MiniJava Compiler (`mjc`)

DD2488 – Project Report

Daniel Månsson
dmans@kth.se

Elvis Stansvik
stansvik@kth.se

15 May 2014

Abstract

In this brief report we introduce `mjc`, a compiler for the MiniJava language targeting the Java Virtual Machine (JVM). The compiler was built as part of the course *DD2488 Compiler Construction* at KTH. We begin the report with a description of the compiler design, followed by instructions for building and running the compiler, and conclude with some thoughts on possible future improvements.

Contents

1 Introduction

MiniJava is a subset[1] of the Java programming language presented in [1] The original language supports the built-in types `int`, `int[]` and `boolean` as well as user defined class types. There is no support for inheritance or method overloading. Operators `+`, `-`, `*` and `<` are defined on integers. Operators `&&` and `!` are defined on booleans.

`mjc` is a compiler for a slightly modified and extended version of MiniJava [3], adding support for operators `<=`, `>`, `>=` on integers; operators `==` and `!=` on integers, booleans and references; operator `||` on booleans; `if` statements without an `else` clause; array bounds checks and nested blocks with new variable declarations. The compiler targets the Java Virtual Machine (JVM) by producing Jasmin [4] pseudo-assembly code as output.

We begin this report with an overview of the design of `mcj`, followed by some instructions for building and running the compiler. We conclude with some thoughts on possible future improvements.

2 Compiler Design

The compilation process in `mjc` proceeds in four phases. The process begins with the input of MiniJava source code and ends with the output of Jasmin assembly code. Each intermediate phase takes its input from the preceding phase and gives as output the input for the next. Figure 1 shows an illustration of the compilation phases.

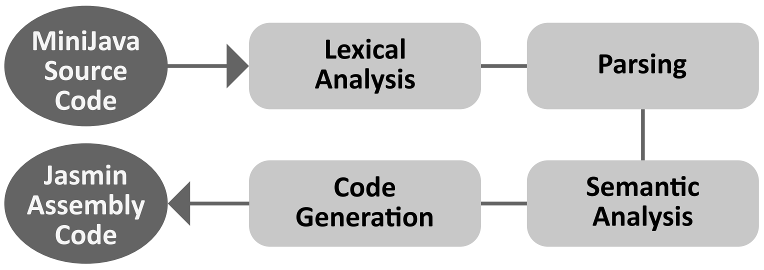

Figure 1: Compilation phases in `mjc`.

In the following subsections, we'll briefly go through each compilation phase and explain it in general terms. We'll also explain design decisions we made and any challenges we faced in constructing `mjc`.

[1]Not exactly true: A Java compiler checks that variables are initialized before use, while in MiniJava this is simply undefined run-time behavior.

2.1 Lexical Analysis and Parsing

Compilation begins by breaking the input byte stream into a stream of terminal symbols (tokens) and parsing the resulting token stream into an abstract syntax tree (AST). The lexical analyzer (lexer) must check that the input consists of valid MiniJava tokens. The parser must check that the resulting token stream is part of the formal language described by the MiniJava grammar.

We decided to use the SableCC [2] parser generator for lexical analysis and parsing in `mjc`. SableCC takes as input a file containing a set of regular expressions describing valid tokens, and an LALR grammar describing the input language. The output is a complete lexer and parser, along with some abstract visitor classes for traversing the AST.

Codifying the provided MiniJava grammar as a SableCC input file was fairly straightforward. To simplify later stages of compilation, a depth first traversal of the AST should correspond to the precedence rules of MiniJava operators. This took some careful composition of the productions for expressions.

We also wanted the parser to reject input such as **`new int`** `x[2][1]` instead of interpreting it as the creation of an integer array followed by an access of its second element. To accomplish this we had to introduce an additional production for primary expressions, one that excluded array creation expressions, and use that in the production for array accesses.

2.2 Semantic Analysis

Past lexical analysis and parsing, we know that the input program is lexically and syntactically correct. But other errors such as uses of undeclared identifiers and type errors may still exist in the program. During semantic analysis, the compiler traverses the AST looking for such errors.

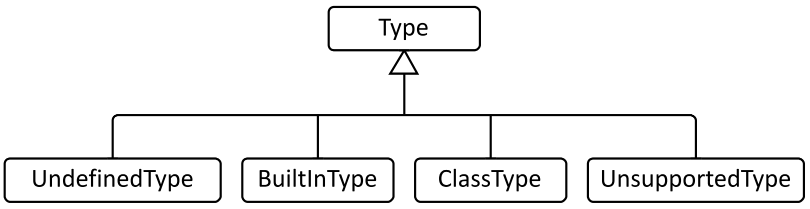

Figure 2: Type class hierarchy.

To represent built-in types `int`, `int[]` and `boolean` as well as user defined class types, we came up with the class hierarchy shown in figure 2. The abstract base class `Type` specifies methods such as `isAddableTo(Type)` to be implemented by subclasses. Each of `int`, `int[]` and `boolean` is represented by a static instance of `BuiltInType`. `UnsupportedType` is used for the `void` and `String[]` "types". These exist grammatically in MiniJava but lack meaningful semantics. `UndefinedType` will be explained shortly.

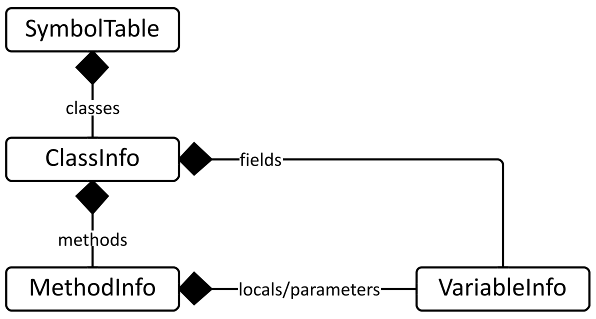

Figure 3: Symbol table class composition.

Semantic analysis begins with the construction of a *symbol table*. For each class name, method name and identifier found in the program, the symbol table stores information such as the name and type of the symbol as well as the source code location where it was declared. Figure 3 shows the composition of the classes we used to represent the symbol table.

`MethodInfo` deserves some special mention. To handle the adding and look-up of local variables, which may be declared in nested blocks, `MethodInfo` has an `enterBlock()/leaveBlock()` API which is used in conjunction with `addLocal(..)/getLocal(..)` to control the scoping of local variables.

Local variables are kept in a multimap, since two variables may have the same name if the block in which the first one is declared is closed at the point of the second declaration. Blocks inside a method are numbered $0, 1, \ldots$ in the order in which they are opened. Internally `MethodInfo` maintains a stack of currently open blocks. `enterBlock()` pushes the next block (taken from a counter) on the stack, while `leaveBlock()` pops the stack.

Each `VariableInfo` holds the number of the block in which the variable was declared. `addLocal(..)` sets the block number of the added `VariableInfo` to the current block (top of the stack). `getLocal(..)` performs a look-up by first consulting the multimap for all variables with the requested name, and then for each such `VariableInfo`, checking if the variable is in scope by searching for its block number on the stack. The first `VariableInfo` with a block number currently on the stack is considered a match. This essentially makes local variable look-up an $O(nm)$ operation in the worst case, where n is the number of local variables in the method and m is the maximum block nesting depth.

Figure 4 shows a simplified illustration of local variables being added to the `MethodInfo` for a small MiniJava method. The mappings in the figure only show the block number for the variable, not the full `VariableInfo`.

```
                            Action        Stack  Multimap
                                          empty  empty
 1 public int f() {  enterBlock()  0      empty
 2   int a;          addLocal(..)  0      (a,0)
 3   {               enterBlock()  01     (a,0)
 4     int b;        addLocal(..)  01     (a,0),(b,1)
 5     {             enterBlock()  012    (a,0),(b,1)
 6       int c;      addLocal(..)  012    (a,0),(b,1),(c,2)
 7     }             leaveBlock()  01     (a,0),(b,1),(c,2)
 8   }               leaveBlock()  0      (a,0),(b,1),(c,2)
 9   {               enterBlock()  03     (a,0),(b,1),(c,2)
10     int b;        addLocal(..)  03     (a,0),(b,1),(c,2),(b,3)
11     b = 3;                      03     (a,0),(b,1),(c,2),(b,3)
12   }               leaveBlock()  0      (a,0),(b,1),(c,2),(b,3)
13   return a;                     0      (a,0),(b,1),(c,2),(b,3)
13 }                 leaveBlock()  empty  (a,0),(b,1),(c,2),(b,3)
```

Figure 4: Adding of local variables to a `MethodInfo`.

The figure only shows the adding of local variables, but as an example, when the symbol `b` in `b = 3;` on line 11 is looked up, `getLocal(..)` will find the mappings `(b,1)` and `(b,3)` in the multimap, and determine that the second one is the one in scope, since its block value (3) is currently on the stack (which contains blocks 0 and 3 at that point).

With these data structures in place, we built a class `SymbolTableBuilder` which traverses the AST in two passes to build the symbol table. In the first pass, information about declared classes is entered into the table. In the second pass the builder goes deeper, adding information about declared fields, methods, parameters and local variables. The reason for the separate passes is that MiniJava allows forward-references to not yet declared classes.

If the declared type of a field, method, parameter or local variable is an undeclared class type, an error is printed and the symbol is entered with the `UndefinedType` type. This type acts as a “chameleon” type in that it is compatible with all other types. This is done to let the type-checker proceed despite the declaration error.

If multiple declarations of the same symbol is encountered, an error is printed. If the re-declared symbol is a class or method, a new unique name is generated for the offending declaration and the AST is updated with the new name. This is done to let the type-checker proceed with type-checking inside the offending class/method.

With the symbol table constructed, the analysis proceeds with type-checking. This involves looking for uses of undeclared identifiers and making sure expressions have the correct types. For this we built a class `TypeChecker` that takes as input the AST and symbol table and visits every such language construct to perform the necessary checks.

The type-checker tries to recover from errors as much as possible. For instance, if an expression has an unexpected type, an error is printed, but the type-checking will proceed as if it had the expected type. If it cannot deduce which type was intended, it will assume the `UndefinedType`, effectively suppressing any further errors involving the expression. This hopefully lets the user focus on the real error without being bothered with secondary errors.

2.3 Code Generation

In the final phase, the compiler generates Jasmin code. For this, we wrote a class `JasminGenerator` which takes the AST, the symbol table and a class implementing the `JasminHandler` interface as input. For each generated class, the `handle(...)` method of the passed in `JasminHandler` is called with the name of the class and the resulting Jasmin code as parameters.

`JasminGenerator` visits each node in the AST and outputs the appropriate Jasmin statements. Since the JVM was originally designed for the Java language, many convenient instructions were available to us, such as `new`, `newarray` and `arraylength`, which all map directly to MiniJava constructs.

Since JVM is a stack-based machine, all operations work with values on the stack. Apart from generating the necessary Jasmin instructions, `JasminGenerator` also keeps track of the current stack size in the currently generated method, to be able to set an appropriate maximum stack size for the method with the `.limit stack` directive.

3 Using the Compiler

3.1 Building

The compiler uses Apache Ant as build system. The default Ant target, invoked by running `ant` in the top-level project directory, builds the compiler, executes the unit tests and produces the compiler JAR file `mjc.jar`. See `ant -projecthelp` for other available Ant targets.

3.2 Running

The compiler is invoked by running the `mjc` shell script in the top-level project directory. The default action is to compile the given input file to Jasmin code files, one for each class, in the current working directory, followed by an invocation of Jasmin on the output to produce `.class` files.

The script accepts the following command line options:

```
usage: mjc <infile> [options]
 -S         only output Jasmin assembly code
 -p         print abstract syntax tree
 -g         print abstract syntax tree in GraphViz format
 -h         show help message
```

4 Future Improvements

Our original goal with this project was to create a compiler that targets the ARM architecture, and which uses an intermediate representation (IR) to increase the portability of the compiler. Unfortunately, due to time constraints, we were not able to complete this compiler, and had to go with the simpler JVM target. A possible future improvement to the compiler is then naturally to continue this work. Other possible improvements include adding support for class inheritance and the `long` integer type.

For those wishing to do further work on the compiler, the source code, including full JavaDoc documentation, is available from the project website at `http://estan.github.io/mjc/`. The code is organized as follows:

- `mjc.types` contains classes for representing MiniJava types.
- `mjc.symbol` contains the symbol table classes.
- `mjc.analysis` contains visitor classes for the semantic analysis.
- `mjc.jasmin` contains the Jasmin assembly code generator.
- `mjc.error` contains a couple of classes related to error reporting.
- `mjc.JVMMain` is the compiler main program.

References

[1] Andrew W. Appel. *Modern Compiler Implementation in Java*. 2nd ed. Cambridge University Press, 2002. ISBN: 978-1-139-43496-6.

[2] Étienne Gagnon. *SableCC*. URL: `http://sablecc.org/`.

[3] Torbjörn Granlund and Andrew W. Appel. *Context-free grammar for Minijava variant*. URL: `http://www.csc.kth.se/utbildning/kth/kurser/DD2488/komp14/project/grammar14v1b.pdf`.

[4] Jon Meyer and Troy Downing. *Jasmin – Assembler for the Java Virtual Machine*. URL: `http://jasmin.sourceforge.net/`.