CS443: Compiler Construction

Lecture 13: Liveness Analysis

Stefan Muller

Based on material by Steve Zdancewic

A variable is “live” when its value is needed

```
int f(int x) {
  int a = x + 2;
  int b = a * a;
  int c = b + x;
  return c;
}
```

- ← x is live (between `int f(int x) {` and `int a = x + 2;`)
- ← a and x are live (between `int a = x + 2;` and `int b = a * a;`)
- ← b and x are live (between `int b = a * a;` and `int c = b + x;`)
- ← c is live (between `int c = b + x;` and `return c;`)

Liveness =/= Scope

```
int f(int x) {
  int a = x + 2;
  int b = a * a;
  int c = b + x;
  return c;
}
```

x is live

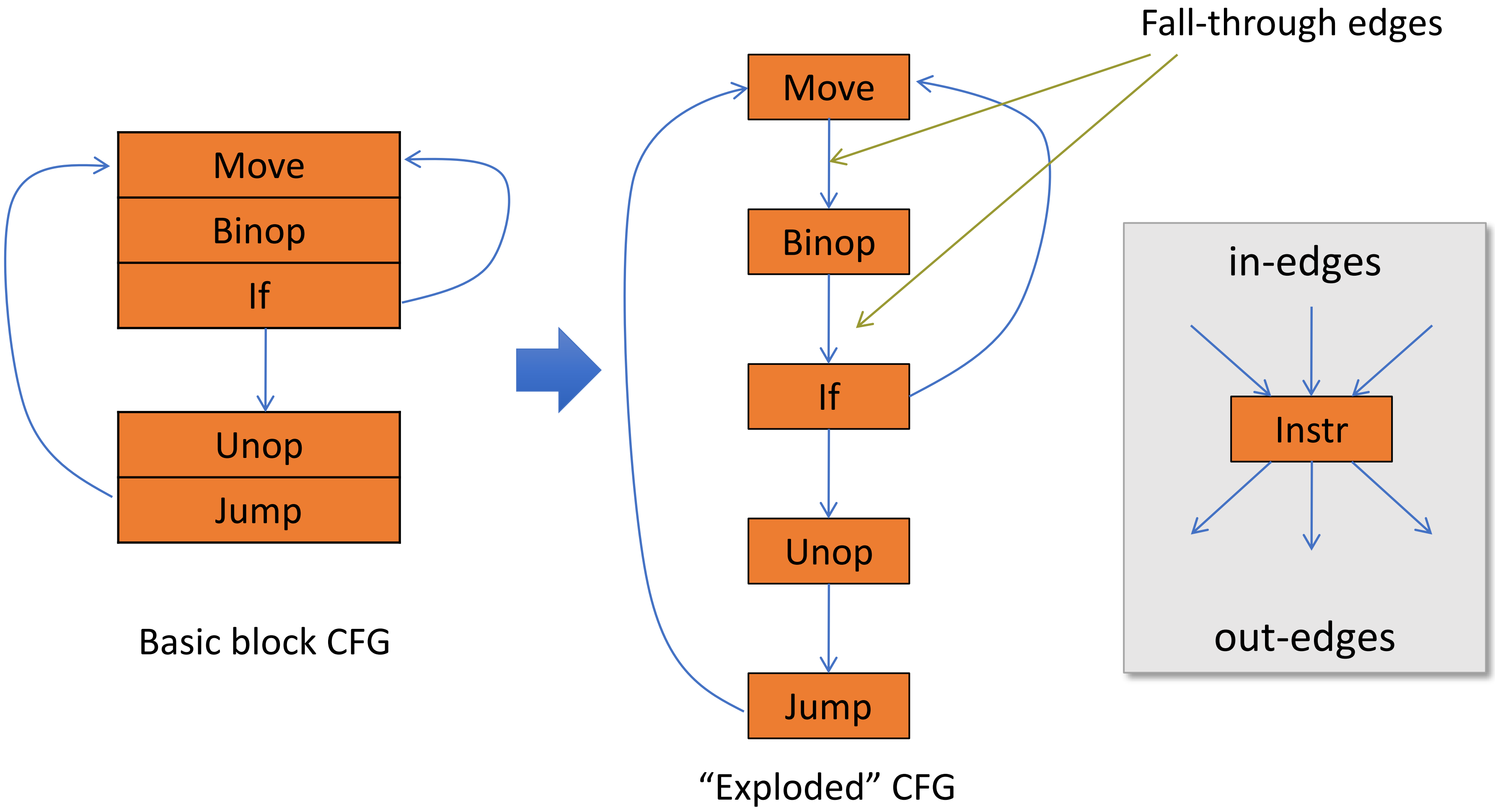
a and x are live

b and x are live

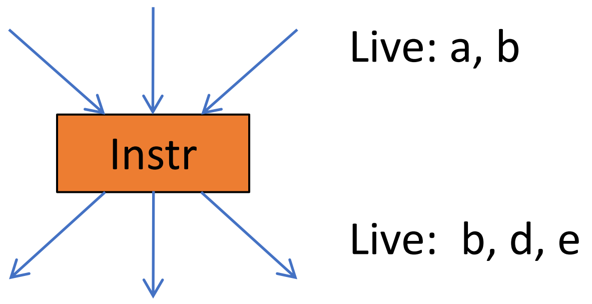
c is live

- *Scopes* of a, b, c, x overlap, *Live ranges* of a, b, c don’t.
- Why is this useful?
  - a, b, c can all be in the same register!

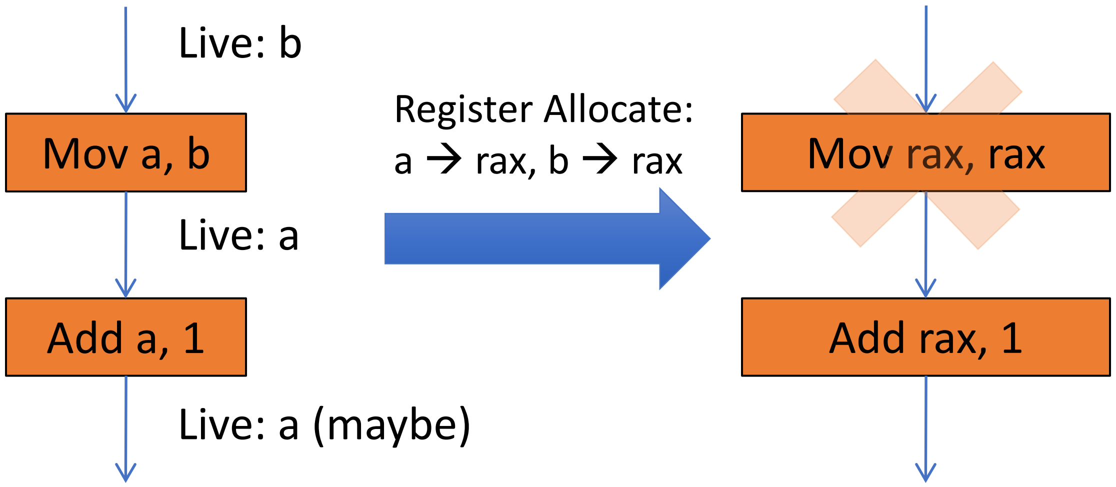
We analyze liveness by looking at CFGs (at different granularities)



Liveness is associated with *edges*



- Example: a = b + 1
- Compiles to:



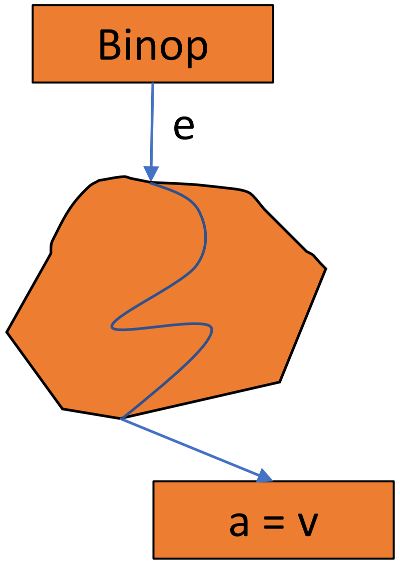
Liveness analysis is based on uses and definitions

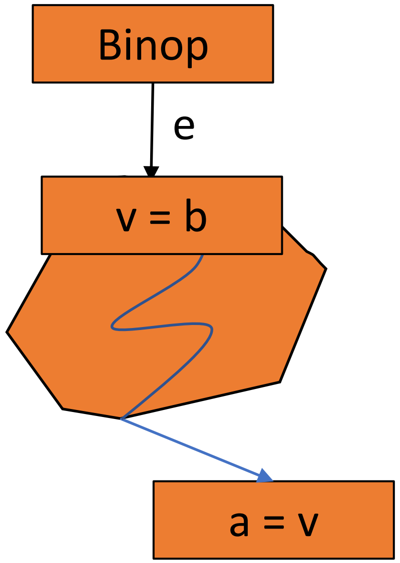
- For a node/statement s define:
  - use[s] : set of variables used (i.e. read) by s
  - def[s] : set of variables defined (i.e. written) by s

- Examples:
  - a = b + c   use[s] = {b,c}   def[s] = {a}
  - a = a + 1   use[s] = {a}   def[s] = {a}

Liveness, formally

- A variable v is *live* on edge e if:
  There is
  - a node n in the CFG such that use[n] contains v, *and*
  - a directed path from e to n such that for every statement s’ on the path, def[s’] does not contain v





A simple inefficient algorithm

- "A variable v is live on an edge e if there is a node n in the CFG using it *and* a directed path from e to n passing through no def of v."

- Algorithm:
  - For each variable v…
  - Try all paths from each use of v, tracing backwards through the control-flow graph until either v is defined or a previously visited node has been reached.
  - Mark the variable v live across each edge traversed.

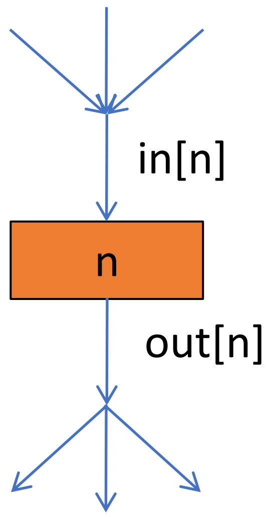
O(number of edges * number of var uses)

Instead, compute liveness info for all variables simultaneously

- Approach: define *equations* that must be satisfied by any liveness determination.
  - Equations based on “obvious” constraints.
- Solve the equations by iteratively converging on a solution.
  - Start with a “rough” approximation to the answer
  - Refine the answer at each iteration
  - Keep going until a *fixed point* has been reached
- This is an instance of a general framework for computing program properties: dataflow analysis

Equations for liveness analysis

- Definitions:
  - use[n] : set of variables used by n
  - def[n] : set of variables defined by n
  - in[n] : set of variables live on entry to n
  - out[n] : set of variables live on exit from n

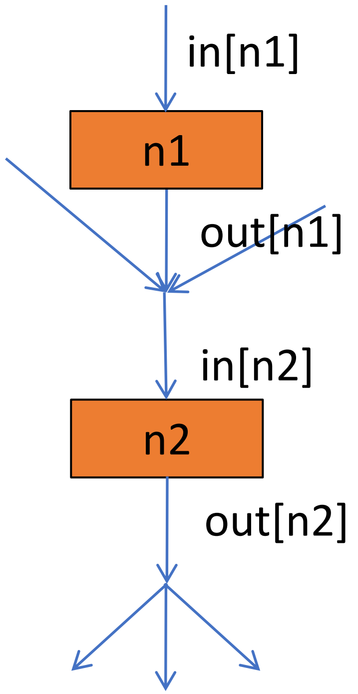


Equations for liveness analysis

- use[n] : set of variables used by n
- def[n] : set of variables defined by n
- in[n] : set of variables live on entry to n
- out[n] : set of variables live on exit from n

- Constraints:
  - in[n] ⊇ use[n]
  - out[n] ⊇ in[n’] if n’ ∈ succ[n]
  - in[n] ⊇ out[n] / def[n]

Propagate
(but not through defs)



Iterative Dataflow Analysis

- Find a solution to those constraints by starting from a rough guess.
  - Start with: $in[n] = \emptyset$ and $out[n] = \emptyset$
- Idea: iteratively re-compute in[n] and out[n] where forced to by the constraints.
  - Each iteration will add variables to the sets in[n] and out[n] (i.e. the live variable sets will increase monotonically)
- We stop when in[n] and out[n] satisfy these equations: (which are derived from the constraints above)
  - $in[n] = use[n] \cup (out[n] / def[n])$
  - $out[n] = \cup_{n' \in succ[n]} in[n']$

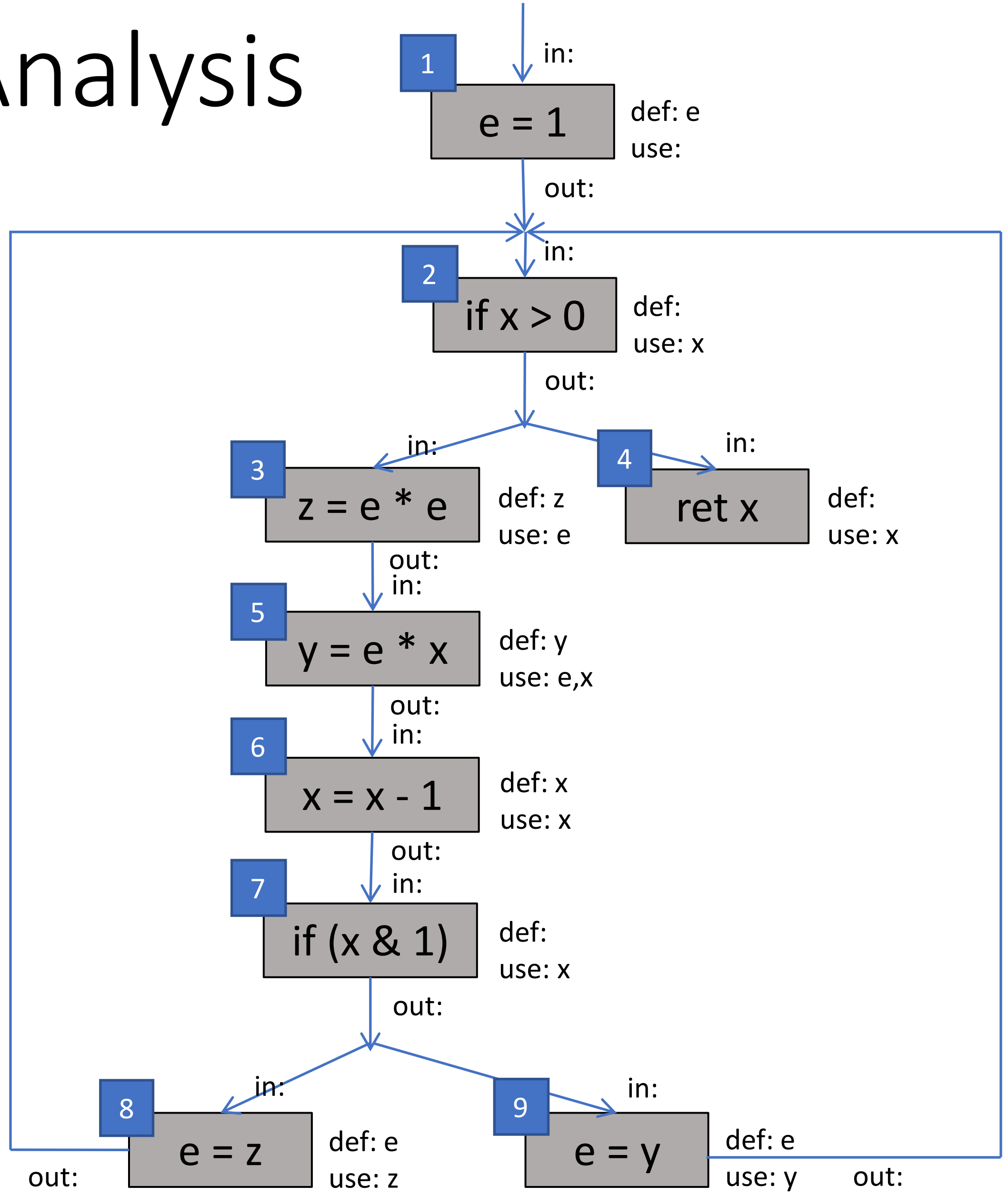
Full Liveness Analysis Algorithm

```
for all n, in[n] := Ø, out[n] := Ø
repeat until no change in 'in' and 'out':
    for all n:
        out[n] := ∪_{n'∈succ[n]} in[n']
        in[n] := use[n] ∪ (out[n] / def[n])
    end
end
```

- Finds a *fixed point* of the in and out equations.
  - The algorithm is guaranteed to terminate… Why?
- Why do we start with Ø?

Example Liveness Analysis

```
e = 1;
while(x>0) {
  z = e * e;
  y = e * x;
  x = x - 1;
  if (x & 1) {
    e = z;
  } else {
    e = y;
  }
}
return x;
```



Example Liveness Analysis

Each iteration update:

$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 1:

in[2] = x

in[3] = e

in[4] = x

in[5] = e,x

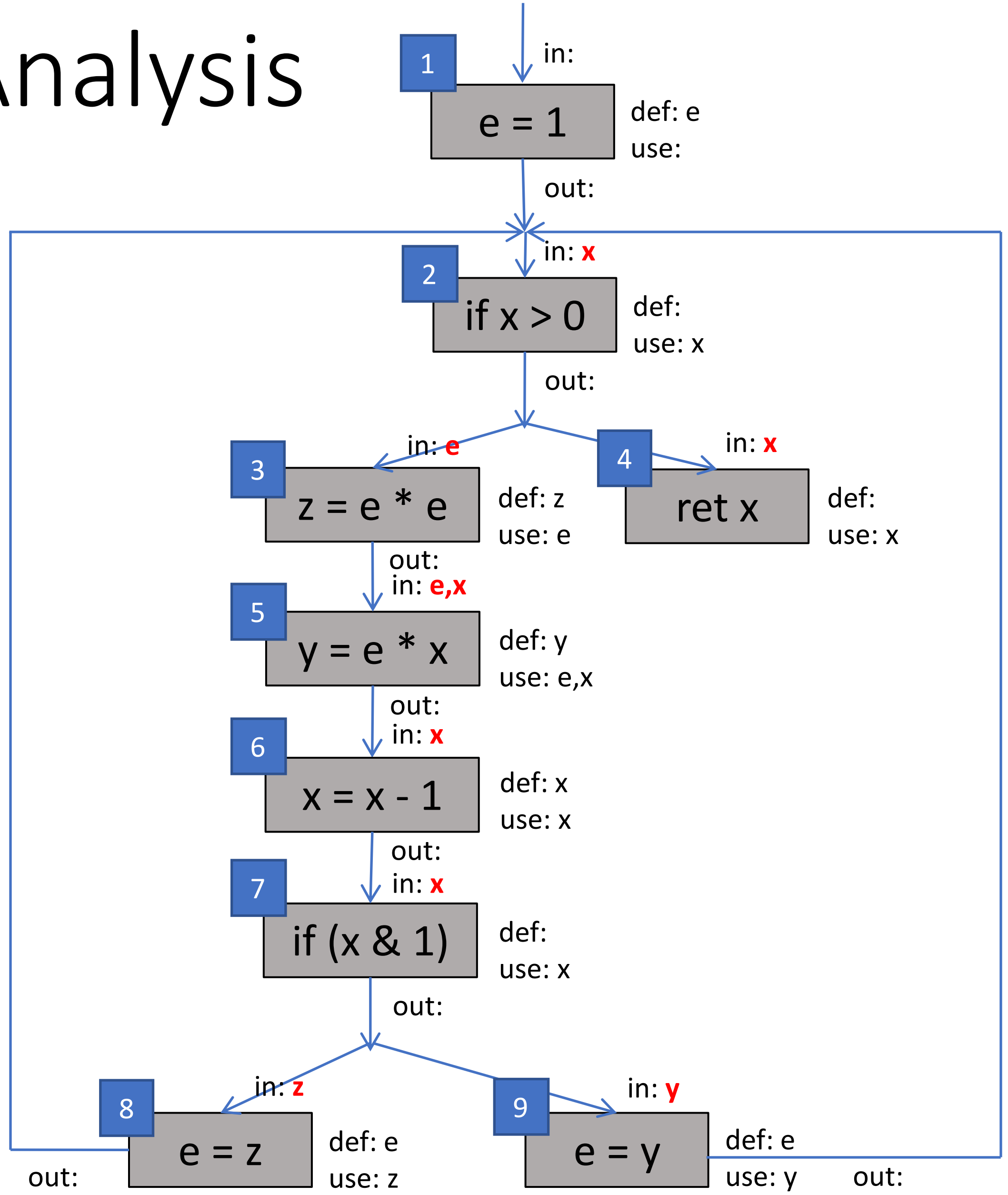
in[6] = x

in[7] = x

in[8] = z

in[9] = y

(showing only updates that make a change)



Example Liveness Analysis

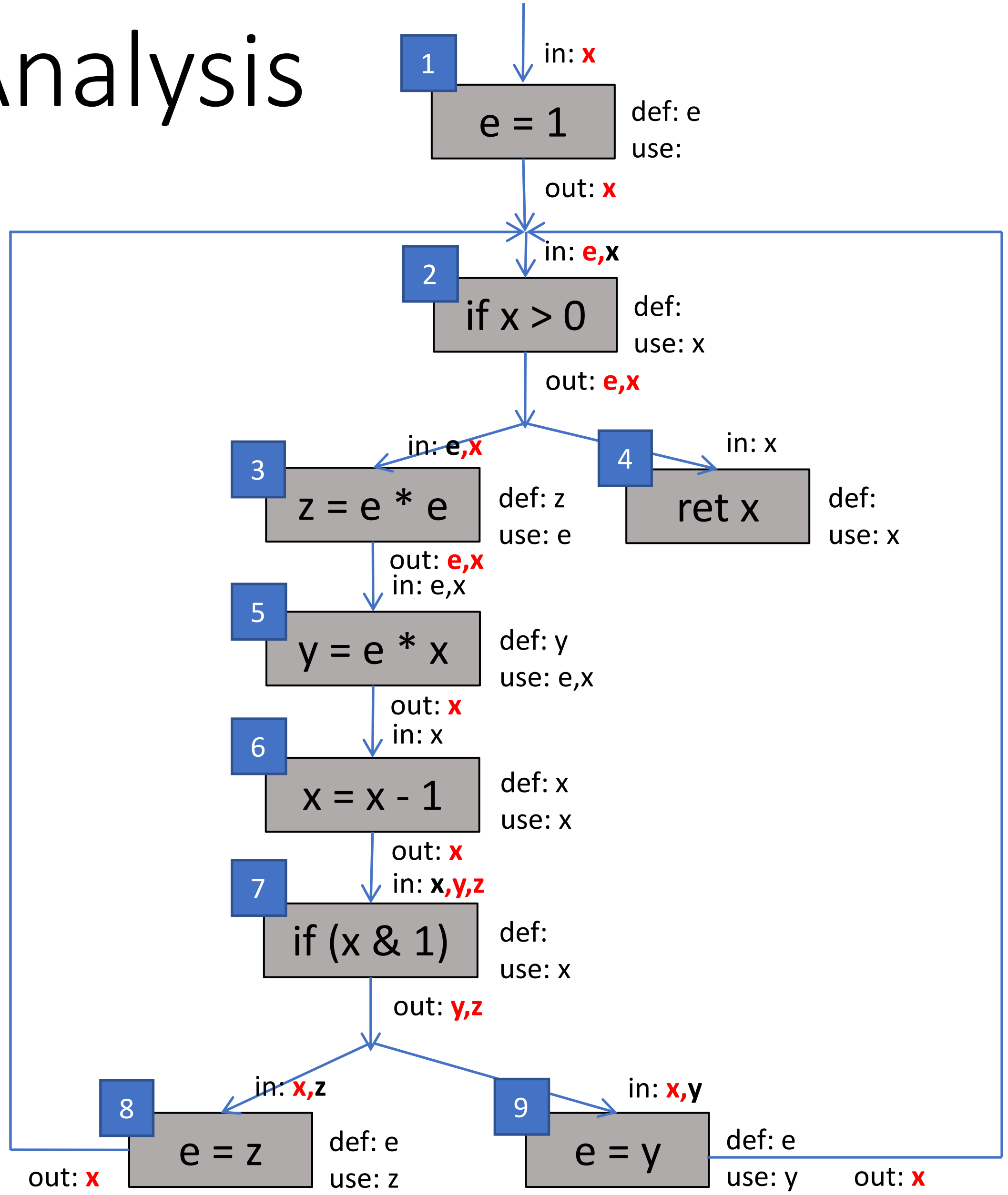
Each iteration update:

$out[n] := \bigcup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 2:

out[1]= x	out[6] = x
in[1] = x	out[7] = z,y
out[2] = e,x	in[7] = x,z,y
in[2] = e,x	out[8] = x
out[3] = e,x	in[8] = x,z
in[3] = e,x	out[9] = x
out[5] = x	in[9] = x,y



Example Liveness Analysis

Each iteration update:

out[n] := $\bigcup_{n' \in succ[n]} in[n']$

in[n] := use[n] ∪ (out[n] - def[n])

- Iteration 3:

out[1]= e,x

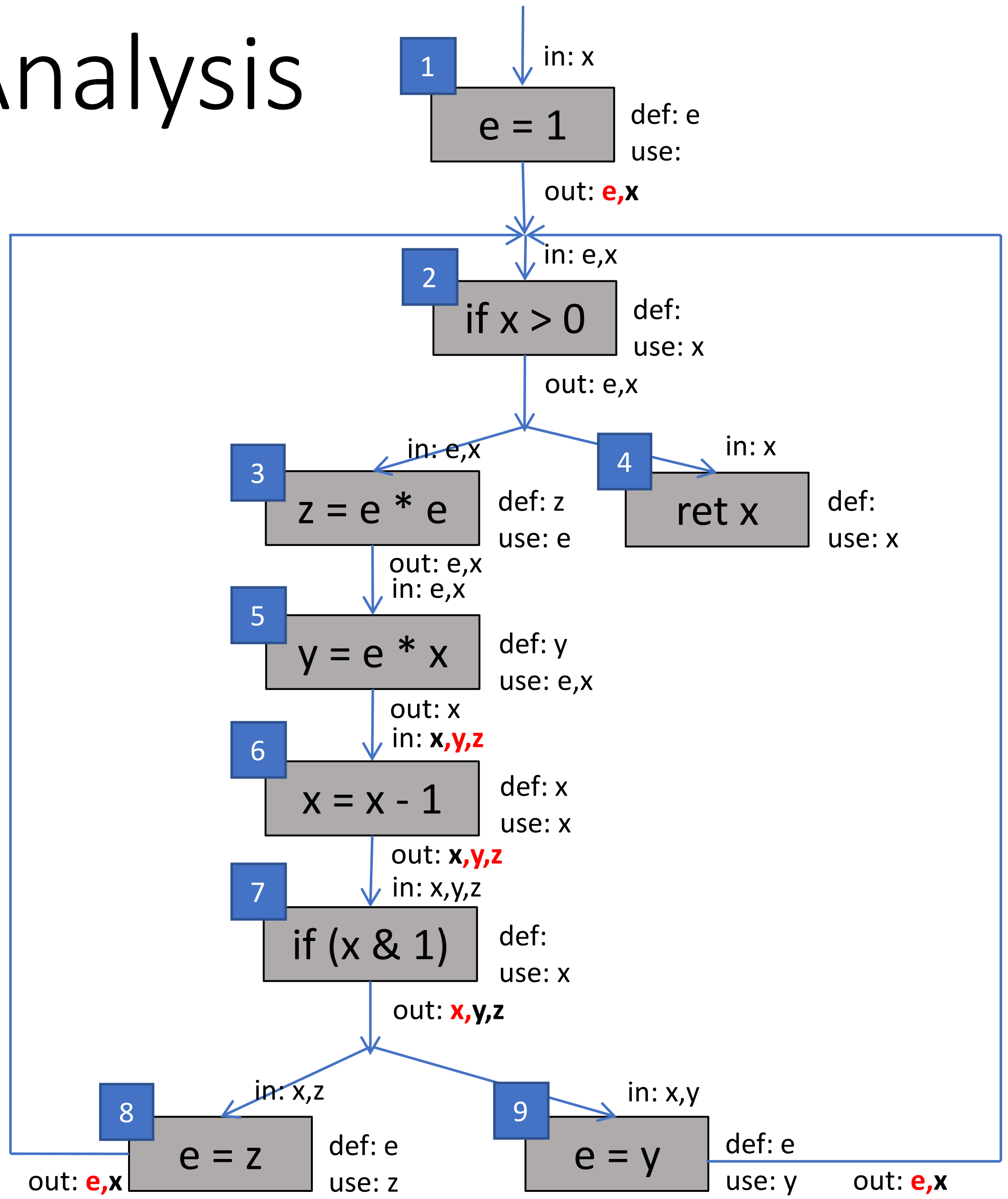
out[6]= x,y,z

in[6]= x,y,z

out[7]= x,y,z

out[8]= e,x

out[9]= e,x



Example Liveness Analysis

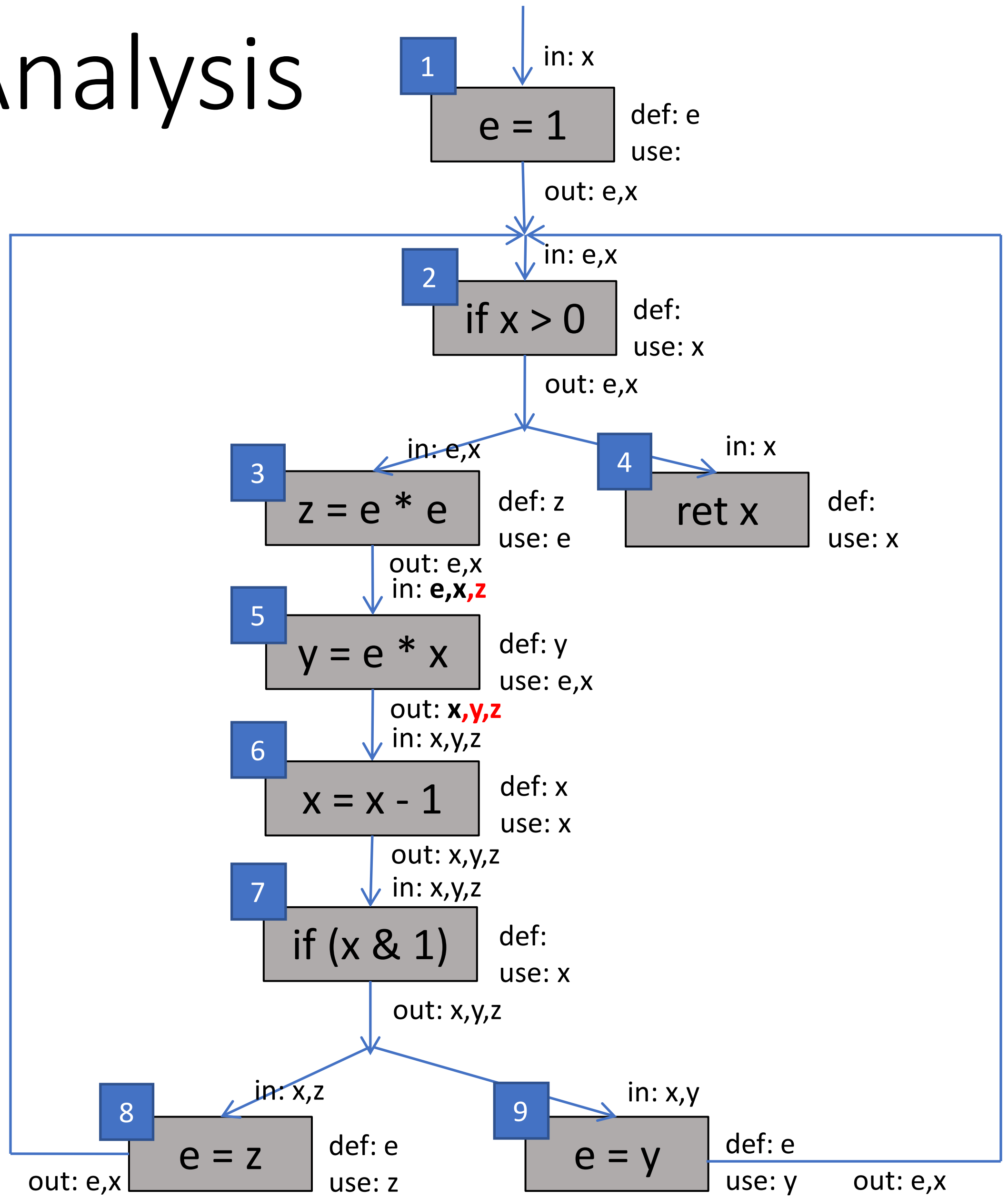
Each iteration update:

out[n] := $\bigcup_{n' \in succ[n]}$ in[n']

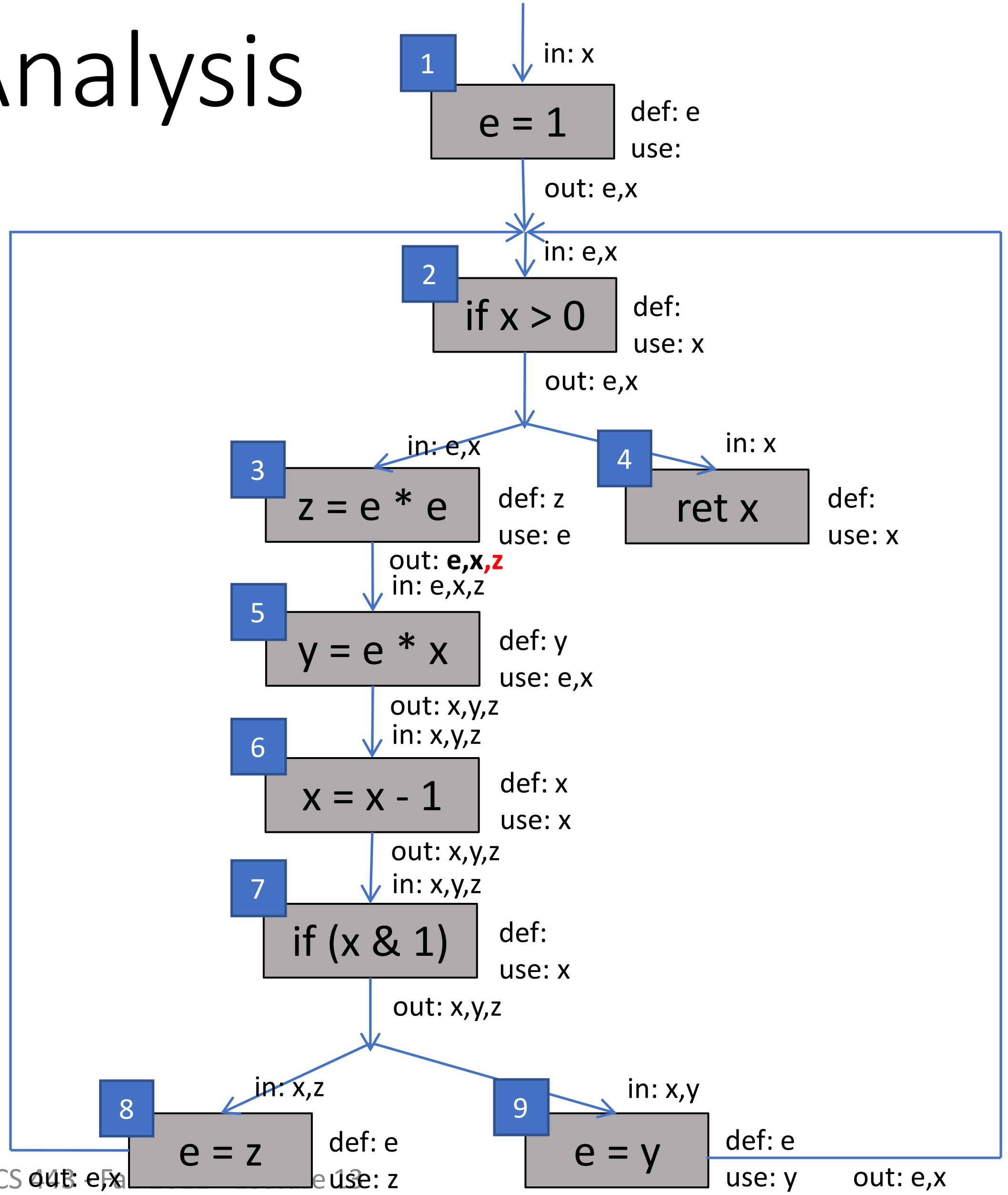
in[n] := use[n] ∪ (out[n] - def[n])

- Iteration 4:

out[5]= x,y,z
in[5]= e,x,z



Example Liveness Analysis



Each iteration update:

out[n] := $\bigcup_{n' \in succ[n]} in[n']$

in[n] := use[n] ∪ (out[n] - def[n])

- Iteration 5:

out[3]= e,x,z

Done!

Improvement: only need to update a node if its successors changed

- Observe: the only way information propagates from one node to another is using: $out[n] := \cup_{n' \in succ[n]} in[n']$
  - This is the only rule that involves more than one node

- Idea for an improved version of the algorithm:
  - Keep track of which node's successors have changed

Worklist algorithm: Use a FIFO queue of nodes that might need to be updated

```
for all n, in[n] := Ø, out[n] := Ø
w = new queue with all nodes
repeat until w is empty:
  let n = w.pop()                              // pull a node off the queue
    old_in = in[n]                             // remember old in[n]
    out[n] := ∪_{n'∈succ[n]} in[n']
    in[n] := use[n] ∪ (out[n] - def[n])
    if (old_in != in[n]):                      // if in[n] has changed
      for all m in pred[n]: w.push(m)          // add pred to worklist
end
```