Decaf: A Compiler for a Subset of the Java Language

Isaac Davis - CSCI 1260, Fall 2017

1 Abstract

The Decaf programming language is a subset of the Java programming language designed for implementation by students in CSCI 1260, "Compilers and Program Analysis." The language includes major features of the Java programming language, including arithmetic, boolean logic, looping and branching, and objects with inheritance and encapsulation. Students are given great latitude in choosing both the implementation and target languages for the compiler - my compiler is written in OCaml and outputs 32-bit Intel x86 assembly code for Linux.

2 Documentation and Source Code

Information about Decaf can be found here, and the source code for my compiler, along with instructions for building and running, can be found here.

3 Implementation

I chose to use OCaml primarily for its support of pattern-matching over user-defined variant data types, which facilitates complex tree traversals of the type commonly encountered in compiler design. OCaml also lends itself well to compiler implementation because of its support for mature compiler-writing tools (`ocamllex` and `ocamlyacc`) and because of its strong static type system, which eases the development of a large codebase.

The compiler accepts as input a single Decaf source file. First, the source code is scanned into tokens using a grammar defined with `ocamllex`. The token stream is then parsed into an abstract syntax tree (AST) using a grammar defined with `ocamlyacc`. The AST is the compiler's first fully-formed representation of the input program. The compiler next type-checks the AST and, if there are no errors, walks the AST and generates from it a tree of intermediate code (icode), which more closely resembles the assembly-language instructions that will eventually be output. Finally, the compiler walks the icode tree and outputs the corresponding x86 assembly instructions to a text file. Assembling the compiler output into an executable binary is beyond the scope of this project - the compiler thus invokes `gcc` on the assembly file, which links in a runtime library for I/O (written in C) and generates an executable binary in the ELF format.

4 Continued Work

There are many avenues for continued work on the compiler, although the course has ended. Code generation is not yet fully implemented - the main missing functionality includes looping and conditionals. Additionally, inheritence and array/field access are partially functional but incomplete. There is also substantial room for optimization. For example, the compiler's current register allocation algorithm is entirely naive - every local variable is “spilled,” which means it is allocated exclusive space in its enclosing function’s stack frame for the entire time the variable is in scope. Various intermediate optimizations, such as constant folding, duplicate code elimination, dead code elimination, and peephole optimization, could also be applied over the AST and icode.