Assignment Description

In this assignment, you will implement several dataflow analyses and optimizations on the Low IR program representation. We expect you to build upon the code that you wrote for the previous assignments. You are required to implement the following:

1. A Generic Dataflow Analysis Framework.

Your implementation of the dataflow analysis framework must consist of: 1) a class that implements a generic dataflow engine; and 2) an interface for the dataflow information and for the transfer functions. For each analysis instance, you only need to implement this interface; the analysis engine is implemented once and reused for each analysis instance.

The dataflow analysis class must have a method which, given a control flow graph and a boundary dataflow information, iteratively solves the dataflow equations (using the worklist algorithm) and computes dataflow information at each program point in the flow graph. You should allow your dataflow analysis class to be instantiated either a a forward or as a backward analysis.

The dataflow information interface must have methods that provide at least the following description of the lattice: a method that yields the top element in the lattice; a method that performs the meet operation between two pieces of dataflow information; and a method which tests if two dataflow information objects correspond to the same element in the lattice. The interface also contains a method that models the transfer functions in the dataflow analysis framework. This method takes a low-level IR instruction and a dataflow information object as parameters, and returns the dataflow information after the execution of the statement.

2. Analysis Instances and Optimizations.

Use your generic dataflow analysis framework to implement the following analysis instances:

- *Constant folding analysis*: determine variables whose values are constant.
- *Live variables analysis*: compute variables which may be live at each program point;
- *Available expressions*: compute expressions available in all of the program executions;
- *Reaching definitions*: for each program point, compute the definitions that may reach that point;

Next, use the analysis results to perform the following optimizations:

- *Constant folding*: use the results from constant folding analysis and replace each constant variable with the computed constant.
- *Dead code elimination*: remove code which updates variables whose values are not used in any executions. This optimization will use the results from the live variable analysis.
- *Common subexpression elimination*: reuse expressions already computed in the program. Here you will use the information computed with the available expressions analysis. If an expression is available before an instruction that re-computes that expression, you have to replace the computation by the variable which holds the result of that expression in all executions of the program. If there is no such variable, you will create a temporary variable to store the result of the expression.
- *Loop invariant code motion*: for a given loop, detect computation which doesn't change in different iterations of the loop and hoist that computation out of the loop. Use reaching definition information to determine loop invariant code. For this transformation, you must identify loops in the CFG.
- *Elimination of run-time checks.* Perform transformations similar to common subexpression elimination and loop invariant code motion to eliminate redundant array bounds checks or null pointer checks, or move such checks outside loops.

Command line invocation. As in the previous assignment, your compiler must be invoked with a single file name as argument: `java IC.Compiler <file.ic>`. With this command, the compiler will parse the input file, perform semantic checks, generate intermediate code, and build the control flow graph. Then, it will perform various analyses and optimizations on the Low IR code. In addition to all of the options from the previous assignments, your compiler must support the following command-line options:

- Option `-cfo` for constant folding;
- Option `-dce` for dead code elimination;
- Option `-cse` for common subexpression elimination;
- Option `-lcm` for loop-invariant code motion;
- Option `-erc` for elimination of run-time checks;
- Option `-opt` to perform all of the above optimizations.

The compiler will perform the optimizations in the order they occur in the command line. The above arguments may be invoked multiple times in the same command line - in that case the compiler will execute them multiple times, in the specified order. The compiler will only perform the analyses necessary for the optimizations requested. When the `-print-lir` also occur in the command line, you must print the Low IR before or after optimizations, depending on where it occurs in the command line. For instance, with: `-cfo -print-lir -dce`, you must print the Low IR after the compiler performs constant folding, but before it removes dead code.

Output dataflow information Your compiler will also print out the computed dataflow information at selected program points. You will determine these program points using method calls of the form `showDFAinfo(msg)`. Whenever such a call occurs in the program, the compiler will print the specified message `msg`, followed by a textual representation of the dataflow information at the program point where the call occurs. You must print only the final result of the analysis (not the information at each iteration!). Make sure your loop-invariant code motion doesn't move these calls out of the loop body. You should not attempt to type-check such calls.

What to turn in

Turn in on paper:

- A brief, clear, and concise document describing the your code structure and testing strategy.
- Feedback. Please continue to tell us your overall thoughts about the assignment – how much time you spent on it and how you think it could be improved.

Turn in electronically:

- A file `pa4.tar.gz` containing all of your source code and test cases (in directories `/src` and `/test`).