Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2018 Handout – Dataflow Optimizations Assignment Friday, Oct 26

DUE: Friday, Nov 9, 11:59 pm

For this part of the project, you will add dataflow optimizations to your compiler. **At the very least, you must implement one of the first three global dataflow optimizations below.** *All other optimizations listed below are optional.* You may also wait until the next project to implement them if you are going to; there is no requirement to implement other dataflow optimizations in this project. We list them here as suggestions since past winners of the compiler derby typically implement each of these optimizations in some form. You are free to implement any other optimizations you wish. Note that you will be implementing register allocation and other assembly level optimizations for the next project, so you don't need to concern yourself with it now.

Global CSE (Common Subexpression Elimination): Identification and elimination of redundant expressions using the algorithm described in lecture (based on available-expression analysis). See §8.3 and §13.1 of the Whale book, §10.6 and §10.7 in the Dragon book, and §17.2 in the Tiger book.

Global Copy Propagation: Given a "copy" assignment like `x = y` , replace uses of `x` by `y` when legal (the use must be reached by only this def, and there must be no modification of `y` on any path from the def to the use). See §12.5 of the Whale book and §10.7 of the Dragon book.

Dead Code Elimination: Dead code elimination is the detection and elimination of code which computes values that are not subsequently used. This is especially useful as a post-pass to constant propagation, CSE, and copy propagation. See the §18.10 of the Whale book and §10.2 of the Dragon book.

The following are other useful dataflow optimizations that are useful to implement, but not required for this stage of the project.

Global Constant Propagation and Folding: Compile-time interpretation of expressions whose operands are compile time constants. See the algorithm described in §12.1 of the Whale book.

Loop-invariant Code Motion (code hoisting): Moving invariant code from within a loop to a block prior to that loop. See §13.2 of the Whale book and §10.7 of the Dragon book.

Unreachable Code Elimination: Unreachable code elimination is the conversion of constant conditional branches to equivalent unconditional branches, followed by elimination of unreachable code. See §18.1 of the Whale book and §9.9 of the Dragon book.

The following are not generally considered dataflow optimizations. However, they can increase the effectiveness for the transformations listed above.

Algebraic Simplification and Reassociation: Basic algebraic simplifications described in class. This includes simplifying the following rules:

$$a + 0 \Rightarrow a$$

$$a - 0 \Rightarrow a$$

$$a * 1 \Rightarrow a$$

$$b == true \Rightarrow b$$

$$b \; != false \Rightarrow b$$

etc

You may find it useful to canonicalize expression orders, especially if you choose to implement CSE. See §12.3 of the Whale book and §10.3 of the Dragon book.

Strength reductions Algebraic manipulation of expressions to use less expensive operations. This includes the following transformations that convert multiplying constants into bit shifts i.e.:

$$a * 4 \Rightarrow a << 2$$

and include turning multiplication operations from within a loop into sum operations.

Inline Expansion of Calls: Replacement of a call to procedure `P` by inline code derived from the body of `p`. This can also include the conversion of tail-recursive calls to iterative loops. See §15.1 and §15.2 of the Whale book.

You will want to think carefully about the order in which optimizations are performed. You may want to perform some optimizations more than once.

All optimizations (except inline expansion of calls) should be done at the level of a single method. Be careful that your optimizations do not introduce bugs in the generated code or break any previously-implemented phase of your compiler. Needless to say, it would be foolish not to do regression testing using your existing test cases. *Do not underestimate the difficulty of debugging this part of the project.*

As in the code generation project, your generated code must include instructions to perform the runtime checks listed in the language specification. It is desirable to optimize these checks whenever possible (*e.g.,* CSE can be used to eliminate array bounds tests). Note that the optimized program must report a runtime error for exactly those program inputs for which the corresponding unoptimized program would report a runtime error (and the runtime error message must be the same in both cases). However, we allow the optimized program to report a runtime error earlier than it might have been reported in the unoptimized program.

What to Hand In

Follow the directions given in project overview handout when writing up your project. You may simply add onto your design doc from project 3, with any additions being in their own paragraph/section. Make sure to include a description of every optimization you implement. For each optimization, please include one example of your IR before and after the optimization (printed on

.txt files from your compiler is fine, included in the doc/ directory) so that we could gain a better understanding of what your optimization is doing.

Your compiler's command line interface must provide the following interface for turning on each optimization individually. Something similar should be provided for every optimization you implement. Document the names given to each optimization not specified here.

- `--opt=cse` — turns on common subexpression elimination only
- `--opt=cp` — turns on copy propagation optimization only
- `--opt=cp,cse` — turns on copy propagation optimization and common subexpression elimination only
- `--opt=all` — turns on all optimizations

This is the interface provided by CLI.opts / optnames facility in the Java and Scala skeletons. See the source for more details. A similar mechanism is provided in the Haskell skeleton. For the full command-line specification, see the project overview handout.

You should be able to run your compiler from the command line with:

```
./run.sh --target=assembly <filename>          # no optimizations
./run.sh --target=assembly --opt=all <filename> # all optimizations
```

Your compiler should then write a x86-64 assembly listing to standard output, or to the file specified by the -o argument.

A reminder that this part of the project won't be graded until you submit your final compiler, but if your team would like feedback, please submit by the deadline listed at the top of this file.