**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 (i.e., 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:

\[
\text{while (e) s}
\]

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 \texttt{break} and \texttt{continue} statements in the context of your design, and in a way that will integrate with use of \texttt{break} and \texttt{continue} in other Java constructs (for loops, \texttt{switch} statements, \textit{etc}). In particular, indicate what should happen as you generate intermediate code for each of those other constructs. The \texttt{break} statement terminates execution of its immediately enclosing loop; it has a different meaning in \texttt{switch} statements, with which you should be familiar. The \texttt{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 \textit{body} (in this case the end of \( s \)), but not outside the loop.

**Answer:**

The code template has the following form:

\[
\begin{align*}
\text{code for condition } e \\
\text{branch on false } L_{\text{break}} \\
L_{\text{body}} : \\
\text{code for body } s \\
L_{\text{continue}} : \\
\text{code for condition } e \\
\text{branch on true } L_{\text{body}} \\
L_{\text{break}} : 
\end{align*}
\]

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

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

\[
\begin{array}{c|c}
 n & \text{DOM}(n) \\
\hline
0 & \{0\} \\
1 & \{0,1\} \\
2 & \{0,1,2\} \\
3 & \{0,1,3\} \\
4 & \{0,1,3,4\} \\
5 & \{0,1,3,5\} \\
6 & \{0,1,3,6\} \\
7 & \{0,1,7\} \\
\end{array}
\]

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

Answer:

\[
\begin{array}{c|c}
 n & \text{DF}(n) \\
\hline
0 & \{\} \\
1 & \{\} \\
2 & \{7\} \\
3 & \{7\} \\
4 & \{6\} \\
5 & \{6\} \\
6 & \{7\} \\
7 & \{1\} \\
\end{array}
\]
(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 $\phi$-functions.

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

```java
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:

```
i ← 0
if (i < len) goto L\text{body} else goto L\text{break}
L\text{body}:
j ← i * 4
p ← a_0 + j
*p ← x * x
i ← i + 1
if (i < len) goto L\text{body} else goto L\text{break}
L\text{break}:
```

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 \( \phi \) nodes only for temporaries that are live across some basic block boundary.

**Answer:**

```
i_1 ← 0
if (i_1 < len_0) goto L_{body} else goto L_{break}
L_{body}:
i_2 ← \phi(i_1, i_3)
j ← i_2 * 4
p ← a_0 + j
*p ← x_0 * x_0
i_3 ← i_2 + 1
if (i_3 < len_0) goto L_{body} else goto L_{break}
L_{break}:
```
(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 $i$ ($j = i \times 4$), and $p$ is in the family of $j$ ($p = a + j$). Thus, introduce $j'$ to hold current value of $j = i \times 4$, $p'$ to hold current value of $p = a + j'$.

First, $j'$:

\[
i_1 \leftarrow 0
\]
\[
\text{if} \ (i_1 < \text{len}_0) \text{ goto } L_{\text{preheader}} \text{ else goto } L_{\text{break}}
\]

$L_{\text{preheader}}$:

\[
j'_1 \leftarrow i_1 \times 4
\]

$L_{\text{body}}$:

\[
i_2 \leftarrow \phi(i_1, i_3)
\]
\[
j'_2 \leftarrow \phi(j'_1, j'_3)
\]
\[
j \leftarrow j'_2
\]
\[
p \leftarrow a_0 + j
\]
\[
*p \leftarrow x_0 \times x_0
\]
\[
i_3 \leftarrow i_2 + 1
\]
\[
j'_3 \leftarrow j'_2 + 4
\]
\[
\text{if} \ (i_3 < \text{len}_0) \text{ goto } L_{\text{body}} \text{ else goto } L_{\text{break}}
\]

$L_{\text{break}}$:

Then $p'$:

\[
i_1 \leftarrow 0
\]
\[
\text{if} \ (i_1 < \text{len}_0) \text{ goto } L_{\text{preheader}} \text{ else goto } L_{\text{break}}
\]

$L_{\text{preheader}}$:

\[
j'_1 \leftarrow i_1 \times 4
\]
\[
p'_1 \leftarrow a_0 + j'_1
\]

$L_{\text{body}}$:

\[
i_2 \leftarrow \phi(i_1, i_3)
\]
\[
j'_2 \leftarrow \phi(j'_1, j'_3)
\]
\[
p'_2 \leftarrow \phi(p'_1, p'_3)
\]
\[
j \leftarrow j'_2
\]
\[
p \leftarrow p'_2
\]
\[
*p \leftarrow x_0 \times x_0
\]
\[
i_3 \leftarrow i_2 + 1
\]
\[
j'_3 \leftarrow j'_2 + 4
\]
\[
p'_3 \leftarrow p'_2 + 4
\]
\[
\text{if} \ (i_3 < \text{len}_0) \text{ goto } L_{\text{body}} \text{ else goto } L_{\text{break}}
\]

$L_{\text{break}}$:
(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 $i$:

\[
\begin{align*}
&i_1 \leftarrow 0 \\
&\text{if } (i_1 < \text{len}_0) \text{ goto } \text{L}_{\text{preheader}} \text{ else goto } \text{L}_{\text{break}} \\
\text{L}_{\text{preheader}} : \\
&j_1' \leftarrow i_1 \ast 4 \\
&p_1' \leftarrow a_0 + j_1' \\
\text{L}_{\text{body}} : \\
&i_2 \leftarrow \phi(i_1, i_3) \\
&j_2' \leftarrow \phi(j_1', j_3') \\
&p_2' \leftarrow \phi(p_1', p_3') \\
&j \leftarrow j_2' \\
&p \leftarrow p_2' \\
*&p \leftarrow x_0 \ast x_0 \\
&i_3 \leftarrow i_2 + 1 \\
&j_3' \leftarrow j_2' + 4 \\
&p_3' \leftarrow p_2' + 4 \\
&\text{if } (j_3' < \text{len}_0 \ast 4) \text{ goto } \text{L}_{\text{body}} \text{ else goto } \text{L}_{\text{break}} \\
\text{L}_{\text{break}} : \\
\end{align*}
\]

Then $j$:

\[
\begin{align*}
&i_1 \leftarrow 0 \\
&\text{if } (i_1 < \text{len}_0) \text{ goto } \text{L}_{\text{preheader}} \text{ else goto } \text{L}_{\text{break}} \\
\text{L}_{\text{preheader}} : \\
&j_1' \leftarrow i_1 \ast 4 \\
&p_1' \leftarrow a_0 + j_1' \\
\text{L}_{\text{body}} : \\
&i_2 \leftarrow \phi(i_1, i_3) \\
&j_2' \leftarrow \phi(j_1', j_3') \\
&p_2' \leftarrow \phi(p_1', p_3') \\
&j \leftarrow j_2' \\
&p \leftarrow p_2' \\
*&p \leftarrow x_0 \ast x_0 \\
&i_3 \leftarrow i_2 + 1 \\
&j_3' \leftarrow j_2' + 4 \\
&p_3' \leftarrow p_2' + 4 \\
&\text{if } (p_3' < a_0 + \text{len}_0 \ast 4) \text{ goto } \text{L}_{\text{body}} \text{ else goto } \text{L}_{\text{break}} \\
\text{L}_{\text{break}} : \\
\end{align*}
\]
(d) (5%) Show the code that results (still in SSA form) after copy-propagation/constant-propagation/constant-folding.

Answer:

\[
\begin{align*}
i_1 & \leftarrow 0 \\
\text{if } (0 < \text{len}_0) & \text{ goto } L_{\text{preheader}} \text{ else goto } L_{\text{break}} \\
L_{\text{preheader}} : \\
& j'_1 \leftarrow 0 \\
& p'_1 \leftarrow a \\
L_{\text{body}} : \\
i_2 & \leftarrow \phi(i_1, i_3) \\
j'_2 & \leftarrow \phi(j'_1, j'_3) \\
p'_2 & \leftarrow \phi(a, p'_3) \\
j & \leftarrow j'_2 \\
p & \leftarrow p'_2 \\
*p'_2 & \leftarrow x_0 \ast x_0 \\
i_3 & \leftarrow i_2 + 1 \\
j'_3 & \leftarrow j'_2 + 4 \\
p'_3 & \leftarrow p'_2 + 4 \\
\text{if } (p'_3 < a_0 + \text{len}_0 \ast 4) & \text{ goto } L_{\text{body}} \text{ else goto } L_{\text{break}} \\
L_{\text{break}} :
\end{align*}
\]

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

Answer:

Loop invariants are \( x, \text{len}, a, x \ast x, a + \text{len} \ast 4 \).

\[
\begin{align*}
i_1 & \leftarrow 0 \\
\text{if } (0 < \text{len}_0) & \text{ goto } L_{\text{preheader}} \text{ else goto } L_{\text{break}} \\
L_{\text{preheader}} : \\
& j'_1 \leftarrow 0 \\
& p'_1 \leftarrow a \\
& c \leftarrow x_0 \ast x_0 \\
& k \leftarrow a_0 + \text{len}_0 \ast 4 \\
L_{\text{body}} : \\
i_2 & \leftarrow \phi(i_1, i_3) \\
j'_2 & \leftarrow \phi(j'_1, j'_3) \\
p'_2 & \leftarrow \phi(a, p'_3) \\
j & \leftarrow j'_2 \\
p & \leftarrow p'_2 \\
*p'_2 & \leftarrow c \\
i_3 & \leftarrow i_2 + 1 \\
j'_3 & \leftarrow j'_2 + 4 \\
p'_3 & \leftarrow p'_2 + 4 \\
\text{if } (p'_3 < k) & \text{ goto } L_{\text{body}} \text{ else goto } L_{\text{break}} \\
L_{\text{break}} :
\end{align*}
\]
(f) (5%) Show the code that results (still in SSA form) after dead variable elimination.

Answer:

```c
if (0 < len0) goto L_preheader else goto L_break

L_preheader :
    c ← x0 * x0
    k ← a0 + len0 * 4

L_body :
    p′_2 ← φ(a, p′_3)
    *p′_2 ← c
    p′_3 ← p′_2 + 4
    if (p′_3 < k) goto L_body else goto L_break

L_break :
```

(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.

```c
if (0 < len0) goto L_preheader else goto L_break

L_preheader :
    c ← x0 * x0
    k ← a0 + len0 * 4
    p′_2 ← a
    goto L_body

L_continue :
    p′_2 ← p′_3

L_body :
    *p′_2 ← c
    p′_3 ← p′_2 + 4
    if (p′_3 < k) goto L_continue else goto L_break

L_break :
```
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 $\text{AnticIn}(n)$ and $\text{AnticOut}(n)$ 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:
$$\text{UEExpr}(b) = \text{the set of upward-exposed expressions } e \text{ in block } b.$$ If $e \in \text{UEExpr}(b)$, evaluating $e$ at the entry to block $b$ produces the same value as evaluating it in its original position.
$$\text{ExprKill}(b) = \text{the set of expressions } e \text{ that are killed in block } b.$$ If $e \in \text{ExprKill}(b)$ 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:

$$n_0 = \text{Exit}$$

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

Answer:

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

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

Answer:
\[ \text{AnticOut}(n) = \text{AnticIn}(n) = \{ \text{all (value-numbered) expressions} \} \]

(g) (5%) Give flow equations for AnticIn(n) and AnticOut(n)

**Answer:**

\[
\begin{align*}
\text{AnticOut}(n) &= \bigcap_{s \in S(n)} \text{AnticIn}(s) \\
\text{AnticIn}(n) &= \text{UEExpr}(n) \cup (\text{AnticOut}(n) \cap \text{ExprKill}(n))
\end{align*}
\]