JLite Compiler Report

CS4212 Compiler Design

Andrea Taglia

November, 2018

Index

Introduction

This is a compiler for Jlite programs. It is fully written in Java and includes both a frontend and a backend.

The compiler frontend aims at isolating in the best possible the way all the programming language details so as to isolate from the backend compiler work. This allows any backend compilers to be used as it can just take care of the machine code details. This way other backend or frontend compilers can be plugged in instead of the one being currently used.

The compiler is divided into three main parts. Each of the following chapter briefly describes each part.

Lexer & Parser

The grammar of jlite can be found in the pdf file `assignment1_text.pdf`.

The Lexer has been created using the jflex tool which creates java classes starting from a .lex specification file. The Parsers has been produced using Java Cup, which produces java classes out of a .cup specification file. The Main class starts the parsing process using the generated classes as described above taking an input file name from command line argument.

Other than just parsing, the parser builds the whole AST. Although the AST is not of great use so far, it will be necessary for the next compilation step. The AST nodes can be found at path `jnodes/` and the root node is `JProgram.java`. Every node has a unique toString() function which is forced to implement being the class extending the abstract JNode. The root toString() will just call in chain all the nodes toString() function to end up with a printed version of the AST. However, it has been preferred to just print out the parsed program from the lexer. The AST will be used later on as mentioned above.

The reference JLite grammar was not well formatted and threw a lot of shift/reduce and reduce/reduce conflicts at first. All of them have been resolved by rearranging the grammar so as to avoid ambiguity. One exception has been seen on the production involving `ClassDecl` where java code has been inserted in the parser to force correct behaviour after have loosened the grammar rules.

Static Checking & Intermediate Code Generation

To better tackle this part the previous AST is taken as input and a whole new tree is built, that is the concrete tree structure. The nodes of these new tree are well separated from the previous tree and are placed in the package `concrete_nodes`. From this point onwards there has been implemented a Visitor pattern to conveniently explore the tree multiple times and with different options. The following code is implemented in classes placed inside *`utils`* package.

A first implementation of the Visitor interface is the PrettyPrintingVisitor which just gives a nice and debugging useful view of the tree.

A second implementation is the StaticCheckingVisitor, which checks for the Distinct-Name Checking at first, and then goes on subsequently applying the typing rules as formally specified in the document *`assignment2_text.pdf`*. During the exploration it fills the Symbol Table and adds all the required type information for the use of IR Generator right after.

A third implementation is the IRGenerator. It creates another, simpler tree which is the structure that will be given as input to the back-end compiler. It holds the required information for abstracting from the initial source code, so the back-end compiler doesn't need to know about the compiled language. The visitor generates an intermediate code representation which adhere to the IR3 specification which can be found in the file *`assignment2_text.pdf`*.

Methods overloading has been implemented, allowing more methods with the same name to be declared on the same class, as long as the signature differ. Informally, the method signature is composed of the Method name + List of Types of params. This has not great implication on the code, as it still searches for a signature match when checking a function call is correctly made, being the code very clean and generic enough. The actual implication is in the Name Checking part at the beginning of the TypeCheckingVisitor class, where instead of just making sure method names are different, we check their signature, so is just a matter of changing the implementation of the equals method of the MethodDecl class. Note that overloading is not further supported by the backend compiler.

Null type specification has not been formalised in details. I just treat it as a normal Type, but not assignable to a var or to anything else, it's not of great use really.

While statement ambiguity has been patched by throwing a type error in case the while has no statement in its body.

Assembly Generation

Given an IR3 tree this part takes care of producing the ARM assembly code which can be further assembled into an executable. The set of instructions used in the assembly code is defined in the Ocaml file arm `structs.ml` in the current directory.

The backend code is located in the `/asm` folder to better separate from the frontend code. The core class is ASMGeneratorVisitor which only needs a IR3 tree which will then visit node by node and the assembly will be printed out.

A few choices have been made in the implementation:

- println(NULL) doesn't print anything
- Variable shadowing has to be taken into account, indeed the inner variable is the one which is considered, unless explicitly pointing to a class field with the special identifier *this*

The ARM Instruction is only able to load a limited range of immediate values with mov. The problem is that the value has to be encoded in the mov instruction itself. As all ARM Instructions are 32-bit wide, the original instruction-set up to ARMv5 only had a total of 8+4 bits to encode immediates. With the first 8-bit being able to load any 8-bit value int in the range of 0-255 and the 4 bit being a right rotate in steps of 2 between 0 and 30[1].

This has been taken care of and, if the immediate value is outside the range 0-255, then the value is loaded from memory with ldr: *ldr r<number>, =#val* The assembler then will create a constant-pool and load the value from there via pc-relative addressing.

[1] Source is a stackoverflow.com topic