Let’s make data flow concrete:

*example from Appel’s book*

- A variable is live if its current value will be used
  - Variable b is used in 4, so it is live on the (3,4) edge
  - Since 3 doesn’t assign into b, b is also live on (2,3)
  - Statement 2 assigns to b, so the contents of b on the (1,2) edge aren’t needed by anyone: b is dead on that edge
  - So variable b is live on (2,3) and (3,4), but nowhere else

```
1  a := 0
2  b := a + 1
3  c := c + b
4  a := b * 2
5  a < n
6  return c
```

**a’s liveness?**

- Live on (1,2) and also on (4,5) and (5,2)
- But it’s not live on (2,3) and (3,4), even though it has a defined value
- It’s not used before a new assignment to a is made
- (Since a and b are never live on the same edges, they could share a register)

```
1  a := 0
2  b := a + 1
3  c := c + b
4  a := b * 2
5  a < n
return c
```

### def-use

<table>
<thead>
<tr>
<th>Node #</th>
<th>def</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{b}</td>
<td>{a}</td>
</tr>
<tr>
<td>3</td>
<td>{c}</td>
<td>{b,c}</td>
</tr>
<tr>
<td>4</td>
<td>{a}</td>
<td>{b}</td>
</tr>
<tr>
<td>5</td>
<td>{a}</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>{c}</td>
<td></td>
</tr>
</tbody>
</table>

**Liveness defined by def-use**

- A variable is live on an edge if there is a directed path from the edge to a use of the variable that does not go through any def
  - That is, it isn’t killed by another def
  1. If a statement uses a variable, the variable is live on entry to that node
  2. If a variable is live on entry to a node, then it is live on exit from all predecessor nodes
  3. If a variable is live on exit from a node and is not defined by the node, then it is live on entry to the node

**In equation form**

- \( \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \)
- Variables live at node \( n \) are those used in \( n \) plus those that are live when they leave \( n \) except those defined in \( n \)
  - \( \text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s] \)
  - Find all nodes that are successors of \( n \); any variable that leaves \( n \) live enters those nodes live
  - Our goal is to compute liveness \( \text{in} \) and \( \text{out} \) given \( \text{def} \) and \( \text{use} \)
  - Note that \( \text{in} \) is defined in terms of \( \text{out} \), and \( \text{out} \) in terms of \( \text{in} \)
Algorithm:
solve these equations by iteration

for each \( n \)

\[
\text{in}[n] := \emptyset; \quad \text{out}[n] := \emptyset;
\]

repeat for each \( n \)

\[
\text{in}'[n] := \text{in}[n]; \quad \text{out}'[n] := \text{out}[n]
\]

\[
\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

\[
\text{out}[n] := \cup s \in \text{succ}[n] \text{ in}[s]
\]

until \( \text{in}'[n] = \text{in}[n] \) and \( \text{out}'[n] = \text{out}[n] \) for all \( n \)

Rows: nodes  • columns: iterations

<table>
<thead>
<tr>
<th>use</th>
<th>def</th>
<th>in1</th>
<th>out1</th>
<th>in2</th>
<th>out2</th>
<th>in3</th>
<th>out3</th>
<th>in4</th>
<th>out4</th>
<th>in5</th>
<th>out5</th>
<th>in6</th>
<th>out6</th>
<th>in7</th>
<th>out7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a b</td>
<td>a b</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
<td>a c</td>
</tr>
<tr>
<td>3</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
<td>b c</td>
</tr>
<tr>
<td>4</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
<td>b a</td>
</tr>
<tr>
<td>5</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
<td>a a</td>
</tr>
<tr>
<td>6</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
<td>c c</td>
</tr>
</tbody>
</table>

Note: Node order (left column) reversed!

Change order of iterations: from 6 to 1

<table>
<thead>
<tr>
<th>use</th>
<th>def</th>
<th>in1</th>
<th>out1</th>
<th>in2</th>
<th>out2</th>
<th>in3</th>
<th>out3</th>
<th>in4</th>
<th>out4</th>
<th>in5</th>
<th>out5</th>
<th>in6</th>
<th>out6</th>
<th>in7</th>
<th>out7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

Some observations

The iteration order is key to performance
- For this data flow computation, since it is in some sense naturally “backward”, it tends to be more efficient iterating over the CFG in “reverse”
- A simple depth-first search can be used to find an effective ordering

In practice, with efficient representations, the algorithm is usually O(N) or O(N^2) for a program of N nodes

- Bit vectors are commonly used to represent the sets of variables; good for dense sets
- Sorted linked lists are also useful; good for sparse sets

It turns out that there are multiple solutions to the dataflow equations; however, there is one least (minimal) solution

Finally, remember this is conservative: it may show a variable is live when it in fact never is