DD2488
Andreas Tarandi
890416-0317

MiniJava Compiler Report

May 12, 2012

1 Introduction

This document describes the design process and decisions behind my compiler for the minijava language. The compiler compiles minijava into jasmine (JVM) assembler.

2 Lexer and Parser

For the lexer and parser I used JavaCC, a LL parser code generator. I decided to use LL since it feelt more natural than LR and the process of converting the grammar to LL was not too tedious. In some places I had to move away from LL(1) grammar (in favour for LL(k) grammars), but in most cases LL(1) sufficed.

The thing that caused most problems writing a LL grammar was inserting new features into the complete code. Since the grammar had to be broken down into small pieces for the LL conversion and what more for keeping the correct order of operations it was at times difficult to insert new grammar into it.

3 Abstract Syntax Tree

The syntax tree created by the parser (in the code called ”basic_tree”) is more complex than is necessary, and is all but a tree representation of the original language. To simplify future change of source language, and make the rest of the language processing easier the basic tree is converted to an abstract syntax tree (AST).

This conversion is done in the syntaxtree.AbstractTree class which recursively converts the tree structure of the original tree.

4 Type checking

For type checking and binding both, the visitor pattern is used. Type checking is done in two steps; first all definitions are handled: classes are defined in the program, methods defined in the methods and variables and formals defined in the methods. In this step the compiler catches any double declarations and like errors.

In the second step the classes are assigned to JVM records, the methods to JVM frames and the variables to accesses. All method calls are also bound to the corresponding method, and the same with classes and symbols. All remaining type checking is done in the second step.

Another thing worth of notice is that in the second step all the methods in the visitor call returns the type that subtree evaluated to.

4.1 Symbol tree

The symbol tree used in the type checking and binding is a stack onto which the scopes are pushed and pop as the code traverses the tree. This has the advantage that only the currently active variables are kept in the symbol table, which increases the lookup speed for symbols, and in turn the performance of the compiler. This is only used for local variables, since class variables, methods and classes can be accessed outside of their declared scope. If a variable is not found in a scope the code proceeds to check the scope below in the stack and so on.

5 Assembling

Since I made the decision to target JVM the process are much easier than would I have targeted a real processor platform, therefore no IR-trees or register handling are needed and I can directly proceed to assembling.

This was done by once again by using the visitor pattern. Since jasmine require one to specify the maximum stack size for each method the assembling is also done in two passes; a first that doesn't write any output but just counts pushes and pops to the stack and calculates the maximum stack size needed, and a second pass that actually outputs assembler. This way I achieve an almost optimal stack limit. Otherwise the assembling part of the compiler is straight forward; it simply translate the AST into jasmin.

6 Extensions

Besides implementing the given base language I have implemented the following predefined extensions:

- Nested block declarations, allow nested blocks with new variable declarations.
- Comparison operators (all of them)
- Inheritance syntax check and code generation
- Long: The primary type long
- Logical or: The ||-operator

I have also added the following non-standard extensions:

- Generic arrays: Support for arrays of any allowed base type
- Declaring methods void: This includes making return statement optional in these methods
- Expressions as statements: Allowing expressions alone as a statement without anything to catch the output.
- Basic method overloading: Method overloading works both with and without inheritance.

7 Code overview

In this section I will provide a brief overview of the packages in this project, with description of the most important or distinct classes in the package.

7.1 mjc

This is the package where the entry point for the program is.

- **JVMMain** The main class for compiling to jvm

7.2 error

Package containing classes for error and exception handling

- **ErrorMsg** A class handling compile errors and warnings
- **InternalError** Error thrown on internal errors (errors in the compiler as opposed to errors in the compiled code)

- **TypeException** Superclass for type exceptions
- **ImplicitCast** Exception thrown when an implicit type cast occurrs, subclass to TypeException
- **LossOfPrecision** Exception thrown when loss of precision would occur if an implicit type cast where done, subclass to TypeException

7.3 basic_tree

This package contains the classes that the parser translates the raw code into. Each class represents a language construct.

7.4 syntaxtree

This package contains the classes for the abstract syntax tree.

- **AbstractTree** This class contains the code for converting the basic tree into an abstract syntax tree.

7.5 symbol

Classes for handling the symbol table

- **Scope** Abstract tree that all classes in the syntaxtree that are scopes should inherit, contains code for symbol store and lookup.
- **SymbolTable** Class handling all the scopes, including additions and lookups into them.

7.6 visitor

Containing visitors for traversing the syntax tree. This is where the main processing is done.

- **Visitor** Base class for visitors
- **TypeVisitor** Base class for type visitors
- **TypeDefVisitor** Pass one of type check, adds definitions for classes methods etc, and adds symbols to all variable definitions.
- **TypeBindVisitor** Pass two of type check, links up all variable uses with corresponding symbol and creates VM records, frames and accesses.
- **AssemblerVisitor** Creates and outputs the final assembler.

7.7 parse

Generated classes from minijava.jj

7.8 frame

Interfaces for low level constructs.

7.9 jvm

Classes implementing the interfaces in frame with JVM specific code, and helper classes for JVM. All the classes in this package have rather self explanatory names.

- **Hardware** Helper class for generating JVM implementation details