

Your goal in this assignment is to implement common subexpression elimination and available expressions augmented with reaching expressions. This project will make our discussions of data-flow analysis more concrete.

The basic idea is to perform available expression analysis to identify expressions that are available at the various points in the program. At any computation of an expression that is already available, you can replace the computation with a temporary variable. You will also have to assign the value of the available expression at the last point it was computed along all incoming control-flow paths. (Thus the reaching expressions information).

1 Available and Reaching Expressions

You will perform global dataflow analysis, *i.e.*, within an entire procedure. Assume that the procedure is represented as a control flow graph with statements in a middle-level intermediate representation. In particular, each statement can have up to one assignment. Let U be defined as the set of all expressions in the procedure. Each expression also has a set of statements associated with it to indicate where it was last computed along all incoming control-flow paths. Flow values for this problem are subsets of U . Each statement S in the procedure has associated with it two sets, $gen[S]$ and $kill[S]$. $gen[S]$ is the subset of U that is generated by S , *i.e.*, all expressions that S computes. $kill[S]$ is the set of expressions that are killed by S .

Your data-flow analysis will be done on the basic-block level, so you will need a notion of $gen[S]$ and $kill[S]$ that applies to basic blocks.

1.1 Creating a control flow graph

Before you can do an iterative data-flow analysis, you will need a CFG whose nodes are basic blocks containing sequences of instructions in a middle-level intermediate representation. (C-Breeze has a data-flow analysis engine that operates at a higher level, but you should not use that for this assignment.)

C-Breeze has a built-in phase (`-dismantle`) that dismantles high-level C code into a MIR form, using `if`'s and `goto`'s instead of arbitrary control structures, and having at most one assignment per statement. Another phase (`-cfg`) divides this dismantled code into basic blocks contained in `basicblockNode`'s (a subclass of `blockNode`) and threads the blocks together in a control flow graph. In the resulting CFG, each procedure will have the following form:

```
procNode
  declNode
  blockNode (proc->body())
    decl_list
    stmt_list
      basicblockNode
        stmt_list
      basicblockNode
        stmt_list
      basicblockNode
        stmt_list
    ...
```

The body of the procedure consists of a `blockNode` in which each statement is `basicblockNode`. All of the local variables in the procedure (excluding formal parameters) will be declared in the top-level `decl_list`. All of the statements that perform computation are contained in the `stmt_lists` of the `basicblockNodes`. So, to traverse all the statements in procedure `proc` you could use the following code:

```
procNode * proc;
blockNode * body = proc->body();

// -- For all basic blocks...
for (stmt_list_p p = body->stmts().begin();
     p != body->stmts().end();
     ++p)
{
    basicblockNode * bb = (basicblockNode *) *p;
    // -- Visit all statements in the basic block...
    for (stmt_list_p q = bb->stmts().begin();
         q != bb->stmts().end();
         ++q)
    {
        stmtNode * s = *q;

        do_something_with(s);
    }
}
```

In the MIR form, the `stmtNode *s` will have one of only five types: `threeAddrNode`, `labelNode`, `gotoNode`, `ConditiongotoNode`, or `returnNode`. See the C-Breeze User Guide for more details.

In addition, each `basicblockNode` has `preds()` and `succs()` fields are lists of `basicblockNode*`'s, pointing to the predecessors and successors of the block. See the C-Breeze *Users' Guide* for more information on the `-dismantle` and `-cfg` phases.

1.2 Iterative data-flow analysis for available expressions analysis

You need to implement an iterative algorithm for solving the data-flow equations for available expressions analysis. You can use the naive algorithm or you might consider a more efficient work-list algorithm. Or, you could implement both and compare performance. If you are really ambitious, consider what it would take to implement a data-flow analysis engine into which you can plug any data-flow problem!

2 Your Assignment

- Turn in a *well-written* plan by November 7th. The plan should be a two-page document specifying the responsibilities of each team member and outlining the project report.
- Write a C-Breeze phase that computes, for each statement in each `procNode` in each unit, the set of available expressions with reaching expression information for that statement.
- Turn in a *well-written* description of the available expressions with reaching expressions data-flow analysis by November 18. How did you organize the data-flow analysis? What problems did you run into? What is the average number of available expressions at each statement, and for each one how many reaching expressions do they typically have?
- Write another C-Breeze phase, invoked by the name `"cse"` that invokes the available expressions phase and performs common sub-expression elimination.

- Come up with at least two examples of small C programs that demonstrate the effect of your phases. Note that before your phase begins, the C code should have already been dismantled and transformed into a control flow graph, so on the command line, your phase names should follow the `-cfg` flag.
- Document your code thoroughly.
- Test your phases on the benchmarks you gathered for project 2.
- Measure the performance improvement of your phases. Compile each benchmark with `gcc -O0` and for comparison with `cbreeze` and then `gcc -O0`.

```
gcc -O0 -o benchmark benchmark.c
cbz -cfg -cse -c-code benchmark.c
gcc -O0 -o benchmark-cse benchmark.p.c
```

Be sure to use `gcc -O0`, *i.e.* `gcc` with no optimizations, because `gcc` might otherwise do common sub-expression elimination itself.

NOTE: Three-person groups should also compare the performance of all different combinations involving constant propagation, copy propagation, dead-code elimination (`C-Breeze` has an implementation of this as well), and `cse`.

- Write up your results. Here are some questions to think about when you write up your results:
 - How did you solve the problem? How did you decompose your problem into smaller pieces?
 - Which was easier, understanding the data-flow analysis or coding it up?
 - Could your code be easily changed to handle a different data-flow analysis (say, constant propagation or liveness)?
 - Did this assignment help you understand the concepts of and motivations behind data-flow analysis? How? (or why not?)

3 Hints and Advice

- Start early. Start by investigating some of the technical aspects of the project: How many expressions per statement occur? How can you declare a new temporary variable (Note that the `dismantle` phase in `C-Breeze` inserts temporary variables)? How can you insert a copy statement from an expression computation value to a temporary variable? How can you replace an expression computation with a temporary variable?
- The `C-Breeze` compiler already includes a phase that performs dead-code elimination and liveness analysis. Check out how they work to help you get started.
- Break the problem into smaller pieces, such as representing the flow values, computing $gen[S]$ and $kill[S]$ for all dismantled node types, implementing the general data-flow analysis engine, implementing dead-code elimination, etc.
- To help test your program, you can use the output from your benchmarks before optimization and after. If your transformation is correct, you should get the same output.
- There are two methods on `procNode`, called `entry()` and `exit()`, that return the entry and exit basicblockNodes, respectively.

- Each statement node has associated with it a string called `comment()`. You can assign to or append to this string any string you like. When the code is unparsed back with the `-c-code` phase, the `comment()` field will appear as a comment in the code. The `-cfg` phase puts in information about predecessors and successors, but you can get rid of this if you like. When debugging your code, it can be hard to tell where in the code something is happening; you can use the `comment()` field to display debug information. For instance, you might want to know the `gen[S]` and `kill[S]` sets for each statement; you can set the comment field of each statement pointer to its hexadecimal representation, then append to the comment field the hexadecimal representation of all the expressions paired with statement sets in the `gen[S]` and `kill[S]` sets. You can also do this for the `in[S]` and `out[S]` sets of each basic block to make sure your iterative data-flow analysis is working.
- You should `delete` and set to `NULL` the `gen()` and `kill()` sets of each statement node after doing the data-flow analysis, but beyond that you don't need to worry too much about cleaning up after yourself. In particular, **don't delete nodes**; a garbage collection system for nodes is being developed for C-Breeze, so that in the future it is not necessary `delete` nodes.
- Run the `-cfg` phase on several sample C programs and examine the output to get an idea of the level of the code you'll be optimizing. You can use the dot walker you wrote in project 2 to help you visualize and debug.

4 What to turn in

You will turn in your work using WebCT. Put your paper in your copy of the `Project2` directory, tar up a clean version of `Project2` (ie. do a `make clean`), and submit the tar ball. The `Makefile` should produce one executable called `cbz` that contains all the phases mentioned above, and you should provide enough code so that the C-Breeze executable containing your phase can be reproduced by typing `make`.

Your writeup should be in one of the following formats: ASCII text, \LaTeX , or PDF.

5 Due dates

The project plan is due Monday November 7 **at 2:00pm**.

The writeup of available expressions extended with reaching expressions is due Friday November 18th **at 2:00pm**.

This final assignment is due Friday December 9 **at 2:00pm**.