CS 380C Compilers — Spring 2014	Programming Assignment #4
Analysis Based Optimizations	Due March 5, 2014

Introduction

In this assignment, you will use the iterative dataflow analysis framework that you built in the previous assignment to eliminate redundant computations, and you will measure the impact of your code transformations on program performance. Specifically, you will implement Dead Code Elimination (*DCE*). You will then use your DCE pass to eliminate the redundant computations in unoptimized LLVM code. As before, you should operate on LLVM code that has been generated with `-O` and processed with `-mem2reg`. You only need concern yourself with LLVM registers, so you need to handle ϕ-nodes carefully in all your analysis, but you do not need to worry about loads and stores.

In an interesting twist, your DCE pass will use a more sophisticated Liveness analysis than the analysis that we discussed in class. This variant of Liveness—which we will call *Faint Variable Analysis*—will be described in more detail below.

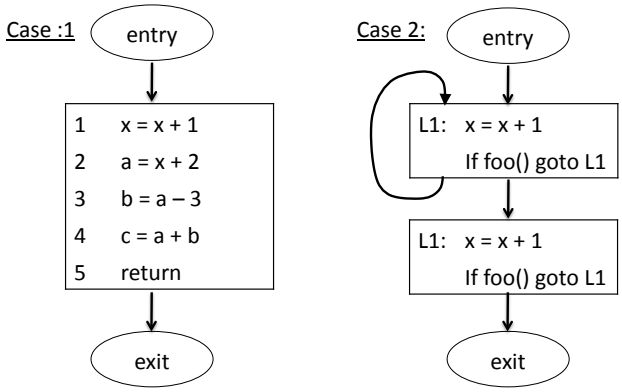
Finally, you will measure the impact of your optimizations on the execution times of programs. Further details that are relevant for completing this assignment are provided below.

Logistics

Any clarifications and revisions to the assignment will be posted on Piazza.

1 Dead Code Elimination

The idea behind Dead Code Elimination (*DCE*) is that an assignment of the form "`x = t`" can be eliminated if its LHS variable `x` is not live at the program point P immediately following the assignment. One of the limitations of DCE is that it cannot directly eliminate the assignment "`x = x + 1`" in the two examples shown below:



In the first case, `x` is not dead after instruction 1 (`x = x + 1` assignment because `x` is used in instruction 2. Instruction 2 is also not dead because its LHS variable (`a`) is used in instructions 3 and 4. However, instruction 4 is in fact dead. If we applied DCE repeatedly to this code, we could eventually eliminate instruction 1. However, it would be more desirable to eliminate it in a single data-flow pass.

In the second case, the LHS of "`x = x + 1`" is not dead because it is used by its own RHS due to the cycle in the flow graph. However, since the ultimate value of `x` is never used, this instruction could in fact be safely eliminated from the loop body.

We say that the LHS variable `x` in an assignment "`x = t`" is *faint* if along every path following the assignment, `x` is either dead or is only used by an instruction whose LHS variable is also faint. Your mission in this assignment is to

write a new data flow analysis called *Faint Variable Analysis* (FVA) that will directly determine that the LHS variable of "`x = x + 1`" is "*faint*" in both of these cases.

Some issues to consider when designing your FVA algorithm:

1. What is the direction of the data flow analysis ?
2. What is the meet operator ?
3. What are the lattice elements ?
4. What are the values of top and bottom ?
5. How do you initialize the iterative algorithm ?
6. What are the transfer functions (hint: these must be done at the instruction level, not the basic block level).

Now, write a DCE pass that uses FVA to eliminate instructions that have either faint or dead LHS variables.

Make sure that your pass is called `dce-pass` and that it is the only pass required to execute all of your optimizations. Similarly, make sure that your `Makefile` builds one `.so` for the pass called `dce-pass.so`.

2 LICM: Loop Invariant Code Motion

We discussed Loop Invariant Code Motion (LICM) in class. The key steps consists of the following:

1. Compute Reaching Definitions using your dataflow framework from Assignment 3.
2. Find loop invariant computations (assume that the variable defined by this statement is called `x`).
3. Find the exits from the loop (i.e., nodes with successors outside the loop).
4. Identify candidate statements for code motion, namely, those that meet the following conditions:
   (a) The computation is loop invariant.
   (b) The candidate is in a basic block that dominates all exits of the loop.
   (c) The variable `x` is not assigned to elsewhere in the loop.
   (d) The candidate is in a block that dominates all uses of variable `x`.
5. Perform a depth-first search of the blocks. Move candidate to the preheader if all the invariant operations it depends upon have been moved.

Note: You are allowed to use LLVM loop structures to find appropriate loops (the pass name is *-loops*; corresponding classes are defined in *LoopInfo.h*) or you are free to implement loop detection yourself.

Make sure that your pass is called `licm-pass`, and that it is the only pass required to execute all of your optimizations. Similarly make sure that your `Makefile` builds one `.so` for the pass called `licm-pass.so`.

3 Performance Evaluation

You should then evaluate the impact of your optimizations on program execution time. At the minimum you should write a few synthetic benchmarks and measure how much your optimizations can improve execution time. You might optionally compare your DCE pass against the LLVM version and similarly for LICM. Include your results and a short (one paragraph) description in a text file called `benchmark.txt` in your submission.

We will also compare all submitted implementations so that you can earn class bragging rights. Our comparisons will be performed by running your LICM pass followed by your DCE pass and then building an x86-64 binary from it with no further optimization.

4 Hand In

Use the `turnin` program to submit a single `tar.gz` or `tar.bz2` file that contains your source code in a directory with a `Makefile` that will build it. The `Makefile` should build two separate `.so`s: `licm-pass.so`, and `dce-pass.so`. For this project, please name the directory `assignment4`. The turn-in command will be: `turnin --submit amp cs380c_assgn4 assignment4.tar.gz`. Make sure your code builds correctly on the provided virtual machine and does not depend on any files outside the code you submit.

The build system is not the subject of this class so feel free to help each other with it or post useful variants of the `Makefile`.

This assignment is due before class on the due date.

Acknowledgments. This assignment was originally created by Todd Mowry and then modified by Calvin Lin and Arthur Peters.