Rationale

List of Optimizations we considered (ones we implemented in **bold**):

- **Constant Propagation** - We wanted to reuse the machinery from Common Subexpression Elimination, and make sure we understood the worklist algorithm.
- Copy Propagation - Similar to Constant Propagation. Too similar. We decided it would be a more interesting venture to work on other optimizations.
- **Dead Code Elimination** - Reuse machinery from CSE. Also would help to clean up our code.
- Algebraic Simplification - Decided against it, because there were more interesting optimizations.
- **Register Allocation** - We thought this would give us significant performance gains, while also being interesting to implement.
- Strength reduction for derived induction variables
- Elimination of superfluous induction variables
- List scheduling Algorithm
- Loop unrolling
- Parallelization through the Distance Vector Method
- Parallelization through Integer Programming

If we did not give a reason, then it was because of deadlines. We elected to pursue Register Allocation over List Scheduling or any of the loop optimizations. Initially, we pursued all three, but as the deadline approached, we had to focus more on what we thought we could get done.

Work done

Refactoring

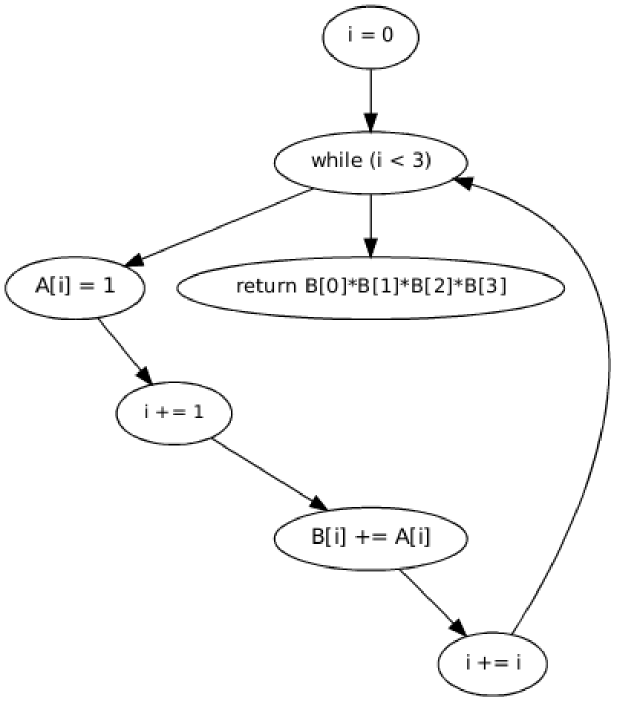
We relied on DataFlowGraphs and ControlFlowGraph for intermediary representations between the parse tree and the assembly code. We realized that the graph functionality for the two representations were exactly the same, so we refactored it into the same class, only that now it takes in generic types. We also refactored the our worklist algorithm for available expressions to be a general worklist algorithm, abstracting away parts unique to each of the three: Available Expressions, Liveness, and Reaching Definitions. These two refactorings have made it significantly easier to implement constant propagation and dead code elimination compared to how long it took us to get available expressions.

Constant Propagation

Implemented constant propagation according to the algorithm shown in class.

Currently, we do not do further analysis on method calls to figure out which globals each function reads or writes, or what output values a function may have. We must therefore assume that method calls cannot be replaced by constants.

We also do not do further analysis on arrays and leave arrays as they are in the program originally. This is to prevent us from messing up cases like one on the right. A[i] should not be replaced by 1.



Dead Code Elimination
Implemented dead code elimination as shown in class.

We do not do further analysis on functions and array indices, so as to ensure that we preserve the semantic meaning of the program, we do not eliminate arrays or method calls.

Register Allocation
Our register allocation code works as follows:

1. Compute reaching definitions.
2. Compute def-use chains.
3. Compute variable use webs.
4. Compute convex sets covered by webs.
5. Construct conflict graph of webs' convex sets.
6. Compute a graph coloring of the conflict graph. (If the graph is uncolorable, leave some nodes uncolored. Uncolored nodes are not stored in registers-- they are loaded from memory each time they are used.)

We sacrificed some functionality in favor of efficiency. Some sacrifices were:

- We do not use any heuristic to determine how to best convert a colorable graph into an uncolorable graph.
- We only do register allocation over the callee-saved registers. Thus, we never do register allocation with the parameter registers, nor with the return registers.

Below, we include a few examples of the register allocator in action.

File: **examples/regalloc/few-vars.dcf.**

Relevant Part of Decaf Source:

```
v = 1; w = 2; x = 3; y = 4; z = 5; return use_vars(v, w, x, y, z);
```

Allocations (using `./run.sh -t regdebug`):

(v, RBX), (w, R15), (x, R14), (y, R13), (z, R12)

Demonstrates: We can allocate up to five registers simultaneously.

***File:* examples/regalloc/many-vars.dcf.**
Relevant Part of Decaf Source:

```
u = 1; v = 2; w = 3; x = 4; y = 5; z = 6;
return use_vars(u, v, w, x, y, z);
```

Allocations (using `./run.sh -t regdebug`):
(u, RBX), (w, R15), (z, R14) (y, R13), (v, R12).
Demonstrates: We allocate as many variables as we can, but do not allocate *x*. The variable x is represented as `-32(%rbp)` in the generated code.

***File:* examples/regalloc/nonconflict.dcf.**
Relevant Part of Decaf Source:

```
u = 1; v = u; w = v; x = w; y = x; z = y; return z;
```

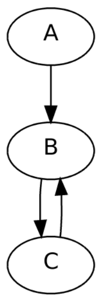
Allocations (using `./run.sh -t regdebug`):

```
(z, R12), (y, R13), (v, R14), (u, R12), (w, R13), (x, R12)
```

Demonstrates: We reuse registers when variables' webs do not conflict.

Optimizations that didn't make it (loop optimizations)

We have code that can find loop headers and back edges, but we don't do anything with it. Along the way, we found that the algorithm presented in class doesn't work, because it doesn't handle the case to the right:



Due to the back edge, the algorithm will never realize that B and C are dominated by A. To fix this, we had to make the worklist algorithm never use back edges when computing the new IN sets of each node. Whenever it finds a new back edge, it also puts the loop header back onto the changed list.

This algorithm terminates, because the new case where nodes are added to the changed list only happens when a new back edge is found. The number of edges in a graph is O(# of nodes^2), which is a ceiling on the number of back edges.

The algorithm produces the correct dominator tree, because it makes sure that loop headers do not consider nodes that they dominate when computing what dominates them.