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. This report is also intended as documentation for ourselves on things we found difficult and their solution.

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
- Code generation
- Liveness
- Register allocation

[1]Wordplay is another of our amusements besides building compilers

- Assembly and linking (not covered)

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[4].

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

All stages up to, and partly including, the stage of intermediate code generation is commonly referred to as the *frontend*, the stages following current stages is referred to as the *backend*, intermediate code is what separates the two. Intermediate code (IR) is a sort of pseudo assembler and used to make the translation several different architectures easier.

What we do during this stage is to transform the incoming program being compiled (now in the form of a tree where scope and type errors should be non existent) and transform it into a *linear structure*. This new structure should mimic that of a real processor which means that all abstract constructions, such as: `int x = 3+4;`, have to be converted into a series of instructions of *how* a processor actually would carry out the, in this case, addition.

To implement this procedure we have utilize an additional structure called a *Translator*. The reason for this, and what the translator does, is due to that some constructs, such as conditional statements, need to be translated using several `jump` instructions. It is possible at later stages of the compilation to detect that some of these `jump`'s is unnecessary or could be rearranged in a more efficient order depending on if the statement need to return a value of not. Using a translator we can instead create a 'meta' statement and at a later stage when the context for this statement is know the translator can easily convert the statement in a efficient way by optimizing it to a, for instance, statement that returns a value. The translator is located in the `se.cortado.ir.translate` package.

6. Code generation

7. Liveness and Register Allocation

Through out the compilation process so far virtual temporaries have been used to hold variable. Since on a real CPU all calculations have to be performed on variables located either in memory, or preferably, in registers we need to link these temporaries to real registers, the problem is that the number of registers is limited and we need to determine at any point in time which variables is currently needed and need to be placed in registers.

This problem is solved using *flow control*. When flow control starts the input for this stages is a linear list of `Assem` instructions. The flow analysis is done in two parts: First, the control flow of the `Assem` program is analyzed by traversing the statements backwards, producing a control-flow graph. The control-flow graph is simply a graph representation of *"how"* the `Assem` program executes and consists of nodes corresponding to each statement (or instruction) and directed edges between them modeling the flow of the program.

Second part is what is called *liveness analysis*. Liveness analysis could be used for various kinds of optimizations but here our goal is to focus on what set of variable is live at the same time. What liveness now does is to traverse the control-graph determining at each node the set of variables that needs to be *live* at that point so that future points that potentially use those variables can be guaranteed correct data. The mathematical definition for liveness at a node is listen below:

$$\begin{aligned} in[n] &= use[n] \cup (out[n] - def[n]) \\ out[n] &= \bigcup_{s \in succ[n]} in[s] \end{aligned}$$

Here $in[n]/out[n]$ is edges, $use[n]$ = a right hand side *usage*, $def[n]$ is a variable definition and $succ[n]$ is the successor node of n. The equation given above is executed for each node and the algorithm terminates when, for a node, both its in and out edges satisfy the equation.

The result of this procedure is an *interference graph* which is a graph consisting of the variables discovered in the control-flow graph and two nodes (i.e. variables) are connected if, and only if, they are *live* at the same time. The reader should note that this graph could, and most likely will, be disconnected.

7.1. **Graph coloring.** During the finalization of the interference graph it is still assumed that the number of registers is unlimited, what we done so far is simply to determine the minimal subset of variables than *must* be live at each point in time. For this we use the *graph coloring* procedure evaluated with K colors, where K is the number of registers available. If we allocate the registers according to the graph coloring scheme we will be guaranteed that variables that are live at the same time never will interfere with each other by, possibly, overwrite each others values.

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`