1 Introduction

In the previous assignment you extended the Javali compiler by adding the Control Flow Graph (CFG). The skeleton in your subversion repository already includes CFG construction, so you can base your solution to this assignment on the skeleton if you did not handle CFGs in the previous assignment. (Or you could merge your solution to Assignment 0 into this skeleton.)

The objective of this assignment is to develop a compiler that uses Static Single Assignment (SSA) form as intermediate representation (IR). Future assignments will ask you to enhance this compiler and to include optimizations.

There are four basic steps that your compiler must include (to be done in this assignment):

1. Compute the dominator tree (DT).
2. Introduce Φ-operations into the control flow graph.
3. Rename the variables to ensure that the correct version of a variable is used.
4. Replace Φ-operations with equivalent assignments.

2 Step 1: Control Flow Graph and Dominator Tree

The construction of CFG and DT is placed into a separate phase of the compiler. This phase executes after the semantic analysis. Here is the recommended sequence of steps that this phase of the compiler is supposed to implement:

- In the class cd.cfg.Dominator, compute the immediate dominator of each basic block $N$ that appears in the CFG.

The class cd.ir.ControlFlowGraph is used to represent CFGs. One ControlFlowGraph instance is associated with each cd.ir.Ast.MethodDecl instance (a new field, cfg, has been added to MethodDecl for this purpose). The class cd.ir.BasicBlock is used to represent basic blocks. The immediate dominator of a basic block should be stored in the dominatorTreeParent field of the class BasicBlock.
• In the class `cd.cfg.Dominator`, construct the dominator tree (one tree for each method). The children of a `BasicBlock` in the dominator tree should be stored in the field `dominatorTreeChildren`.

• In the class `cd.cfg.Dominator`, compute the **dominance frontier** for each block in each method. The dominance frontier of a `BasicBlock` should be stored in the field `dominanceFrontier`.

See the dragon book [Dragon] or [Cooper et. al.] for algorithms to compute dominators.

### 3 Step 2 and 3: SSA Form

In `cd.cfg.SSA`, the following steps are to be done for each method:

• For each simple source-language variable (i.e., variables defined in the current method), determine the set of nodes where a \( \Phi \)-operation must be inserted. Start with the set of nodes \( N_i \) that set the variable and then insert a \( \Phi \)-operation at the nodes that are a member of \( \text{DF}(N_i) \). Iterate until no more \( \Phi \)-operations must be inserted. See [Appel], [Cytron et. al.], [Muchnick] for more details.

• Introduce version numbers for each local variable. You should create a new `VariableSymbol` for each distinct version, and add that symbol to the `locals` field of the corresponding `MethodSymbol`. To keep track of the various versions, you have to use a stack, as explained in class and the literature.

These two steps convert the IR into SSA form (and we can show that the SSA form is minimal, i.e., there are no unnecessary \( \Phi \)-operations). In later assignments, you will perform optimizations on the SSA form.

### 4 Step 4: De-SSA

We have provided a code generator (`CfgCodeGenerator`) which works directly from the control-flow graph. However, it cannot handle SSA form. Therefore, you must "de-ssa" the program by editing `cd.cfg.DeSSA`. We suggest you do this by converting each \( \Phi \)-statement into copy statements, as discussed in class.

• Prior to code generation, each \( \Phi \)-operation must be translated into a sequence of assignments. These assignments are placed at the end of the predecessor blocks. See [Appel].

  It is easy to insert these assignments if no node with multiple successors has an edge to a node with multiple predecessors. We recommend that you ensure that this property holds while constructing the CFG, if necessary by inserting a block with an empty assignment (a NOP).

Please test your compiler to demonstrate that it translates programs correctly. You do not have to worry about copy operations that you may consider unnecessary. The objective is to get a working, correct compiler – not to produce the fastest code possible. Future optimizations will take care of many of these copy operations.
References


