CS 164, Fall 2006 **CS 164: Homework #6** **P. N. Hilfinger**

Due: Monday 4 December 2006

General instructions about homework. Check out the homework framework with the command:

```
svn checkout svn+ssh://cs164-tb@HOST/_hw/LOGIN
```

where *LOGIN* is your instructional login. In this checked-out directory, add and commit a file `hw6.txt` containing your solutions to the problems below.

1. A definition (that is, an assignment) of a simple variable is said to *reach* a point in the program if it *might be* the last assignment to that variable executed before execution reaches that point in the program. So for example, definition A below reaches points B and C, but not D:

```
x = 3         # A
if a < 2:
   x = 2
   pass       # D
else:
   y = 5
   pass       # B
pass          # C
```

Suppose we want to compute $R(p)$, the set of all definitions that reach point p in a program. Give forward rules (in the style of the lecture) for computing the *reaching definitions,* $R_{\text{out}}(s)$ for a statement s (the set of definitions that reach the point immediately after the statement) as a function of $R_{\text{in}}(s)$ (the definitions that reach the beginning) for each assignment statement s and give the rules for computing $R_{\text{in}}(s)$ as a function of the R_{out} values of its predecessors.

2. Suppose that L is a set of basic blocks, a subset of some large control-flow graph, G. Suppose also that P is a basic block outside of L with a single successor, that this successor is in L, and that P *dominates* the blocks in L, meaning that all paths from the entrance block of G to a block in L go through P first (typically L is a loop, and we call P a *preheader*). Finally, suppose that you have computed all reaching definitions (see last exercise) at all points in the program. How do you use this information to determine whether the calculation of a certain expression in one of the blocks of L, such as the right-hand side of the assignment statement

```
x := a * b
```

may be moved out of L and to the end of P? How exactly could you go about moving `a*b` without moving the assignment statement (since `x` may have been used in L *before* this point).

3. To access memory, we can use two additional intermediate-code operations:

```
r1 := *(r2+K)
*(r1+K) := r2
```

where `*` is intended to denote a memory access to an address computed from a constant (`K`) plus a register. (`K` may be an integer literal or the label of static storage—a constant address in memory). Unlike C, however, these operations just do a straight add of the register and K, with no scaling by word size.

Consider the loop

```
for i := 0 to n-1 do
   for j := 0 to n-1 do
      for k := 0 to n-1 do
         c[i,j] := c[i,j] + a[i,k] * b[k,j]
```

In this nested loop, `a`, `b`, and `c` are two-dimensional arrays of 4-byte integers. Here is a translation into intermediate code (assume that `a`, `b`, and `c` are addresses of static memory, and that all other variables are in registers):

```
  i := 0                #1
  goto L6               #2
L1:
  j := 0                #3
  goto L5               #4
L2:
  k := 0                #5
  goto L4               #6
L3:
  t1 := 4 * n           #7
  t2 := t1 * i          #8
  t3 := 4 * j           #9
  t4 := t2 + t3         #10
  t5 := *(t4 + c)       #11
  t6 := 4 * n           #12
  t7 := t6 * i          #13
  t8 := 4 * k           #14
  t9 := t7 + t8         #15
  t10 := *(t9 + a)      #16
  t11 := 4 * n          #17
  t12 := t11 * k        #18
  t13 := 4 * j          #19
  t14 := t12 + t13      #20
  t15 := *(t14 + b)     #21
  t16 := t10 * t15      #22
  t17 := t5 + t16       #23
  t18 := 4 * n          #24
  t19 := t18 * i        #25
  t20 := 4 * j          #26
  t21 := t19 + t20      #27
  *(t21+c) := t17       #28
  k := k + 1            #29
L4:
  if k < n: goto L3     #30
  j := j + 1            #31
L5:
  if j < n: goto L2     #32
  i := i + 1            #33
L6:
  if i < n: goto L1     #34
```

a. According to this code, how are the elements of the three two-dimensional arrays laid out in memory (in what order do the elements of the arrays appear)?

b. Divide the instructions into basic blocks (feel free to refer to them by number) and show the flow graph.

c. Now optimize this code as best you can, moving assignments of invariant expressions out of loops, eliminating common subexpressions, removing dead statements, performing copy propogation, etc.

Here, "eliminating a common subexpression" refers to the case where two assignments:

```
ta := E1
...
tb := E2
```

are known to give both `ta` and `tb` the same value, and where the uses of `tb` are eliminated by being replaced with `ta`.