1 IR and liveness

Consider the following program, which counts primes from 2 to n using the sieve method:

```c
for (i = 0; i < n; i++)
    a[i] = true;

count = 0;
for (i = 2; i*i < n; i = i + 1)
    if (a[i]) {
        count = count + 1;
        for (j = 2*i; j < n; j = j + i)
            a[j] = false
    }
```

In this program, `a` is an array of integers. At the end of the program, the `count` is the only live variable.

(a) Write three-address code for this program. Use the following operations:

<table>
<thead>
<tr>
<th>x = y OP z</th>
<th>Binary operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = OP y</td>
<td>Unary operation</td>
</tr>
<tr>
<td>x = y</td>
<td>Copy</td>
</tr>
<tr>
<td>x = k</td>
<td>Load constant k</td>
</tr>
<tr>
<td>x = [y]</td>
<td>Load from address y into x</td>
</tr>
<tr>
<td>[x] = y</td>
<td>Store y at address x</td>
</tr>
<tr>
<td>L:</td>
<td>A label</td>
</tr>
<tr>
<td>tjump x L</td>
<td>Jump to label L if x is non-zero</td>
</tr>
<tr>
<td>fjump x L</td>
<td>Jump to label L if x is zero</td>
</tr>
<tr>
<td>jump L</td>
<td>Jump to label L unconditionally</td>
</tr>
</tbody>
</table>

x, y, and z above are temporaries. k is an integer constant.

All local variables should be compiled to temporaries. Booleans can be represented as the integers 0 and 1. Arrays are allocated on the heap (i.e., the local variable `a` contains a pointer to the array data). You do not need to implement bounds checks or null checks for array accesses.

**Answer:**

```c
i = 0
L0:
```
c1 = i < n
fjump c1 L1
p1 = a + i
[p1] = 1
i = i + 1
goto L0
L1:
count = 0
i = 2
L3:
t1 = i * i
c2 = t1 < n
fjump c2 L4
p2 = a + i
c3 = [p2]
fjump c3 L2
count = count + 1
j = 2 * i
L6:
c4 = j < n
fjump c4 L5
p3 = a + j
[p3] = 0
j = j + 1
goto L6
L5:
L2:
i = i + 1
goto L3
L4:

(b) Construct the control flow graph from the IR in (a).
(c) Compute the live variables at each program point in the CFG.
2 Dataflow

In this problem, we will design a dataflow analysis that computes whether an expression is partially available. An expression is partially available if it has been computed on some path from the entry and has not been modified subsequently.

(a) Is this a forward or backward analysis? Is this a may or must analysis?

Answer: Forward, may

(b) Define the partial order and the meet operator for this lattice.

Answer: Ordering: set inclusion, meet: union
(c) Define the transfer functions for each of the following three-address instructions:

- \( x = y \text{ OP } z \)
- \( x = [y] \)
- \([x] = y\)
- \( x = y \)
- \( x = k \)

\textit{Answer:}

- \( x = y \text{ OP } z \)
  
  Generate \( y \text{ OP } z \). Kill any expression with \( x \).

- \( x = [y] \)
  
  Generate \([y]\). Kill any expression with \( x \).

- \([x] = y\)
  
  Generate \([x]\). Kill nothing. Partial credit for generate nothing.

- \( x = y \)
  
  Generate nothing. Kill any expression with \( x \).

- \( x = k \)
  
  Generate nothing. Kill any expression with \( x \).

(d) Using the lattice and the transfer functions of the previous parts, show how the dataflow analysis works for the following program:

\begin{verbatim}
  i = 0
  s = 0
  TOP:
  z = i < 10
  fjump z END
  v = a + i
  x = [v]
  w = a + 40
  y = [w]
  t = s + x
  s = t + y
  jump TOP
END:
  u = a + 40
  [u] = s
\end{verbatim}

Note that the transfer functions for jump instructions do not change the flow information, although the meet operations may change the flow information.

\textit{Answer:}

First note, I left out \( i = i + 1 \) in the loop. As a consequence, the program, when run, will loop forever. This should not affect the answer except that \( i \langle 10 \) would be killed.

Pass 1:

\begin{verbatim}
{}  i = 0
\end{verbatim}
s = 0
{}  TOP:
{}  z = i < 10
{i < 10}
   fjump z END
{i < 10}
   v = a + i
{i < 10, a + i}
   x = [v]
{i < 10, a + i, [v]}
   w = a + 40
{i < 10, a + i, [v], a + 40}
   y = [w]
{i < 10, a + i, [v], a + 40, [w]}
   t = s + x
{i < 10, a + i, [v], a + 40, [w], s + x}
   s = t + y
{i < 10, a + i, [v], a + 40, [w], t + y} -- kills s + x
   jump TOP
{i < 10, a + i, [v], a + 40, [w], t + y}  END:
{i < 10}
   u = a + 40
{i < 10, a + 40}
   [u] = s
{i < 10, a + 40, [u]}

Pass 2:

{}  i = 0
{}  s = 0
{}  TOP:
{}  z = i < 10
{i < 10, a + i, [v], a + 40, [w], t + y}
   fjump z END
{i < 10, a + i, [v], a + 40, [w], t + y}
   v = a + i
{i < 10, a + i, [v], a + 40, [w], t + y}
   x = [v]
{i < 10, a + i, [v], a + 40, [w], t + y}
   w = a + 40
{i < 10, a + i, [v], a + 40, t + y} -- kills [w]
   y = [w]
{i < 10, a + i, [v], a + 40, [w]} -- kills t+y, gens [w]
   t = s + x
{i < 10, a + i, [v], a + 40, [w], s + x}
   s = t + y
\{i < 10, a + i, [v], a + 40, [w], t + y\}
    jump TOP
\{i < 10, a + i, [v], a + 40, [w], t + y\}
    END:
\{i < 10, a + i, [v], a + 40, [w], t + y\}
    u = a + 40
\{i < 10, a + i, [v], a + 40, [w], t + y\}
    [u] = s
\{i < 10, a + i, [v], a + 40, [w], t + y, [u]\}

Now at a fixed point.