MiniJava Compiler Project Report

KTH DD2488 Compiler Construction

Hampus Liljekvist
Christian Lidström

May 19, 2014

Contents

Chapter 1

Introduction

1.1 Background

As the need for good software rises, so does the need for understanding the process behind compilation. In order to write proper code you must grasp what actually goes into creating a runnable program from source code, and a useful approach to learning this is to do the opposite of abstraction.

This project thus aims to create a fully functioning MiniJava compiler to generate Jasmin assembly code. Additionally, tests will be written to verify the functionality of the program. The *Tigris* [2] judging system will provide the full test suite that passes or fails the compiler. The project is based on the MiniJava project by Andrew W. Appel [3], and collaboration and versioning is done through *GitHub*.

Through getting familiar with the basic principles of compilation, more robust and efficient code can be written when developing software in the future.

1.2 Usage

The compiler is built by standing in the root folder and invoking the keyword `ant`. A jar file called `mjc.jar` will be created. The first argument to this file should be the source code file to compile, which will cause the program to syntax check it. If code generation is desired, then the compiler must be invoked with the flag `-S` as the second argument.

1.3 Functionality

The compiler will syntax check MiniJava code according to the grammar specified at the course page [1]. The generated output will be Jasmin byte code instructions, which in turn can be compiled by Jasmin to create a proper runnable Java program.

1.3.1 Extensions

The compiler generates code for the following extensions:

- **JVM:** Jasmin backend

- **ISC:** Inheritance syntax check
- **IWE:** `If` without `Else`
- **NBD:** Nested blocks with new variable declarations
- **CLE, CGT, CGE:** Comparison operators `<=, >, >=`
- **CEQ, CNE:** Comparison operators `==, !=`
- **BDJ:** Logical connective `||`

Additionally, the compiler can syntax and type check the following extensions without code generation:

- **LONG:** Long integer type
- **ICG:** Inheritance code generation

Chapter 2

Technical Overview

2.1 Lexing

For the lexing and parser creation stage stage *JavaCC* was chosen. This was based on JavaCC being fairly well used, and after the initial researched it seemed fast to get going with. Since there were no previous experience using tools like these, any choice would need a certain amount of research. $LL(k)$ parsing is also rather intuitive, even if slightly limiting. In retrospect it might have been better to choose a more powerful $LALR$ parser, but JavaCC had made it easy to write the grammar. It took some research to handle the precedence order of operators, and to remove left recursion.

2.2 Parsing

JavaCC uses the provided code skeleton of classes when building the parser. This way it was mostly a matter of translating the grammar to the JavaCC syntax in order to get it to work. Few checks are made in this stage over just grammatical correctness, but the compiler does handle the size of integers and multidimensional arrays here.

2.3 Abstract Syntax Tree Creation

The abstract syntax tree (AST) is built using the *visitor* pattern, as described in Appel's book. A basic framework to visit all the syntax tree classes of the code skeleton was provided as `DepthFirstVisitor.java`, which made it easy to start working on this part. The various classes are visited, starting at the `Program` node, in a depth-first manner. When the relevant classes and method are visited they are saved in the symbol table, which is of the functional type, meaning it is never destroyed and all classes/methods are saved. At first an imperative approach with a stack was tried, but the book did not describe the process well enough to get it working as expected. The functional symbol table made more sense, even if it might be more memory heavy.

To mitigate some of the inefficiencies when dealing with `String`s in Java, a `Symbol` type is used to keep track of the various identifiers. This class creates one unique `String` and hashes it upon first usage, then the subsequent usages avoid allocating new `String` objects by reusing the hashed one. Even if this makes sense in theory, it is hard to justify the extra effort of constantly having to use the `Symbol.symbol(String)` method when dealing with identifiers, but this was chosen as it was recommended in the Appel book.

A custom hash table is made up of `HashT.java` and `Table.java`. This basic class work as an ordinary hash table, and is implemented as described in Appel's book. The size of the `Buckets` where decreased, considering how few items are stored in each table. Upon building the tree, variables are bound to their types using `Bindings`, a simple class to hold the relevant data.

Separate tables are used for classes, methods and blocks. When looking for a variable, the block is first consulted, then the method, and finally the class. Helper methods such as `varInScope(Symbol)` are used. The current class, method and blocks are always stored in the variables `currClass`, `currMethod` and `currBlock` for all visitors. This information is needed as you don't know where you are when vising nodes in the AST, and they are updated upon visiting new `ClassDecls`, et cetera.

`Blocks` are handled by using a `blockId` variable which are increased for each new block in a method. This id is used as a unique name to refer to a given block, and are used in later stages to retrieve them from the symbol table. Each block has its own local variables, and if blocks are nested, they are recursively looked up in the outer blocks.

2.4 Type Checking

The program is visited a second time using `TypeDepthFirstVisitor.java`. In this phase all type checking is performed by looking up the generated return types, and comparing it to the entries in the symbol table. Helper methods such as `getVarType(Symbol)` and `getClassNameFromVar(Symbol)` are used to find entries based on identifiers. Every `visit` method returns the corresponding type of the AST node, or `null` if the node lacks a proper type.

`Calls` are the hardest AST nodes to handle. They require rigorous lookup and type checking which differs depending of the type of the expression they where called on, which has made the `visit(Call)` method the most code heavy in the class.

2.5 Jasmin Code Generation

The AST is visited a third time to generate the corresponding *Jasmin* assembly code for each node in the tree. This visitor is called `JasminVisitor.java` and is built on the skeleton of the type visitor, since it is required to know the type of expressions and statements here as well.

To avoid filling this visitor with the actual `Strings` to print, a separate file is responsible for all the code generation, called `JasminFileWriter.java`. This class contains methods corresponding to different Jasmin structures such as method declarations and jumps. Java's internal `StringBuilder` class is responsible of accumulating the string which will eventually be written to file, as appending to regular `Strings` is expensive in Java due to immutability. When the code for a class has been accumulated, a file called `<classname>.j` will be created and the `StringBuilder` reset.

Jasmin require that you specify the stack depth for each method, and even though it is possible to set an arbitrarily high value for it, the exact stack depth needed must be calculated per course instructions. This is done while traversing the tree in `JasminVisitor` by having a variable `stackDepth` that is increased and decreased based on the stack usage of the generated Jasmin instructions. `stackDepthMax` holds the maximum stack depth so far, and is updated if it is exceeded.

Branching instructions are given different labels based on which Jasmin structure they belong to, which are kept unique through having a `branchId` variable which is increased for each AST node that needs it, and reset for each new method.

A basic code skeleton for Jasmin code generation was given, which aids in knowing how method calls are variable loading should be handled, for instance. This is used by creating `VMAccess`es for variables and `VMFrame` and `VMRecord`s for methods and classes, coupled with other helper files such as `Hardware` to generate the proper instructions based on Jasmin types. This information needs to be stored in the symbol table as well, since it may be used at various places in the tree.

2.6 Error Handling

Error handling is done through `ErrorHandler.java`, a class which contains `enum`s for different error codes such as `TYPE_MISMATCH`, `SELF_INHERITANCE` or `NOT_FOUND`. When an error is encountered during parsing, an `ErrorMsg` object is created with the corresponding error code and a supplied error message. These errors are then stored as they arise in the program. If errors are encountered in a visitor, the compilation is aborted by skipping the remaining visitors. It then finally exits with exit code 1, as required by the Tigris specification.

If an error in the compiler is encountered that is not based on the grammar of the parsed source file, or if an unhandled Java exception is caught, an `INTERNAL_ERROR` is generated. In an ideal world, this `ErrorMsg` should never be generated.

2.7 Debugging

All the visitors and many of the helper classes has a global `DEBUG` variable which can be set to `true` to enable printing of debug information. The primary form of debugging is to simply print out which nodes are visited in the AST, and if a certain value is found or not, for instance. When many `DEBUG` flags are set, the printed output can be hard to read. Therefore it is recommended to not debug too many classes at once.

A more rigorous approach would be to use proper assertions in the program code, but for the scope of this project there was not time to implement this degree of debugging.

Chapter 3

Class and Method Descriptions

3.1 Packages

3.1.1 error

This package contains the classes `ErrorHandler` and `ErrorMsg` which are responsible for error handling in the compiler. `ErrorHandler` houses the error message codes as `enums`. To report an error in the error handler, the method `complain(String, ErrorCode)` is used. To print them, `printErrors()` is used.

3.1.2 frame

This package contains interfaces and general classes for Jasmin code formatting, see the `jvm` package for the implementation. For the Java virtual machine, `VMAccess`, `VMFrame` and `VMRecord` are the appropriate interfaces to inherit from.

3.1.3 jasmin

This package contains the classes and methods for printing the generated Jasmin code, using the `JasminFileWriter` class. Most methods are named after the corresponding Jasmin instructions/code structures such as `newArray()` and `printInt()`. The method `createSourceFile(String)` take the class name as parameter and prints the accumulated `String` in the `StringBuilder sb` to file. After creating the file, the `StringBuilder` is reset. Various helper methods are used to aid the generation of Jasmin code.

For information about Jasmin, see the relevant documentation.

`JasminReservedWords.java` is a file which gives static access to most reserved words in Jasmin code that cannot be used for variable names, method declarations and similar statements. The method `reservedWord(String)` will indicate whether the given `String` is reserved or not. This class is used to escape those words, should they be used by the parsed MiniJava source code.

3.1.4 jvm

This package contains all code used to generate the various Jasmin instructions for the generated source files. `OnHeap` represents heap variables such as fields, while `IntegerInFrame` represents a local integer variable. `Hardware` formats method signatures based on types. `Frame`s and `Record`s also have their corresponding classes.

3.1.5 parser

This package contains the MiniJava grammar according to the JavaCC syntax. It also contains all the classes generated by JavaCC for the lexer and parser (these files are always regenerated upon running `ant`). `MiniJavaLexerParser.jj` is the file which has the grammar specification with all the handled tokens.

For information about JavaCC, see the relevant documentation.

3.1.6 symbol

This package contains the classes used to create the symbol table of the compiler, starting with `SymbolTable` which holds all the class tables. `BlockTable`, `ClassTable` and `MethodTable` are all similar, but kept distinct to differentiate between variables in the corresponding scopes. In practice, scopes are handled through different hash tables of type `HashT`. `Table` is a helper class which is used to interact with the hash tables through methods such as `put(Symbol, Object)` and `get(Symbol)`. `Bucket`s are the internal containers for the hash table, and `Binding`s are used to map variable names to types.

The scope tables contain all the necessary information used to syntax check the source files, and stores local variables and accesses, for instance. Typical methods to use are `getMethod(Symbol)` in the class table to get the corresponding method table from an identifier, or `getVar(Symbol)` to get a binding for a given variable.

3.1.7 syntaxtree

This package contains all the Java classes which represent the different nodes in the AST, as well as type representations and abstract classes. Due to the large amount of classes in this package, the readers are encouraged to study these classes themselves to see which sub nodes are saved in them. As an example, `Print` only contains the expression `Exp` which should be printed, while `BooleanType` is a representation of the boolean type used for type checking.

3.1.8 visitor

This package contains all the visitors which traverse the AST during compilation, and thus represents the core of the compiler. The visitors use the *visitor* pattern to visit the nodes the AST in a depth-first manner. This is achieved through `visit(<class>)` methods for all these nodes, and every class in the tree has a corresponding `visit` method which is called. For the type visitor, these methods also return the type.

`DepthFirstVisitor` will build up the symbol table during traversal and check if variables are defined and not double declared. Some checking must be done in later visitors, due to the depth-first nature of the visitor. `TypeDepthFirstVisitor` does all the type checking with various helper methods such as `getVarType(Symbol)`. `JasminVisitor` is responsible of the Jasmin code generation.

There are additional visitors such as `ASTPrintVisitor` and `PrettyPrintVisitor` which can be used to print out a graphical representation of the AST, but they are not used for compilation. The package also contains the interfaces that define the visitors.

3.2 Main Class

3.2.1 JVMMain

This class is the entry point of the compiler and thus contains the `main` method. This main method in turn contains the code to interpret the user input and compilation flags, and creates instances of the needed visitors. It also creates the generated parser `Program` and the `SymbolTable`, as well as the `ErrorHandler`. The `ASSEM` variable is set to `true` if the command line argument `-S` is given as the second argument, which will cause the `JasminVisitor` to be run and generate the Jasmin code.

If any errors are encountered during compilation, they will be printed at the end of the `main` method and the program will exit with status 1.

Chapter 4

Appendices

4.1 Encountered Problems and Bugs

4.1.1 Hash Tables

The compiler uses custom hash tables as specified in Appel's book, which may have a negative impact on the compiler's performance. It would probably be better to use Java's internal `Hashtable` class instead.

4.1.2 The Symbol Table

As methods where added during the compiler construction, the symbol table grew to become very unwieldy. Due to lack of previous experience it was hard to predict what would be needed to save in the table, and this was constructed as the coding went by. Another issue was the `VMAccesses` and `VMFrames` which needed to be saved for the `JasminVisitor`, which causes duplication when the same information is saved in different forms. Formal arguments are also saved in both ordered and unordered manner, a mistake as a result of demands arising along the way.

4.1.3 Handling Bad User Input

Problems where had with bad user source files which caused the compiler to crash. It is hard to predict all the errors that can arise from bad user input, which leads to various unhandled exceptions.

4.1.4 Null Pointers

Lots of exception and problems where related to null pointers, as a result of expected information not being saved in the symbol table. This has been hard to debug, as the visitor stack trace can be hard to interpret.

4.1.5 Blocks

The extension which allows new variable declarations in blocks has been hard to implement, and it has gone through many different iterations. It was hard to handle the scope of nested blocks, and lots of problems where related to blocks getting too big, too small or simply incorrect scope. The nested expressions in the blocks caused the scopes to be lost, and the solution was to save more information locally in the blocks' `visit` method. To retrieve the blocks in later visitors, the fact that the AST is always visited in the same order was used, through having a block id being increased for each new block.

4.1.6 Type Checking

Some AST nodes were hard to type check, as the type is not known until after the visit of the nested nodes. This caused problems for the long integer type, where different code should be generated based on if the visited expressions in a `Plus` nodes where of type `long` or `int`, for instance. If an integer literal was visited in a long `Plus` node, then corrections must be done after the fact when the type is known.

Once you are in a certain node in the tree, it is impossible to know what type of housing node it is contained within, which caused problems and required some re-thinking.

4.1.7 Method Calls

The `Call` AST node was by far the hardest to handle. You can't easily know what AST node the `Call` was made upon, causing lots of `instanceof` checks and type casting with different behaviour. It would probably have been better to save more information for `Call`s, at least the name of the class which contains the method declaration.

4.1.8 Comments

There were two annoying bugs related to the parsing of multiline comments. While these bugs certainly could be considered trivial in most aspects, they were among the hardest to actually locate. The first of these was that two different multiline comments were parsed as one, including everything in between, resulting in parts of the program code being discarded. The second one, which was a little bit more obscure and thus only showed up in a single test case, made the compiler fail when the comment ended on a line containing an even amount of subsequent star symbols.

References

[1] *Project in DD2488, Compiler Construction (komp14)*, KTH CSC, DD2488
`http://www.csc.kth.se/utbildning/kth/kurser/DD2488/komp14/project/`

[2] *The TIGRIS testing system*, KTH CSC, DD2488
`http://www.csc.kth.se/utbildning/kth/kurser/DD2488/komp14/tigris/`

[3] *Modern Compiler Implementation in Java: the MiniJava Project*, Cambridge University Press, Book Resources
`http://www.cambridge.org/resources/052182060X/`