CS 502 – Compiling and Programming Systems
Final/Qualifying Examination, 12/16/04

Instructions: Read carefully through the whole exam first and plan your time. Note the relative weight of each question and part (as a percentage of the score for the whole exam). The total points is 100 (*ie*, your grade will be the percentage of your answers that are correct).

This exam is **open book, open notes**. You are free to refer to any book or other study materials you bring to the exam room.

You have **120 minutes** to complete all four (4) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature

1. (Semantics and code generation; 20%): Recall that a Java while loop has the following syntax:

```
while (e) s
```

Expression `e` is a Boolean expression evaluated before each execution of the loop body. If `e` evaluates to false then execution continues with the statement following the loop, otherwise the body `s` is executed. Assume you are generating intermediate code trees from an abstract syntax tree as in the MiniJava project compiler. Exhibit a tuple-like template for the loop that has the minimum number of branches per iteration of the loop.

Describe how to support the `break` and `continue` statements in the context of your design, and in a way that will integrate with use of `break` and `continue` in other Java constructs (`for` loops, `switch` statements, *etc*). In particular, indicate what should happen as you generate intermediate code for each of those other constructs. The `break` statement terminates execution of its immediately enclosing loop; it has a different meaning in `switch` statements, with which you should be familiar. The `continue` statement stops execution of the current loop iteration and continues with the next iteration of the loop; it is equivalent in meaning to branching to the end of the loop *body* (in this case the end of `s`), but not outside the loop.

Answer:

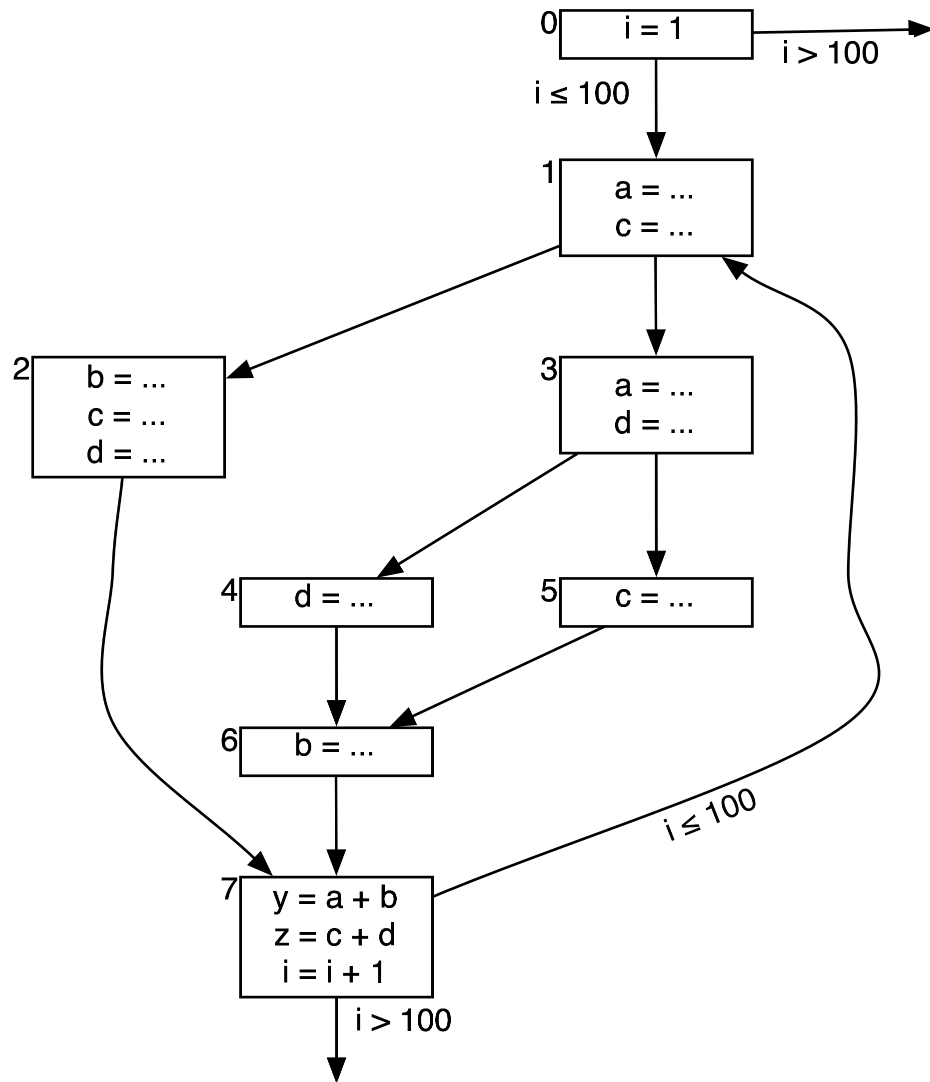
The code template has the following form:

    code for condition `e`
    branch on false L_{break}
L_{body} :
    code for body `s`
    $L_{continue}$:
    code for condition `e`
    branch on true L_{body}
L_{break} :

A `break` will generate a branch to L_{break} (outside the loop) and a `continue` will generate a branch to $L_{continue}$ (to re-execute the body).

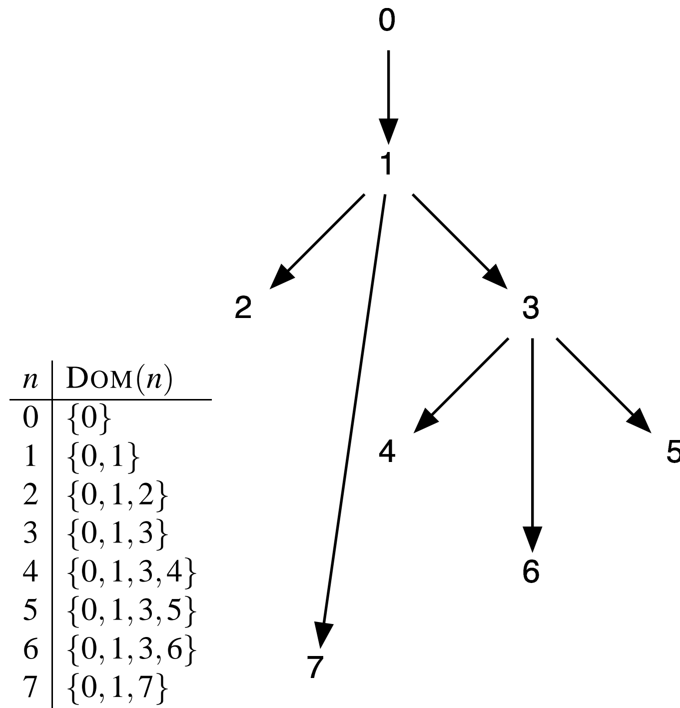
To handle `break` and `continue` more generally, we need to keep a stack of loop information. Before we process the body of a given loop (or `switch`), we push information for that loop, and after we process the body, we pop the information off. The information consists of the current `break` and `continue` labels. For example, in the above template, we would indicate L_{break} as the `break` label and $L_{continue}$ as the `continue` label. Java `for` and `do` loops would push similar label information. The `switch` statement would set up a `break` label, since it has its own meaning for `break`, but it would use the `continue` label of the next outer loop (if any; we need a way of indicating that there is none, *etc*).

2. (Dominators, SSA form; 20%) Consider the following control flow graph (CFG):



(a) (5%) Derive dominators for each basic block and draw the dominator tree for the CFG.

Answer:



n	$\text{DOM}(n)$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,3\}$
4	$\{0,1,3,4\}$
5	$\{0,1,3,5\}$
6	$\{0,1,3,6\}$
7	$\{0,1,7\}$

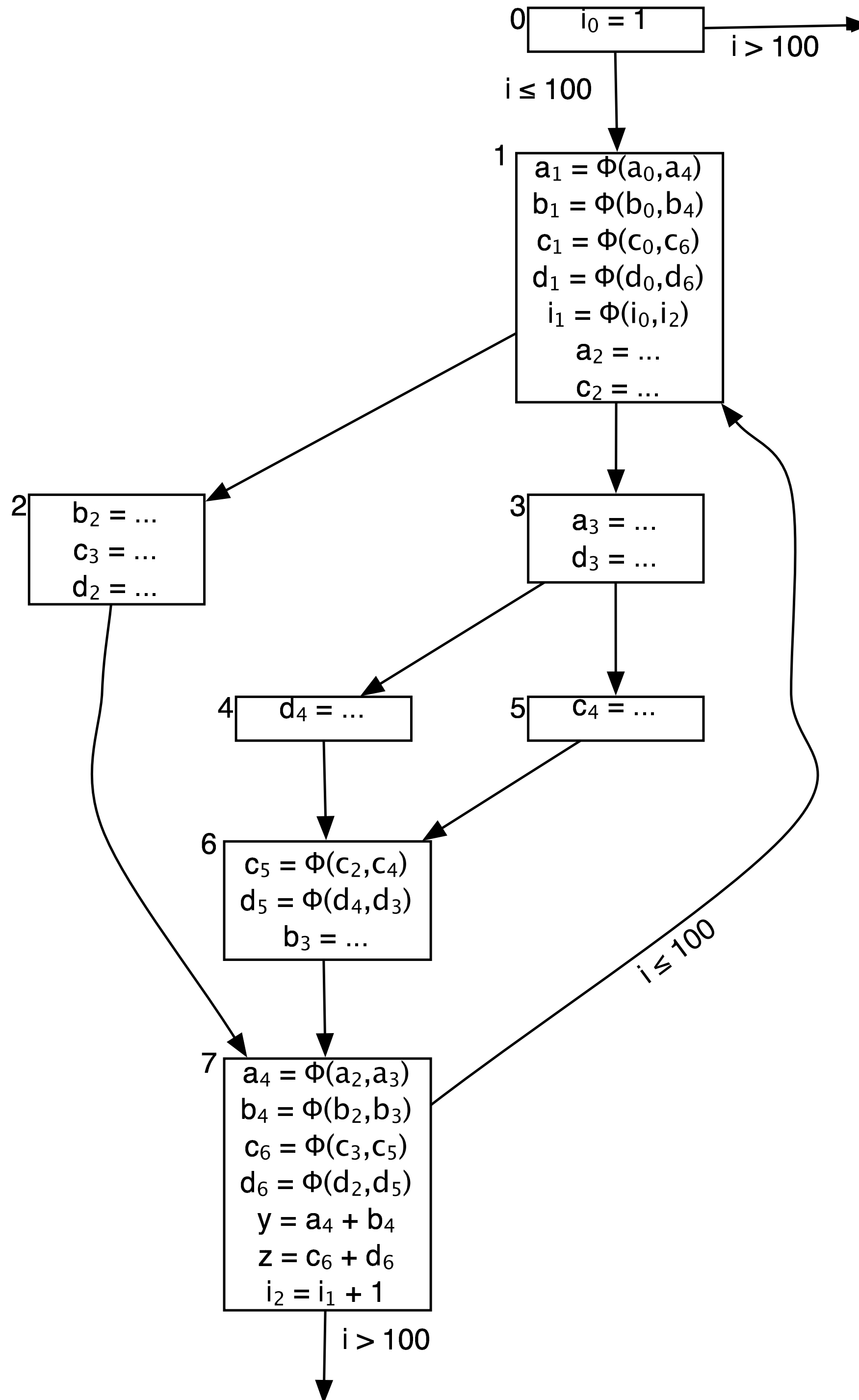
(b) (5%) Derive the dominance frontier for each basic block in the CFG.

Answer:

n	$\text{DF}(n)$
0	$\{\}$
1	$\{\}$
2	$\{7\}$
3	$\{7\}$
4	$\{6\}$
5	$\{6\}$
6	$\{7\}$
7	$\{1\}$

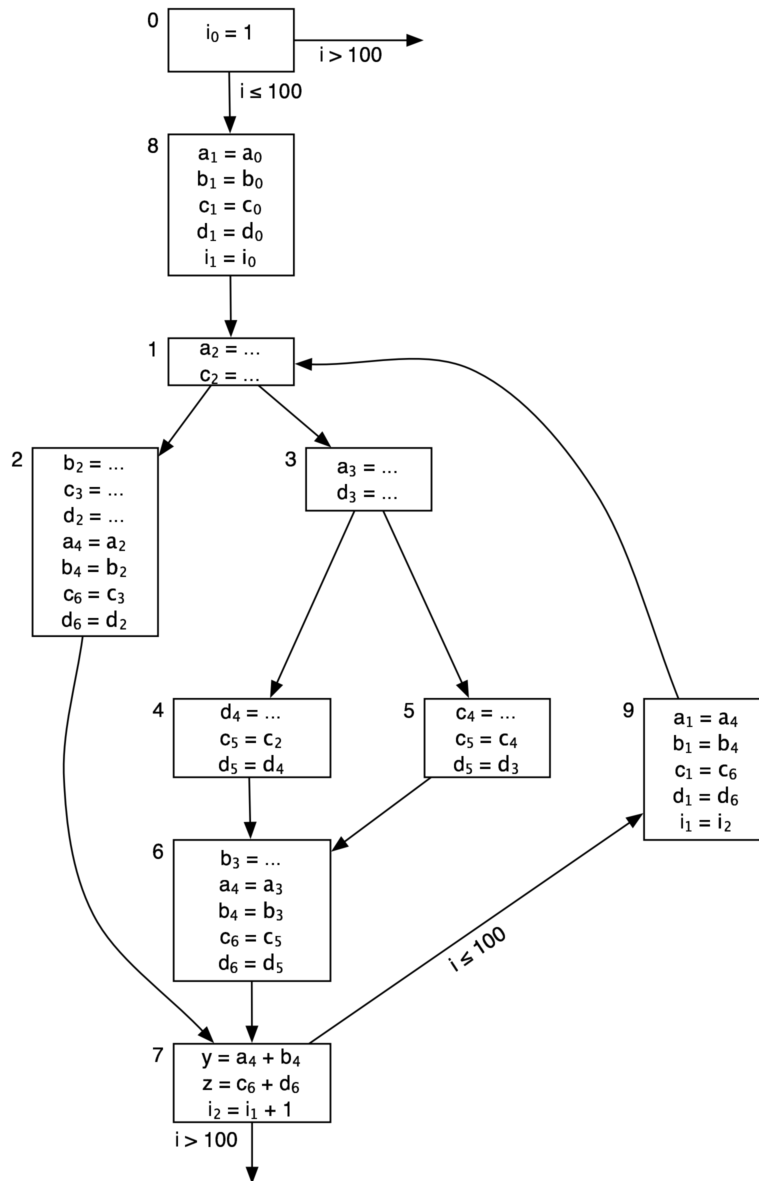
(c) (5%) Redraw the CFG in *semi-pruned* SSA form: *ie*, place ϕ nodes only for temporaries that are live across some basic block boundary.

Answer:



(d) (5%) Convert out of SSA form, redrawing the CFG with copies inserted to implement the effects of the φ-functions.

Answer:



3. (SSA-based loop optimizations; 35%) Consider the following Java method:

```
void init(int[] a, int len, int x) {
  int i = 0;
  while (i < len) {
    a[i] = x * x;
    i = i + 1;
  }
}
```

The MiniJava compiler produces intermediate code for this program along the lines of:

$$
\begin{array}{l}
\quad i \leftarrow 0 \\
\quad \texttt{if } (i < len) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{body}: \\
\quad j \leftarrow i * 4 \\
\quad p \leftarrow a_0 + j \\
\quad *p \leftarrow x * x \\
\quad i \leftarrow i + 1 \\
\quad \texttt{if } (i < len) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

where we use a C-like tuple syntax (*eg*, $*p$ denotes the word of memory referred to by the address held in temporary p) and we omit null pointer and array bounds checks.

(a) (5%) Convert this intermediate code into *semi-pruned* SSA form: *ie*, place ϕ nodes only for temporaries that are live across some basic block boundary.

Answer:

$$
\begin{array}{l}
\quad i_1 \leftarrow 0 \\
\quad \texttt{if } (i_1 < len_0) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{body}: \\
\quad i_2 \leftarrow \phi(i_1, i_3) \\
\quad j \leftarrow i_2 * 4 \\
\quad p \leftarrow a_0 + j \\
\quad *p \leftarrow x_0 * x_0 \\
\quad i_3 \leftarrow i_2 + 1 \\
\quad \texttt{if } (i_3 < len_0) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

(b) (5%) Identify the induction variables and show the code (still in SSA form) that results after strength reduction.

Answer:

Insert a preheader for the loop in which to insert initialization statements for induction variables. The basic induction variable is i, j is in the family of i ($j \leftarrow i*4$), and p is in the family of j ($p \leftarrow a+j$). Thus, introduce j' to hold current value of $j = i*4$, p' to hold current value of $p = a+j'$.
First, j':

$$
\begin{array}{l}
\quad i_1 \leftarrow 0 \\
\quad \texttt{if } (i_1 < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad \boxed{j'_1 \leftarrow i_1 * 4} \\
L_{body}: \\
\quad i_2 \leftarrow \phi(i_1, i_3) \\
\quad \boxed{j'_2 \leftarrow \phi(j'_1, j'_3)} \\
\quad j \leftarrow \boxed{j'_2} \\
\quad p \leftarrow a_0 + j \\
\quad *p \leftarrow x_0 * x_0 \\
\quad i_3 \leftarrow i_2 + 1 \\
\quad \boxed{j'_3 \leftarrow j'_2 + 4} \\
\quad \texttt{if } (i_3 < len_0) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

Then p':

$$
\begin{array}{l}
\quad i_1 \leftarrow 0 \\
\quad \texttt{if } (i_1 < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad j'_1 \leftarrow i_1 * 4 \\
\quad \boxed{p'_1 \leftarrow a_0 + j'_1} \\
L_{body}: \\
\quad i_2 \leftarrow \phi(i_1, i_3) \\
\quad j'_2 \leftarrow \phi(j'_1, j'_3) \\
\quad \boxed{p'_2 \leftarrow \phi(p'_1, p'_3)} \\
\quad j \leftarrow j'_2 \\
\quad p \leftarrow \boxed{p'_2} \\
\quad *p \leftarrow x_0 * x_0 \\
\quad i_3 \leftarrow i_2 + 1 \\
\quad j'_3 \leftarrow j'_2 + 4 \\
\quad \boxed{p'_3 \leftarrow p'_2 + 4} \\
\quad \texttt{if } (i_3 < len_0) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

(c) (5%) Show the code (still in SSA form) that results after linear test replacement. Make sure you do this iteratively for each family of induction variables.

Answer:

First i:

$$
\begin{array}{l}
\quad i_1 \leftarrow 0 \\
\quad \texttt{if } (i_1 < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad j'_1 \leftarrow i_1 * 4 \\
\quad p'_1 \leftarrow a_0 + j'_1 \\
L_{body}: \\
\quad i_2 \leftarrow \phi(i_1, i_3) \\
\quad j'_2 \leftarrow \phi(j'_1, j'_3) \\
\quad p'_2 \leftarrow \phi(p'_1, p'_3) \\
\quad j \leftarrow j'_2 \\
\quad p \leftarrow p'_2 \\
\quad *p \leftarrow x_0 * x_0 \\
\quad i_3 \leftarrow i_2 + 1 \\
\quad j'_3 \leftarrow j'_2 + 4 \\
\quad p'_3 \leftarrow p'_2 + 4 \\
\quad \texttt{if } (\boxed{j'_3 < len_0 * 4}) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

Then j:

$$
\begin{array}{l}
\quad i_1 \leftarrow 0 \\
\quad \texttt{if } (i_1 < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad j'_1 \leftarrow i_1 * 4 \\
\quad p'_1 \leftarrow a_0 + j'_1 \\
L_{body}: \\
\quad i_2 \leftarrow \phi(i_1, i_3) \\
\quad j'_2 \leftarrow \phi(j'_1, j'_3) \\
\quad p'_2 \leftarrow \phi(p'_1, p'_3) \\
\quad j \leftarrow j'_2 \\
\quad p \leftarrow p'_2 \\
\quad *p \leftarrow x_0 * x_0 \\
\quad i_3 \leftarrow i_2 + 1 \\
\quad j'_3 \leftarrow j'_2 + 4 \\
\quad p'_3 \leftarrow p'_2 + 4 \\
\quad \texttt{if } (\boxed{p'_3 < a_0 + len_0 * 4}) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

(d) (5%) Show the code that results (still in SSA form) after copy-propagation/constant-propagation/constant-folding.

Answer:

$$
\begin{array}{l}
\quad i_1 \leftarrow 0 \\
\quad \texttt{if } (\boxed{0} < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad j'_1 \leftarrow \boxed{0} \\
\quad p'_1 \leftarrow \boxed{a} \\
L_{body}: \\
\quad i_2 \leftarrow \phi(i_1, i_3) \\
\quad j'_2 \leftarrow \phi(j'_1, j'_3) \\
\quad p'_2 \leftarrow \phi(\boxed{a}, p'_3) \\
\quad j \leftarrow j'_2 \\
\quad p \leftarrow p'_2 \\
\quad *\boxed{p'_2} \leftarrow x_0 * x_0 \\
\quad i_3 \leftarrow i_2 + 1 \\
\quad j'_3 \leftarrow j'_2 + 4 \\
\quad p'_3 \leftarrow p'_2 + 4 \\
\quad \texttt{if } (p'_3 < a_0 + len_0 * 4) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

(e) (5%) Identify loop invariants and show the code (still in SSA form) that results after loop-invariant code motion.

Answer:

Loop invariants are x, len, a, $x * x$, $a + len * 4$.

$$
\begin{array}{l}
\quad i_1 \leftarrow 0 \\
\quad \texttt{if } (0 < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad j'_1 \leftarrow 0 \\
\quad p'_1 \leftarrow a \\
\quad \boxed{c \leftarrow x_0 * x_0} \\
\quad \boxed{k \leftarrow a_0 + len_0 * 4} \\
L_{body}: \\
\quad i_2 \leftarrow \phi(i_1, i_3) \\
\quad j'_2 \leftarrow \phi(j'_1, j'_3) \\
\quad p'_2 \leftarrow \phi(a, p'_3) \\
\quad j \leftarrow j'_2 \\
\quad p \leftarrow p'_2 \\
\quad *p'_2 \leftarrow \boxed{c} \\
\quad i_3 \leftarrow i_2 + 1 \\
\quad j'_3 \leftarrow j'_2 + 4 \\
\quad p'_3 \leftarrow p'_2 + 4 \\
\quad \texttt{if } (p'_3 < \boxed{k}) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

(f) (5%) Show the code that results (still in SSA form) after dead variable elimination.

Answer:

$$
\begin{array}{l}
\quad \texttt{if } (0 < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad c \leftarrow x_0 * x_0 \\
\quad k \leftarrow a_0 + len_0 * 4 \\
L_{body}: \\
\quad p'_2 \leftarrow \phi(a, p'_3) \\
\quad *p'_2 \leftarrow c \\
\quad p'_3 \leftarrow p'_2 + 4 \\
\quad \texttt{if } (p'_3 < k) \texttt{ goto } L_{body} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

(g) (5%) Convert out of SSA form and show the resulting code.

Answer:

Break critical edge on loop continue, by inserting a continue block in which to place copy statements.

$$
\begin{array}{l}
\quad \texttt{if } (0 < len_0) \texttt{ goto } L_{preheader} \texttt{ else goto } L_{break} \\
L_{preheader}: \\
\quad c \leftarrow x_0 * x_0 \\
\quad k \leftarrow a_0 + len_0 * 4 \\
\quad p'_2 \leftarrow a \\
\quad \texttt{goto } L_{body} \\
L_{continue}: \\
\quad p'_2 \leftarrow p'_3 \\
L_{body}: \\
\quad *p'_2 \leftarrow c \\
\quad p'_3 \leftarrow p'_2 + 4 \\
\quad \texttt{if } (p'_3 < k) \texttt{ goto } L_{continue} \texttt{ else goto } L_{break} \\
L_{break}:
\end{array}
$$

4. (Global data-flow analysis; 25%) In performing *lazy code motion* (LCM) an optimizer must compute information about both *availability* and *anticipability* of expressions. As discussed in class, an expression is *available* at a given point in a program if recomputing it there is redundant. Availability provides LCM with information about moving evaluations *later* in the program. Anticipability is a related notion: an expression is *anticipable* at a given point in a program if it can safely be evaluated *earlier* in the program.

   You may assume the program consists of a set of basic blocks N. Set up data flow equations to solve for anticipability, to produce solution sets $\mathsf{AnticIn}(n)$ and $\mathsf{AnticOut}(n)$ for each node $n \in N$, specifically answering these questions:

   (a) (2%) Is the problem forward-flow or backward-flow?

   Answer:

   Backward-flow

   (b) (2%) Is the problem any-path or all-paths?

   Answer:

   All paths

   (c) (4%) What are the flow values (you may assume the program consists of a set of blocks N)?

   Answer:

   The (value-numbered) expressions anticipated on entry to $n \in N$: evaluating the expression at the beginning of n has the same effect as evaluating it at its original position.
   Define:

   $\mathsf{UEExpr}(b)$ = the set of of upward-exposed expressions e in block b. If $e \in \mathsf{UEExpr}(b)$, evaluating e at the entry to block b produces the same value as evaluating it in its original position.

   $\mathsf{ExprKill}(b)$ = the set of expressions e that are killed in block b. If $e \in \mathsf{ExprKill}(b)$ then b contains a redefinition of one or more operands of e. As a consequence, evaluating e at the entry to b may produce a different value than evaluating it at the end of b.

   (d) (2%) Which is the boundary node $n_0 \in N$ (*ie*, *Entry* or *Exit*)?

   Answer:

   $n_0 = Exit$

   (e) (5%) What are the boundary conditions (*ie*, the initial values $\mathsf{AnticIn}(n_0)$ and $\mathsf{AnticOut}(n_0)$ for the boundary node)?

   Answer:

   $\mathsf{AnticOut}(Exit) = \{\}$
   $\mathsf{AnticIn}(Exit) = \{\text{all (value-numbered) expressions}\}$

   (f) (5%) State initial values of $\mathsf{AnticIn}$ and $\mathsf{AnticOut}$ for interior nodes (*ie*, $n \in N, n \neq n_0$)

   Answer:

$\mathsf{AnticOut}(n) = \mathsf{AnticIn}(n) = \{$all (value-numbered) expressions$\}$

(g) (5%) Give flow equations for $\mathsf{AnticIn}(n)$ and $\mathsf{AnticOut}(n)$

Answer:

$\mathsf{AnticOut}(n) = \bigcap_{s \in S(n)} \mathsf{AnticIn}(s)$

$\mathsf{AnticIn}(n) = \mathsf{UEExpr}(n) \cup (\mathsf{AnticOut}(n) \cap \overline{\mathsf{ExprKill}(n)})$