DD2488
Andreas Tarandi
890416-0317

Complier Report

May 9, 2012

1 Introduction

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

2 Lexer and Parser

For the lexer and parser I used JavaCC, a LL parser code generator. LL feels 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, 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 convertion and what more for keeping the correct order of operations it was tedious to insert new grammar parts into this. This is the main reason why I have refrained from implementing the logical or extention.

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 convertion is done in the syntaxtree.AbstractTree class which recursivly converts the tree structure of the original tree.

4 Typechecking

For typechecking and binding both the visitor pattern is used. The typechecking is done in two steps; first all definitions are handled: classes are defined in the program, methods defined in the methods and variabels and formals defined in the methods. In this step the complier 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 variabels to accesses. Also all method calls are bound to the corresponding method, and the same with classes and symbols. All remaining typechecking 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 evaled to.

4.1 Symbol tree

The symbol tree used in the typechecking 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 stored, which increases the performance of the compiler. This is only used for local variables, since class variables, methods and classes can be accessed outside of their 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 proccessor platform, therefor 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 stack size required 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 stacks, and a second that actually writes this. This way I achieve an almost optimal stack limit. Otherwise the assembling part of the compiler is nothing special; it simply translate the AST into jasmin.

6 Extentions

Besides implementing the given base languange I have implemented the following predefined extentions:

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

I have also added the following non-standard extentions:

- 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.