6.035 Project Documentation

Eileen Chau, Kelly Lam, Erastus Murungi, Gianna Torpey

1. Design

Our compiler is composed of five different parts. The first part of our compiler is the Scanner and Parser, which scans the input code into tokens and parses the tokens into an abstract syntax tree (AST). The second part is the Semantic Checker which performs Decaf semantic checks by traversing the AST and creating an intermediate representation (IR). The third part is Code Generation where the IR will be used to create a control flow graph (CFG) and three address codes (TAC) to help generate assembly code. Lastly, the final two parts of our compiler: the Data Flow Analysis and Optimizer focuses on optimizing code generation. These five steps together will make up our Decaf compiler.

1.1 Scanner and Parser

Phase 1: No documentation required

1.2 Semantic Checker

Our team chose to perform the semantic checks by implementing two visitors that visit our AST, named `IRVisitor` and `TypeCheckVisitor`, respectively. The first visitor builds the symbol tables associated with each distinct scope of the program and effectively transforms the AST into an IR tree, and the second visitor returns the type of the root of each subtree in the AST. We decided to implement two visitors because we divided the semantic checks into 2 categories: checks that concern a variable's scope (e.g. identifiers within the same scope must be unique, identifiers must be declared before used, etc.) and checks that impose restrictions on the type of expressions and operands (e.g. the expression of a return statement must match the type of the method, the operand of conditionals must have type boolean, etc.).

The `IRVisitor` performs the scope checks as it builds the symbol tables by simply searching for an identifier either within itself or in one of its parents. We decided to maintain 3 symbol tables, for the global fields, imports, and methods. The symbol tables map an identifier to a specific type of descriptor; for example, `GlobalDescriptor, MethodDescriptor, ParameterDescriptor`, etc. We chose to implement these as separate objects because each descriptor stores slightly different information. Symbol tables also have parent pointers to "travel up" the scopes to determine whether an identifier has been declared.

The `TypeCheckVisitor` handles all remaining checks that the `IRVisitor` doesn't handle. This visitor effectively computes the type of each node in the AST by recursively checking its children. The type of each node is then checked against the expected type defined by the semantic checks.

We also chose to create `DecafException` objects that take in a token position and an error message to keep track of the errors that we discover in the program. We stored these objects in an array so all our errors are stored in one place for easy accessibility at the end of the execution.

1.3 Code Generation

We chose to do code generation in several small steps rather than going straight from our IR to assembly code because this allows for greater modularity and less room for potential bugs. Our first step was to create an initial CFG (iCFG) that short circuits from our IR. Then, we maximized each CFG block and removed the NOPs from the CFG. Next, we use our CFG to create a TAC, which flatten expressions and bring it one step closer to assembly. Lastly, we can generate assembly code by using our TAC.

Our CFGs consist of `CFGBlock`s that store a list of `CFGLine`s and have pointers to its parent(s) and child(ren). There are three types of `CFGLine`s: `CFGAssignment`, `CFGDeclarations`, `CFGExpression`. There are also three types of `CFGBlock`s: `CFGConditional`, `CFGNonConditional`, and `NOP`. A `CFGConditional` ends with a conditional `CFGExpression` and therefore has two children: one for the true block and one for the false block. A `CFGNonConditional` contains lines and does not end with a condition `CFGExpression`, so it only has one child. Lastly, a `NOP` has no `CFGLine`s. This structure allows for modularity and clarity in our CFG design.

We create the iCFG by using a series of destructs and short circuits on our IR. Short circuiting conditional expressions helps in optimizing our assembly code. Short circuiting is done in `ShortCircuitProcessor`, which tries to simplify expressions by using De Morgan's Law before applying the short circuiting methods as described in class.

After implementing short circuiting in our CFGs, we decided to convert our CFG to contain `ThreeAddressCode`s (`TAC`s) instead of our initial `CFGLine`s by visiting each AST node that is stored in a `CFGLine` and producing the (potentially multiple) `TAC` lines that correspond with that node. We also flatten expressions in this step by recursively stepping down the expression subtree, creating temporary variables for every intermediate operation, and consolidating them to generate a list of three-address instructions for an expression.

The final step in our design is to convert the `TAC`s to x64 assembly code. We do this by implementing another visitor that visits all our TAC lines and generating the appropriate assembly code. We also implement the method prologue and epilogue for each method in this step.

1.4 Data Flow Analysis

Our team implemented most of the data flow optimizations. For the global CSE and copy propagation, we implemented the respective available expressions, and reaching definitions as discussed in lecture. We computed the `IN` and `OUT` bit vectors (represented by `HashMaps`) and worklist algorithms for each basic block according to the transfer function for each analysis.

In addition to these two primary dataflow optimizations, we also implemented dead code elimination and dead store elimination, both of which remove instructions that define variables that are not used from that point. In dead store elimination, we use the worklist algorithm and liveness analysis to specifically remove assignments to variables that are not used further on. Dead code elimination removes unnecessary pushes, and pops, due to inlining of functions. Overall, we reduce the number of assembly instructions such as expensive stores, improving performance.

Furthermore, we implemented constant propagation using the worklist algorithm and reaching definitions. In implementing this optimization, we also added the functionality of detecting constant reaching definitions within the same block to add more avenues of replacing operands in assignments with constants. As a result, when possible, variable reads will be replaced with a constant. This improves performance as we eliminate the need to read from the stack or registers and simply use constants.

We also found it useful to run the algebraic simplification optimization after constant propagation to further reduce expressions that now only consist of constants after constant propagation. This optimization reduces the number of instructions we need, as we precompute the constant result of some expressions, thereby improving performance in terms of how long it takes to execute the assembly code.

We implement unreachable code elimination; in the case where the condition for an if-statement is constant, we eliminate the unreachable branch and the condition, and convert the conditional branch to an unconditional branch. In doing so, we have fewer instructions. We also eliminate the need to evaluate a condition and the need to perform a jump necessary for a conditional branch, thereby improving performance of the assembly code generated.

Before and after each of these optimizations can be found in `/doc/dataflowOptimizationEvidence`.

1.5 Optimizations

Our team decided to implement the linear scan register allocation algorithm because it has a smaller computation overhead compared to the graph coloring algorithm. Although the performance of the graph coloring algorithm is better than the linear scan, both algorithms are correct and improve compilation time.

The linear scan algorithm works as follows. First, a linear pass is made through the input code to determine the live intervals of every variable. Next, variables are assigned to available registers sequentially starting at the first line of the input code. When a variable is dead, its register becomes available again. If there are no more available registers, then we split a variable by storing it on the stack instead of in a register. The variable with the highest number endpoint is the one that is split. This algorithm ensures that we will never use unavailable registers at any point in time. A before and after of this optimization is in `/doc/registerAllocationEvidence`.

2. Extras

Some extra features in our design to help us debug include a function that prints an AST tree to console and printing CFGs to PDF using graphviz.

3. Status/List of Current Bugs

- Possible bug in semantic checker about shadowing parameters
- Division in assembly
- Array accesses are wrong
- Array bounds are not correctly implemented

4. Contribution

We tried to divide the work evenly among team members but division is difficult at times because of our different work schedules and how all the code is inter-connected. This usually resulted in a slow start to our project, and consequently, intense cramming and debugging sessions at the end. Aside from the communication challenges, everyone in our team is willing to work so our work sessions are very productive. We plan to resolve the communication challenges by taking a more proactive approach at the beginning of each phase and going to office hours together as a team.