6.035 Team **Antidaephobiax**: Project 3 Writeup

Abel Tadesse, Devin Morgan, Madhav Datt

October 29, 2016

1 Overview

Our objective for this project was to compile Decaf scripts into unoptimized Assembly Language code for the x86 Architecture. To accomplish this, we wrote Decaf scanner rules and Decaf parser rules for the Antlr4 parser generator. We then used the Decaf Listener class Antrl4 generated to implement the traditional visitor pattern allowing us walk through the AST Antr4 built so that we could assemble our IR.

---

2 The Intermediate Representation

We designed our IR with 3 objectives in mind. First, we wanted the IR to intuitively organize the specific program it represented. Next, we wanted the IR to perform the entire semantic analysis of the program (once the IR was built) so that the Decaf Listner would be strictly responsible for building the IR. Finally, we wanted the IR to be able to directly produce executable (though clearly inefficient) assembly code.

To address our goal of organization, we constructed created a Ir class hierarchy that used a combination of abstract parent classes and concrete child classes. We also used a naming convention that clearly illustrated which part of the Decaf language, the class represented. By building our IR this way, we were able to achieve our goals of semantic analysis and code generation by implementing the functions semanticCheck() and codeGeneration() in each concrete IR class and with the help of some external data structure classes. We are using a 3-operand IR because of its greater intuitiveness, and are then going to a 2-operand implementation for the asm.

---

3 The Code Generation

Our overall code generation strategy is as follows: We have a global AssemblyBuilder object which is empty initially and holds the different parts of the assembly code while it is being built. There is a stackFrame object which is associated to the block corresponding to each method and keeps track of local objects/objects and their corresponding storage locations in terms of registers or positions in the stack corresponding to the function.

Most classes that inherits from Ir give an implementation of a method with signature as follows, that modifies that produce assembly code for the Ir component they correspond to.

```
generateCode(AssemblyBuilder assembly, Register register,
             StackFrame stackFrame);
```

Some of these classes mutate the assemblyBuilder, meaning they write stuff on the global assembly code, while others are given a local assemblyBuilder and write to that.

For example, IrMethodDecl will need the number localVariables to construct a header. And then it has to generate code to the method body (IrCodeBlock) and provide a footer, with return and leave instructions. Therefore it passes a new AsssemblyBuilder and StackFrame to the block Object and the StackBlock generates the local code using them and writes to the passed assemblyBuilder. The methodDecl then uses that data to figure out how many local vars it needs and finish up generating the code.

We have done the code generation process by iterating over the intermediate representation of the program, however, we have not built a control flow graph.