compiler written in the komp14 course at the Royal Institute of Technology (KTH). MiniJava is a subset of Java and this particular compiler is based on the MiniJava language described in the course book [Appel] but with some modification made by the course leader Torbjörn Granlund. The compiler itself is written in Java and compiles into code runnable by the Java Virtual Machine (JVM) backend.

2 Building

The MiniJava compiler is built with the Ant build system. To build the compiler run:

```
# ant jar
```

This will create a runnable jar file in the root catalog of the project named `mjc.jar`.

3 Running

To run the compiler use the following command:

```
# java -jar ./mjc.jar <path to file to compile>
```

To list possible commands for running the compiler just run it with no in parameters

```
# java -jar ./mjc.jar
```

4 External dependencies

The following external dependencies are necessary for the compiler to work

4.1 JavaCC

JavaCC (Java Compiler Compiler) is a parser generator and lexical analyser generator for the Java language. JavaCC generates a parser from a formal grammar written in Extended Backus–Naur Form (EBNF) notation and outputs Java source code. The parser is generated top-down which limits it to read input Left to right, and constructs a Left-most derivation of the sentence, i.e. LL(k) class of grammars. Hence, left recursion cannot be used.

JavaCC is licensed under a BSD license.

Our toughts: One of the feature we liked about JavaCC from the start was that is had both parser and a generator in the same tool. Furthermore, LL felt more intuitive than LR (left to right, right most derivation), so the conversion from the MiniJava grammar to EBNF would be easier. Another thing we liked the number of example grammar file available from the course catalog. This made it easy to start and reverse engineer different grammar. As a bonus we

discovered that Apache Ant had built support for JavaCC, thus making the programming and testing less tedious. In otherwords a big time-saver.

4.2 Apache Ant

Apache Ant (Ant) is a software tool for automating software build processes, similar to its predecessor Make but is implemented using the Java language. Hence it requires the Java platform, and is best suited to building Java projects.

The most differentiable property between Ant and Make is that Ant uses extensible mark-up language (XML) to describe the build process and its dependencies, whereas Make uses Makefile format.

Ant Apache project is an open source software licensed under a Apache license.

Our toughts: Due to Tigris submit system only support Ant or make as the compiler build system, we chose to use Ant. The reason for using Ant over make, is Ant is made for building java application, where make were intended for c-lang family(although it can be used for other languages as well). Another feature we liked about Ant was the support for JavaCC. This made it easy to generate new parser on each build.

5 Implementation

5.1 Overview

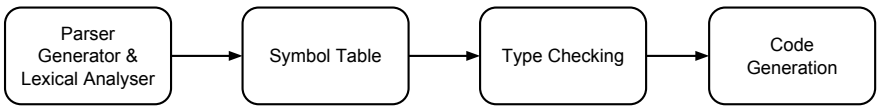

Figure 1: Compiler overview

5.2 Lexical parser

The first step of the compiler incorporate a lexical analyser and parser to decipher files containing programs written in MiniJava programing language. JavaCC was used for this purpose since it handles both steps.

The lexical analyser breaks the MiniJava code into a sequence of tokens, such as:

"`public`", "`static`", "`void`", "`main`", "`(`", "`String`", "`[`", "`]`", "`args`" , "`)`"

Then the lexical analyser also identifies the *kind* of token; in our example the sequence of:

```
PUBLIC, STATIC, VOID, MAIN, LPAREN, STRING, LBRACKET, RBRACKET,
IDENTIFIER, RPAREN, EOF
```

The sequence of tokens is passed to the parser, however all tokens are not needed. In our example, spaces, newlines, t are ignored/skipped and not passed on to the parser.

In our implementation the parser analyses the sequence of tokens and then builds a abstract symbol tree (AST). This is achieved by letting each production it visits return an AST node. Building this AST tree proved to be far more difficult than we previously estimated it to be because of Java operator precedence. For example the "And" operator must bind harder than "Or" operator in order to evaluate and build the sub-tree for an expression correctly. This was solved by recursively exhaust all productions before matching next level of precedence recursively. If the compiler is to be extended with more expressions then both precedence level and associativity must be taken into account.

Since left recursion could not be used in conjunction with an LL parser the trees became right associative. We didn't for-see this as a problem but indeed proved to be problematic for equality and subsequent subtractions, which required to be left associative. This was solved by accumulating the productions passed as parameters into a left associative tree before returning.

5.3 Symbol table

The next step of the compiler was building the symbol table. The symbol table is an intermediate step which the Bytecode generator and type checker is highly dependent on.

The symbol table visitor uses the generated AST from parser to visit each node. It uses the AST the build correct scopes for variables, method and classes. The scopes is stored in tables referencing to their parent scopes, as well as containing variables, methods and classes stored as `bindings` in separate hash maps.

These bindings are mapped with `symbols`. The `symbol` are interned so that two symbols with the same character content contains the internal representation of the corresponding identifier string. This avoids collisions. More information about this structure can be read in Modern Compiler Implementation in Java by Andrew W. Appel.

Each object is also stored with it's corresponding scope in scope lookup table. This gives the advantage of fast and easy lookups during type checking as well as Bytecode generation.

5.4 Type checking

This was done by once again go through the AST with visitors. Each time a variable, method or class (node object) is visited a `symbol` with its identifier is used to lookup in the current scope. It searches for the corresponding `binder` in current scope and all its parent scopes recursively. If the `binding` is not found, it does not exist in scope. However if it is found, it continues on to perform type checks, such as the expected return type and actual return type of methods must be equal.

The most challenging part in the type checking was the inheritance for classes. References to the extensions is stored in the class binders during the symbol table building phase. In the type checker it then checks for cycle inheritance as well as building correct parent scope referencing for each extended classes by traversing recursively through the classes and their extensions. The cycle inheritance is solved by having a visited map that keep tracks of all visited classes.

5.5 Code generation

Lastly, is the Bytecode generation which is done through a visitor as well. It traverses the nodes of the AST. For each vistor function it then generates corresponding Jasmin code writing it to a Bytecode buffer.

To reduce code duplication we made `Bytecode.java`. This is an helper class that contains utility functions such as buffering Bytecode and writing to Jasmin files.

In some cases such as expressions, the type it evaluates was required. It was hard shell to crack, since the return type of the Bytecode generator visitor returns void. In order to solve this the visitor reuses the type checker visitor to get the type of the expression, etc.. However, since the visitor must be in the correct scope it must keep track of the current scope in the Bytecode visitor as well. This is solved by using the scope look up table from the type checker visitor.

6 Files

6.1 Overview of all the folders and files

```
./
├── build.xml
├── DESC
├── lib
│   └── javacc.jar
├── random.txt
├── README.md
├── report.pdf
├── runtigrissubmit.sh
├── src
│   ├── assembler
│   │   └── Bytecode.java
│   ├── error
│   │   ├── CompileError.java
│   │   └── ErrorObject.java
│   ├── javacc
│   │   └── minijava1.0.jj
│   ├── symboltree
│   │   ├── Binder.java
│   │   ├── ClassBinding.java
│   │   ├── MethodBinding.java
│   │   ├── Symbol.java
│   │   ├── Table.java
│   │   └── VariableBinding.java
│   ├── syntaxtree
│   │   ├── And.java
│   │   ├── ArrayAssign.java
│   │   ├── ArrayLength.java
│   │   ├── ...
│   │   ├── ...
│   │   ├── SyntaxTreePrinter.java
│   │   └── ...
│   └── visitor
│       ├── BytecodeEmitterVisitor.java
│       ├── DepthFirstVisitor.java
│       ├── SymbolTableBuilderVisitor.java
│       ├── TypeDepthFirstVisitor.java
│       ├── TypeVisitor.java
│       └── Visitor.java
├── testcases
│   ├── BoubleSortTester.java
│   ├── BoubleSortTester.out
│   ├── Matrix.java
│   └── Matrix.out
└── tigrissubmit.sh
```

6.2 Important files

This section will cover the most important files in the project. Not every file is covered because some is self explaining or they are auto generated from JavaCC. For more detailed information about individual methods, please see the source code.

`src/javacc/minijava1.0.jj`: This file contains the JavaCC grammar. This is where both the parsing and the lexical analysis is done. The abstract syntax tree is the result from this file.

`src/symboltree/Binder.java`: This file holds the core functionality for the symbol table. Its internal structure is based on a hash map. It's the ground functionality for looking up types with symbols. The following classes; `ClassBinding.java`, `MethodBinding.java` and `VariableBinding.java` extends this class.

`src/symboltree/Symbol.java`: Base class for the symbol table. This is the object that is saved in the `hashmap` and contain the necessary information about the symbol that is stored, such as name and type.

`src/symboltree/Table.java`: Table is the class thats used from the outside of the symboltree package. It's main task is to handle insertions and lookups in the internal tables.

`src/assembler/Bytecode.java`: This file is called from `BytecodeEmitterVisitor.java` for each step in the AST walk through. The file holds all of the assembly code in a linked list of `String`. When everything is done, this file write down the code to file.

`src/visitor/BytecodeEmitterVisitor.java`: This file goes through the AST and for each step generate the JVM byte code by using the the `Bytecode.java` class.

`src/visitor/SymbolTableBuilderVisitor.java`: Like the name says, this class is used for creating the symbol table. It extends the class `DepthFirstVisitor.java` for visiting the the AST and the generated symbol table is available from the variable `tableStack`.

`src/visitor/TypeDepthFirstVisitor.java`: This AST visitor checks that varaibles that is used is avaible from the current scrope or a parent scope. If an error is found, the end user will alerted using the `error` package.

A Bugs

In the following appendix we're going to list some of the major bugs and design traps/flaws that we encountered making of the compiler.

Abstract Syntax Tree: We had some problems to get the AST right. One of the problem that we had was precedence levels of operators such as `OR` operator was binding harder than `AND` operator. Furthermore, we had problems with right and left associativity. Since left-recursion could not be used in a LL-parser the first solution that circumvented this resulted in right associativity. This was a big issue since especially equality and additivy evaluated incorrectly in expressions. This was solved by passing the right hand side as a parameter recursively and accumulating the expression and building a left-associative sub-tree in the callbacks.

Symbol table: One major bug we had as well was conflicts between shared identifier names leading to incorrect scope lookups. We solved this by separating variables, methods and classes into separate hashmap structures. This resulted in having to specify which hashmap an object should be inserted to.

Long extension: Another major bug that we had, which resulted into a major re-factor of the code was that we forgot that long requires more stack space than integer. Furthermore, they required one extra index slot thus leading to errors such as `register pair x/z contains wrong type` which was a hard bug to debug. The solution was to allocate two blocks for each long on the stack as well as increasing the offset indexes accordingly.

Code comments: A bug that got found with the student case suite was comment with EOF on the last row. Although it was not to hard to fix, it was a special case that we did not think of when we made the grammar.