

Contents of Lecture 9

- Depth First Search for DCE (Dead Code Elimination)
- Liveness Analysis for DCE
- SSA-based DCE
- Control Dependencies
- Control Flow Graph Simplification

Two Simple Forms of Dead Code Elimination

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int    a;
```

```
    a = 1;
```

```
    a = a + 2;
```

```
    goto L;
```

```
    printf("a = %d\n", a);
```

```
L:
```

```
    return 0;
```

```
}
```

- DFS
- Liveness Analysis

Depth First Search and Dominance Analysis

- DFS from the start vertex visits all basic blocks reachable from the start vertex, obviously.
- All other vertices are removed before performing dominance analysis.
- For some minor modifications of the control flow graph an existing dominator tree can be updated.
- In general, it's easier and probably faster to recompute the dominator tree from scratch.

Limitations of DCE Based on Liveness Analysis

```
for (i = 0; i < n; ++i)
    a = a + i * i;
return;
```

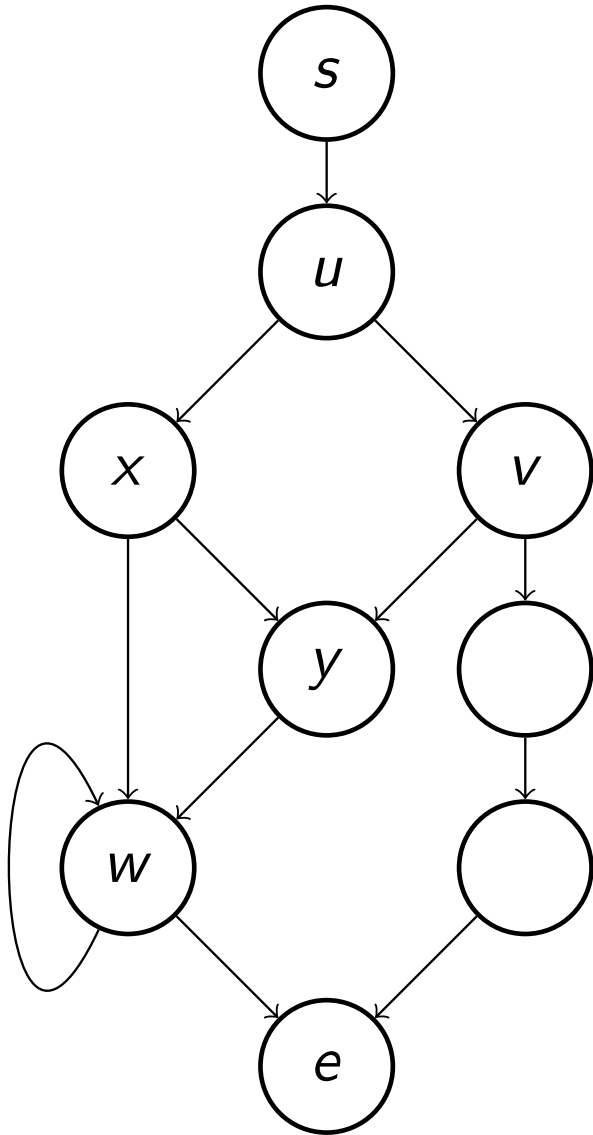
- The variable `a` is live in the loop but will not affect program output.
- The loop should be deleted but it cannot be using DCE based on liveness.

DCE Based on Observable Output

- The correct approach to DCE is to delete all code which cannot affect the observable output.
- In each function, some instructions are marked as **live**, e.g. calls to `printf`, and are put in a worklist.
- Then, recursively, all instructions which provide input to a live instruction is marked as live and put on the worklist.
- Eventually no new instructions are marked as live and all other instructions can be deleted (but read more about branches first!).
- Instructions initially marked live include: function calls, memory writes, and return instructions, and in **vcc** additionally the **put** and **get** instruction.
- Why did it take more than 30 years to discover form of DCE?

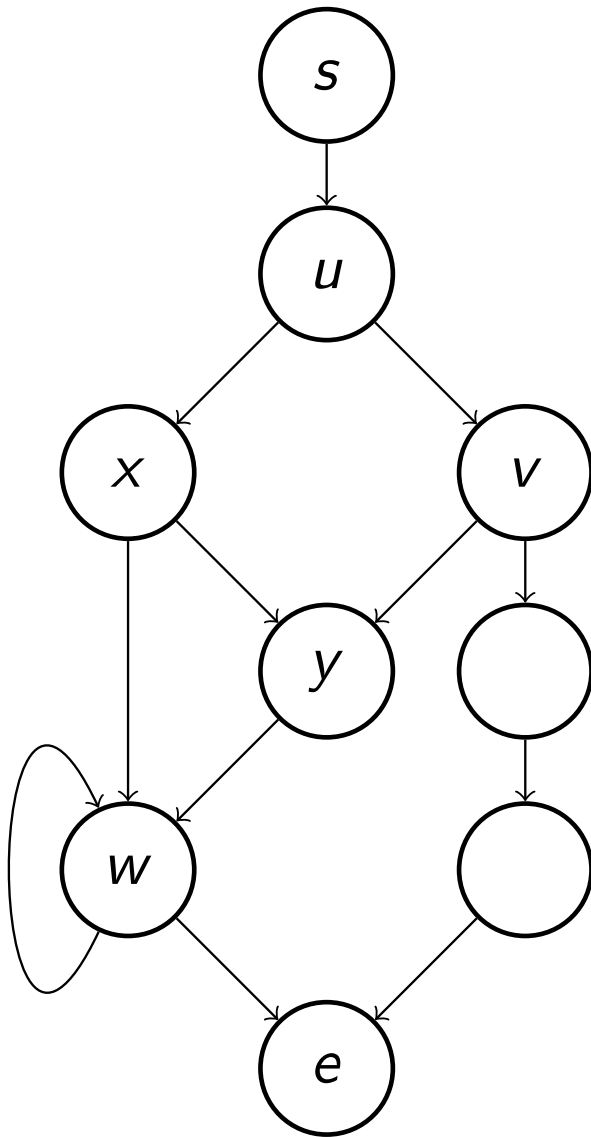
- The main reason why it was not invented earlier is that the other approaches usually were sufficient.
- With SSA Form, however, it's more likely there will be lots of instructions, in particular ϕ -functions, which remain after other optimizations.
- For example, operator strength reduction explicitly copies and modifies the strongly connected components in the SSA Graph of induction variables, which can leave a lot of work to DCE.
- The article in Transactions on Programming Languages and Systems (TOPLAS) which presented SSA Form also presented the DCE algorithm we will study.

Conditional Branches



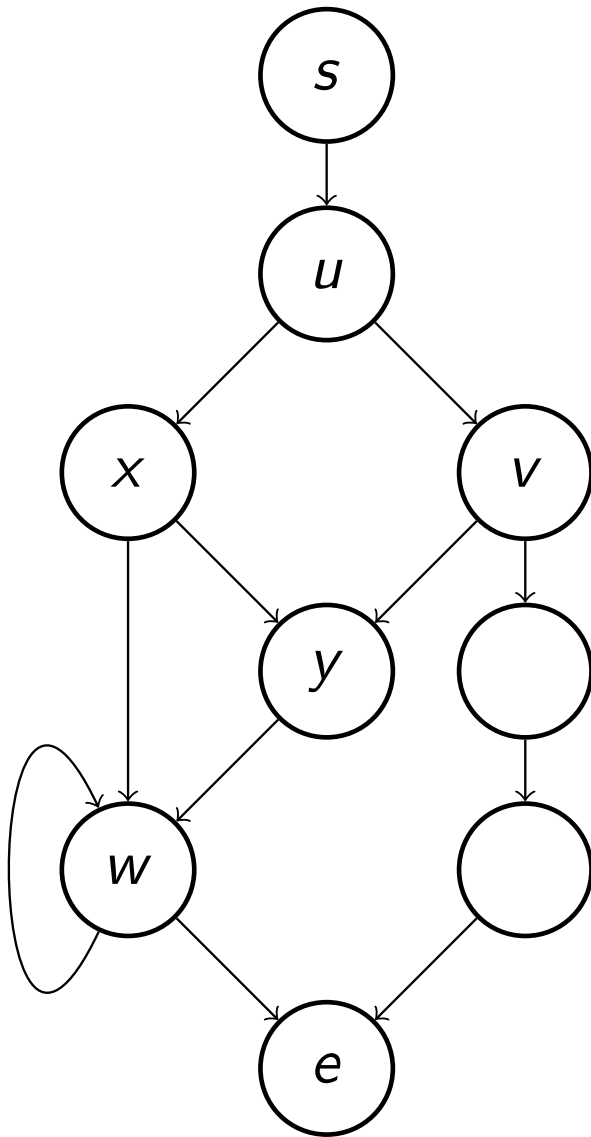
- Assume there is a live instruction in vertex *x*.
- The DCE algorithm must assure execution actually reaches *x* exactly as the original program would.
- Therefore some conditional branch instructions (and the instructions providing their input etc) which branch to *x* must also be marked live.
- In this example the branch in *u* controls whether *x* certainly will be executed.
- For vertex *w*, the vertices which can control that *w* will be executed are *u*, *v*, and *w*.

Reverse Control Flow Graph and Post Dominance



- The reverse control flow graph is the control flow graph with the direction of each edge reversed, where s and e have switched roles, and is written **RCFG**.
- A vertex w **postdominates** v if every path from v to the exit vertex e contains w , and we write it $w \leq v$.
- A vertex w **strictly postdominates** v if $w \leq v$ and $w \neq v$, and we write it $w \ll v$.
- Thus we have $w \leq x$ and $w \leq y$.
- Post dominance can be computed as dominance in the RCFG.

Control Dependence



- A non-null path is a path with at least one edge: w is a null path, while (w, w) and (u, x, w, w, w, e) are not.
- A vertex v is control dependent on vertex u , written $u \delta^c v$ if
 - 1 there exists a non-null path from u to v and v postdominates every vertex on the path after u , and
 - 2 v does not strictly postdominate u .
- The set of vertices which are control dependent on u is denoted $CD(u)$ and the set of vertices a vertex v is control dependent on is denoted $CD^{-1}(v)$.

Lemma 5.13

Lemma

Assume $v \in \text{succ}(u)$ and there is a path $p = (v_0 = v, v_1, \dots, v_k = w)$ from v to w . Then $w \preceq v \Leftrightarrow w \preceq v_i$ for every vertex v_i on p .

Proof.

Let us show \Rightarrow first. Assume therefore in contradiction that there exists some $0 < i < k$ such that $w \not\preceq v_i$. Thus there exists a path from v_i to e which does not include w . Then there is a path from v to v_i to e which avoids w which is a contradiction. Hence $w \preceq v_i$. Since v is on the path, \Leftarrow follows directly. □

Dominance Frontiers

- Recall that the *dominance frontier* of a vertex u is the set of vertices v such that u dominates a predecessor of v but does not strictly dominate v :

$$DF(u) \stackrel{\text{def}}{=} \{v \mid (\exists p \in \text{pred}(v)) \ u \succeq p \wedge uv \not\rightarrow\}$$

- With Lemma 5.13 we can simplify the definition of control dependence and show that it is equivalent to dominance frontiers in the reverse control flow graph.
- First the simplified definition: a vertex v is control dependent on $u \in CD^{-1}(v)$ if v postdominates a successor of u but does not strictly postdominate u :

$$CD^{-1}(v) \stackrel{\text{def}}{=} \{u \mid (\exists s \in \text{succ}(u)) \ \wedge \ v \preceq s \wedge v \not\leftarrow u\}$$

Equivalence of CD in CFG and DF in RCFG

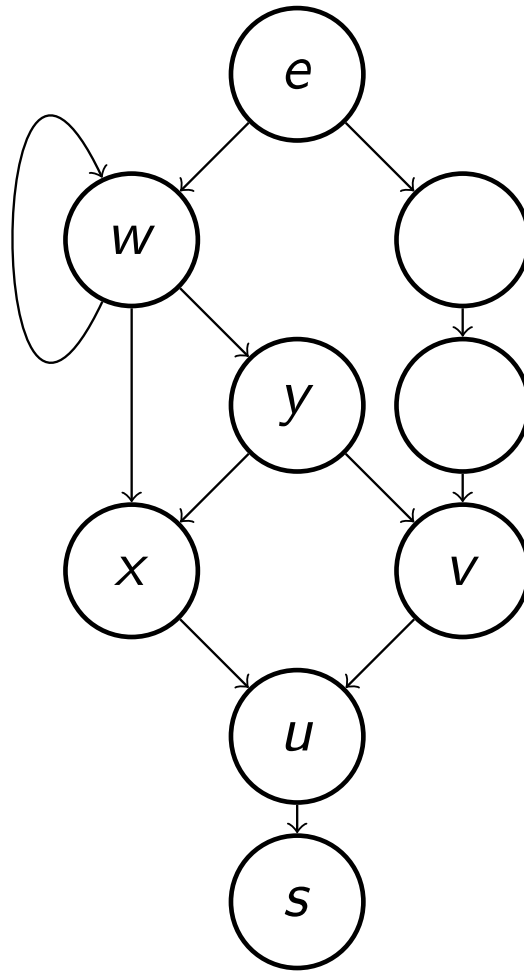
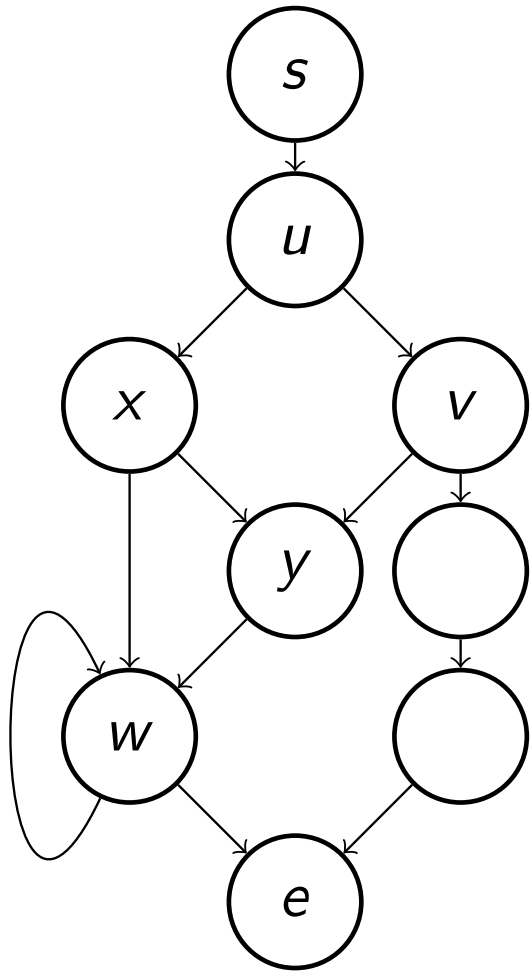
Theorem

$u \delta^c v$ in CFG $\Leftrightarrow u \in DF(v)$ in RCFG.

Proof.

This follows from Lemma 5.13, since $u \delta^c v$ in CFG means v postdominates a successor of u but does not strictly postdominate u , which in RCFG means v dominates a predecessor of u but v does not strictly dominate u , ie $u \in DF(v)$. □

Example CFG and RCFG

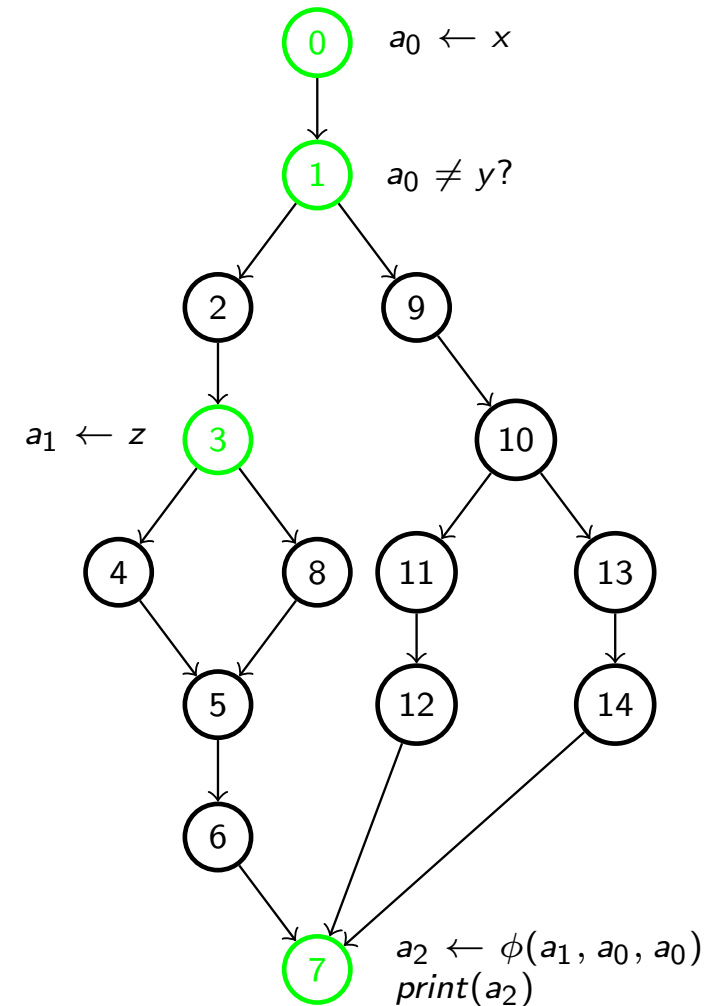


The DCE Algorithm

```
procedure eliminate_dead_code(G)
  for each statement s do
    if (s is prelive) {
      live(s) ← true
      add s to worklist
    } else
      live(s) ← false
  worklist ← prelive
  while (worklist ≠ ∅) do {
    take s from worklist
    v ← vertex(s)
    live(v) ← true
    for each source operand  $\omega$  of s do {
      t ← def( $\omega$ )
      if (not live(t)) {
        live(t) ← true
        add t to worklist
      }
    }
  }
  for each vertex  $v \in CD^{-1}(\text{vertex}(s))$  do {
    t ← multiway branch of v
    if (not live(t)) {
      live(t) ← true
      add t to worklist
    }
  }
}
for each statement s do
  if (not live(s) and  $s \notin \{\text{label}, \text{branch}\}$ )
    delete s from vertex(S)
simplify(G)
```

Simplifying the CFG after DCE

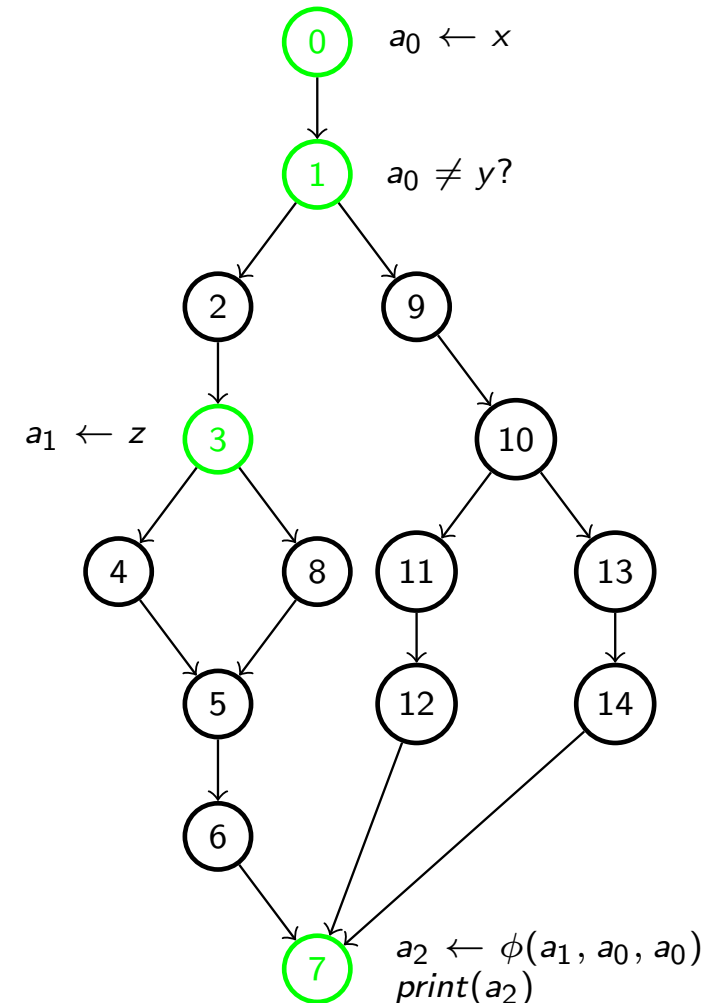
```
procedure simplify(G)  
  live(e)  $\leftarrow$  true  
  modified  $\leftarrow$  false  
  for each vertex u  $\in$  G do {  
    if (not live(u))  
      continue  
    for each v  $\in$  succ(u) do {  
      if (live(v))  
        continue  
      w  $\leftarrow$  ipdom(v) /* idom in RCFG */  
      while (not live(w))  
        w  $\leftarrow$  ipdom(w)  
      replace (u, v) with (u, w)  
      update the branch in u to its new target w  
      update  $\phi$ -functions in w if necessary  
      modified  $\leftarrow$  true  
    }  
  }  
  if (modified) {  
    delete vertices from G which now have become unreachable  
    update dominator tree DT  
  }  
end
```



- Green denotes live vertices

Processing 0

```
procedure simplify(G)
  live(e) ← true
  modified ← false
  for each vertex u ∈ G do {
    if (not live(u))
      continue
    for each v ∈ succ(u) do {
      if (live(v))
        continue
      w ← ipdom(v) /* idom in RCFG */
      while (not live(w))
        w ← ipdom(w)
      replace (u, v) with (u, w)
      update the branch in u to its new target w
      update  $\phi$ -functions in w if necessary
      modified ← true
    }
  }
  if (modified) {
    delete vertices from G which now have become unreachable
    update dominator tree DT
  }
end
```



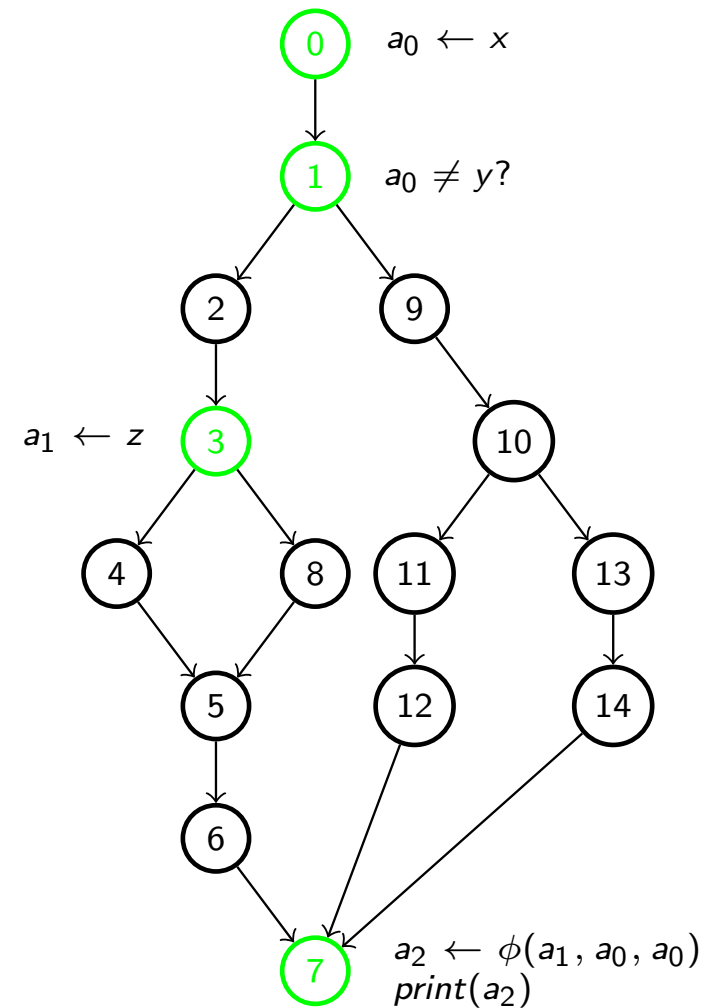
- Only successor is live.

Processing 1: Edge (1, 2)

```

procedure simplify(G)
  live(e)  $\leftarrow$  true
  modified  $\leftarrow$  false
  for each vertex u  $\in$  G do {
    if (not live(u))
      continue
    for each v  $\in$  succ(u) do {
      if (live(v))
        continue
      w  $\leftarrow$  ipdom(v) /* idom in RCFG */
      while (not live(w))
        w  $\leftarrow$  ipdom(w)
      replace (u, v) with (u, w)
      update the branch in u to its new target w
      update  $\phi$ -functions in w if necessary
      modified  $\leftarrow$  true
    }
  }
  if (modified) {
    delete vertices from G which now have become unreachable
    update dominator tree DT
  }
end

```

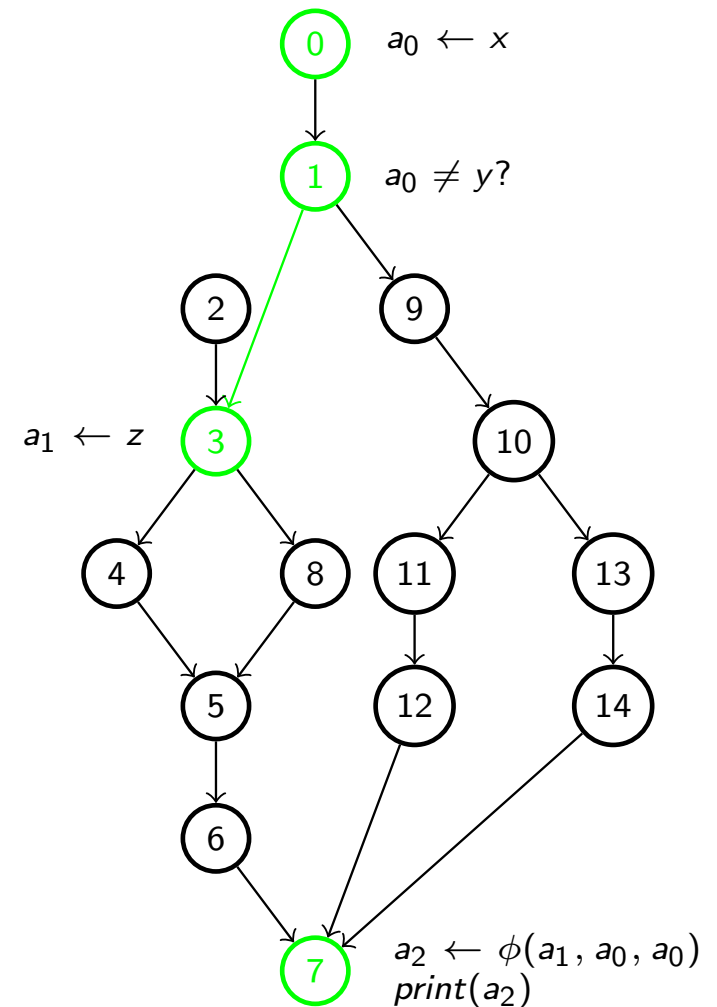


- 2 is dead. Nearest live is 3.

Processing 1: Edge (1, 2)

```

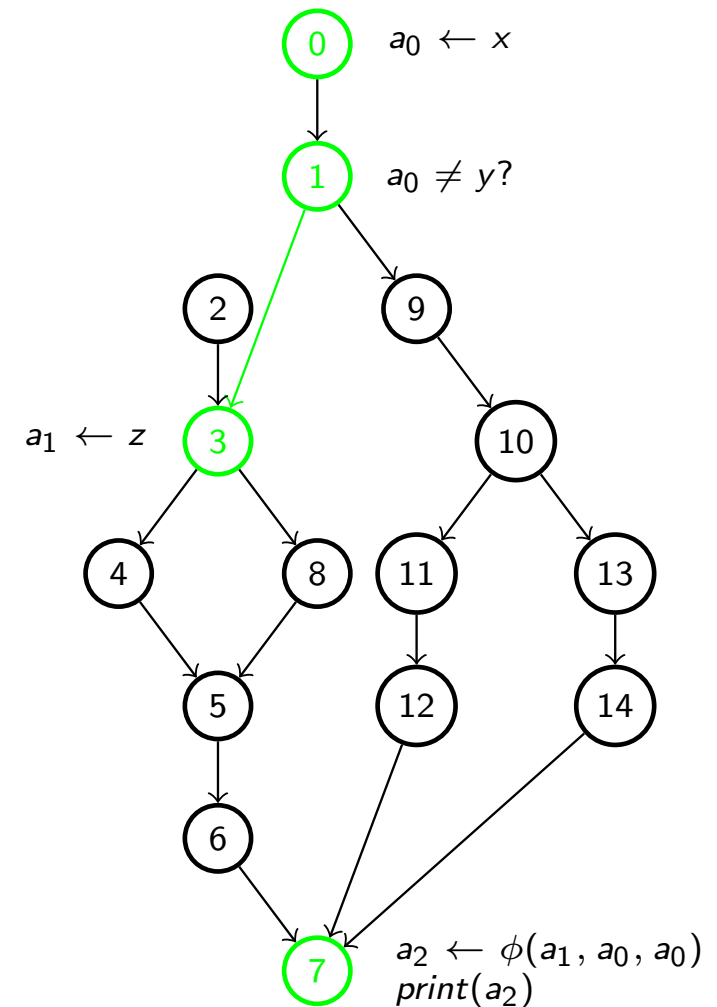
procedure simplify(G)
  live(e)  $\leftarrow$  true
  modified  $\leftarrow$  false
  for each vertex u  $\in$  G do {
    if (not live(u))
      continue
    for each v  $\in$  succ(u) do {
      if (live(v))
        continue
      w  $\leftarrow$  ipdom(v) /* idom in RCFG */
      while (not live(w))
        w  $\leftarrow$  ipdom(w)
      replace (u, v) with (u, w)
      update the branch in u to its new target w
      update  $\phi$ -functions in w if necessary
      modified  $\leftarrow$  true
    }
  }
  if (modified) {
    delete vertices from G which now have become unreachable
    update dominator tree DT
  }
end
  
```



- 2 is dead. Nearest live is 3.

Processing 1: Edge (1, 9)

```
procedure simplify(G)
  live(e) ← true
  modified ← false
  for each vertex u ∈ G do {
    if (not live(u))
      continue
    for each v ∈ succ(u) do {
      if (live(v))
        continue
      w ← ipdom(v) /* idom in RCFG */
      while (not live(w))
        w ← ipdom(w)
      replace (u, v) with (u, w)
      update the branch in u to its new target w
      update  $\phi$ -functions in w if necessary
      modified ← true
    }
  }
  if (modified) {
    delete vertices from G which now have become unreachable
    update dominator tree DT
  }
end
```



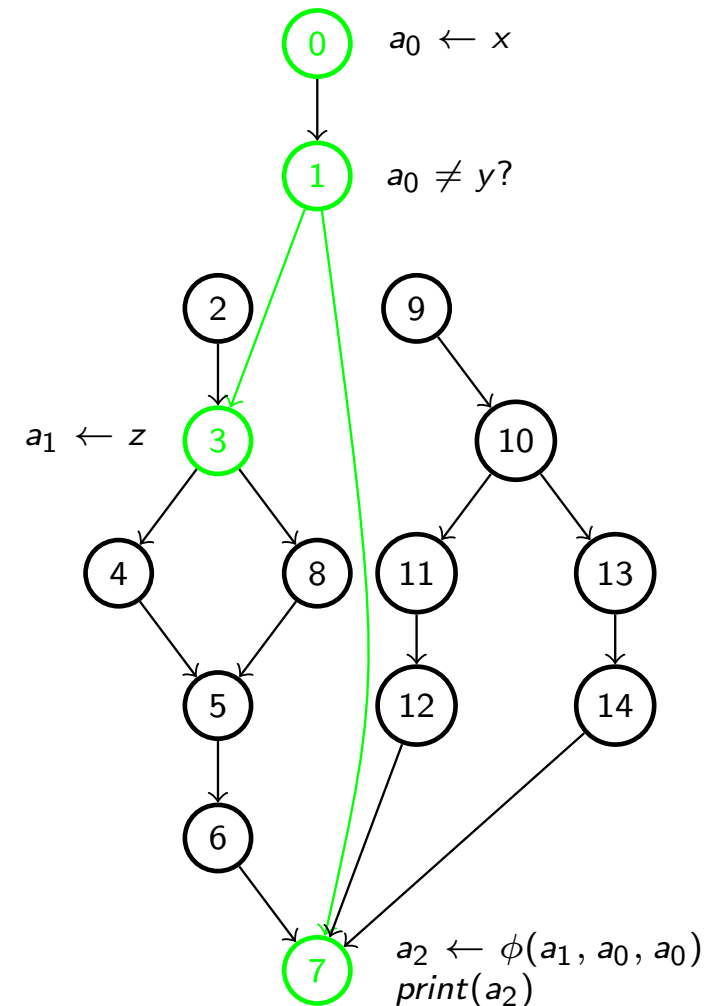
- 9 is dead. Nearest live is 7.

Processing 1: Edge (1, 9)

```

procedure simplify(G)
  live(e)  $\leftarrow$  true
  modified  $\leftarrow$  false
  for each vertex u  $\in$  G do {
    if (not live(u))
      continue
    for each v  $\in$  succ(u) do {
      if (live(v))
        continue
      w  $\leftarrow$  ipdom(v) /* idom in RCFG */
      while (not live(w))
        w  $\leftarrow$  ipdom(w)
      replace (u, v) with (u, w)
      update the branch in u to its new target w
      update  $\phi$ -functions in w if necessary
      modified  $\leftarrow$  true
    }
  }
  if (modified) {
    delete vertices from G which now have become unreachable
    update dominator tree DT
  }
end

```



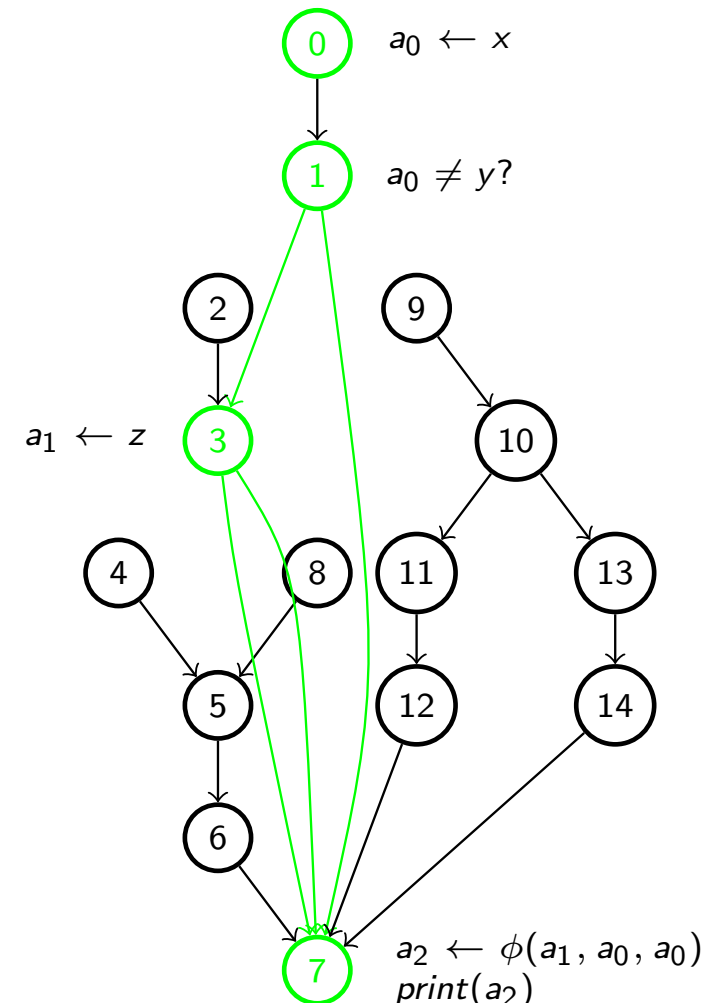
- Must fix $\phi(a)$ in 7.

Result of Processing 3

```

procedure simplify(G)
  live(e)  $\leftarrow$  true
  modified  $\leftarrow$  false
  for each vertex u  $\in$  G do {
    if (not live(u))
      continue
    for each v  $\in$  succ(u) do {
      if (live(v))
        continue
      w  $\leftarrow$  ipdom(v) /* idom in RCFG */
      while (not live(w))
        w  $\leftarrow$  ipdom(w)
      replace (u, v) with (u, w)
      update the branch in u to its new target w
      update  $\phi$ -functions in w if necessary
      modified  $\leftarrow$  true
    }
  }
  if (modified) {
    delete vertices from G which now have become unreachable
    update dominator tree DT
  }
end

```



- Later remove one (3, 7)!
- Keep only live vertices.