DD2488 - COMPILER CONSTRUCTION
PROJECT REPORT

SAMUEL WEJEUS AND EMIL ARFVIDSSON

1. Introduction

A compiler is a complicated piece of machinery, a computer program, designed to transform high level source code written in some programming language into some lower level language understandable, and runnable on various target architectures i.e. different types of processors.

This report presents the construction, and inner workings, of a compiler for the high level programming language *MiniJava* (which is a subset of the Java programming language) created as a project int the course *DD2488 - Compiler Construction*[1]. We named the project **Cortado** which is a small and strong type of italian coffee. [1]

The report will present the various tools used to facilitate the construction of the compiler and explain the how the different stages of compilation have been implemented. The intended reader is a person who has rudimentary understanding of the workings of a compiler and our goal is to give the reader insight and a basic understanding of the inner workings and design choices we made for our implementation of MiniJava.

2. Overview

During the course of compilation a compiler goes through several stages where the input (i.e. the file of MiniJava source code) is tested and/or transformed into various representations (most common: a tree structure in various forms). This is to make the compiler more modular and easier to maintain, an input at some stage is usually the output from the previous stage. On the whole, the entire process of transforming input source code to an executable file is usually separated into the following stages:

- Lexicographical analysis
- Parsing
- Type and scope checking
- Translation to intermediate code
- Liveliness analysis
- Instruction selection
- Register allocation
- Assembly and linking (not covered)

[1]Wordplay is another of our amusements

As a reference and as an insight into our project each step will be explained briefly and for focus is towards how we covered each stage in our implementation. The last step of compilation *Assembly and linking* was not part of this project and have been excluded from the compiler. To complete the process of creating a real exactable binary we here rely on ordinary assemblers such as *nasm*[2] to preform the last step. We note that upon successful compilation using our compiler a complete, syntactically correct, assembler source file will be given as output.

The project of creating a compiler for the MiniJava programming language (hereby just referred to as 'MiniJava') was implemented using the ordinary Java programming language.

2.1. **System Requirements.** For running a precompiled binary of our complier the only requirement is a runtime environment for Java (JRE). Requirements for building the source code the following additional tools and libraries are needed. The MiniJava compiler have been successfully built with the versions of relevant libraries as noted below but more recent versions should work to.

- Java-cup-11a
- JFlex 1.4.3
- JUnit 4.10 (only needed for test cases)

To build the project simply extract the project package and type **make** in the root folder, this creates the compiler as a executable named **cortado**. To compile your own MiniJava source code files type **java ./cortado your-source-file**. This will preform all stages of compiling and output a .s files of assembler code.

2.2. **Architecture and MiniJava Extensions.** The target architecture i.e. the platform for with assembler instructions will be generated is currently AMD64. We have limited ourselves to one architecture at the moment but have plans to extend the choice of target platform to include ARM in the near future. Since MiniJava is a limited subset of Java many operators and functionality have been exclude such as for loop and much of the boolean logic. As a bonus in the DD2488 course we were encouraged to implements some of this missing functionality as extensions, none of these have been implemented and our goal is to instead focusing on an other type of bonus, namely, the mentioned future implementation of the ARM backend.

3. Lexing and Parsing

The first phase of the compiler consists of transforming the input source into tokens and parsing these against the grammar for MiniJava to discover possible syntax errors. At this stage we utilize two different tools: JFlex and JavaCUP.

JFLex is a scanner generator that scans a textfile and transforms input, matched against patterns written using regular expressions, to tokens (also called *lexing*). This is the first point of error checking, if any invalid MiniJava characters or strings are encountered the compilation is halted.

A natural companion to JFlex is JavaCUP (Constructor of Useful Parsers) which is a parser generator that creates LALR parsers[1].

Both JFlex and JavaCUP works as follows: each tool take an input file consisting of instructions, specified according to the syntax of the tool, of how the to perform its actions. Running the tool in question on the corresponding file generates a new Java *source code file* that is later included and instantiated as an object in the framework for the compiler.

Even though learning how to use both JFlex and JavaCUP can be a bit tricky in the beginning the benefit gained is *much* easier maintenance and development since the alternative, building the automatons[3] yourself, is considered way more advanced.

The output form the lexing and parsing phase is a tree created and linked together by various subclasses in the *syntaxtree* package. These corespond to a top-level object called *Program* consisting of *Classes* which in turn consists of *Methods* and so on corresponding to the grammar given for MiniJava[**?**].

Input specifications for lexing and parsing resides in the *res/* folder of the project and output is generated in *gen/*. Classes used to create the syntax tree is located in the *syntaxtree* package.

4. Scope and Type Checking

Topics: scope and type, symbol tables

5. Translation to Intermediate Code

Topics: frames, IR tree, canonicalization

6. Instruction Selection

Topics: maximal munch, architecture specifics of frames

7. Register Allocation

Topics: liveness analysis, graph coloring

8. Conclusions

References

[1] Andrew W. Appel, *Modern Compiler Implementation in Java, 2nd edition.*
[2] Nasm OpenSource Project, *The Netwide Assembler.* `http://www.nasm.us/`
[3] *Automata Theory.* `http://en.wikipedia.org/wiki/Automata_theory`
[4] *MiniJava Grammar.* `http://www.csc.kth.se/utbildning/kth/kurser/DD2488/komp12/project/newgrammar.pdf`