



《编译原理》

第8讲： 活性分析

何冬杰
重庆大学

Proper Register Allocation

❑ Assign each virtual register to an architectural register (if possible)

❑ Example

```
.data
x : .space 4
y : .space 4
.text
la v0, x
lw v1, v0
add v2, v0, v1
la v3, y
lw v4, v3
sub v5, v4, v2
add v6, v2, v4
sw v5, 0(v0)
sw v6, 0(v3)
```

Using virtual registers

```
.data
x : .space 4
y : .space 4
.text
la t0, x
lw t1, t0
add t2, t0, t1
la t3, y
lw t4, t3
sub t5, t4, t2
add t6, t2, t4
sw t5, 0(t0)
sw t6, 0(t3)
```

Using t0-t6 physical registers

❑ What if more virtual registers are used than the available architectural registers?

● **Solution:** Recycle architectural registers

● => Need to know which values are going to be used in the **future**

Liveness

□ From now on, in this lecture, we will use the term *variable* to denote a *virtual register*

Definition

A variable (virtual register) is **live** at some point in the program if it has previously been **defined** by an instruction and will be **used** by an instruction in the future. It is **dead** otherwise.

□ Example

□ Register allocation uses liveness information

Two variables can share the same register if they are not **live** at the same time.

.text	Live after instruction:
la v0, x	v0
lw v1, (v0)	v0 v1
add v2, v1, v1	v0 v2
la v3, y	v0 v2 v3
lw v4, (v3)	v0 v2 v3 v4
sub v5, v4, v2	v0 v2 v3 v4 v5
add v6, v2, v4	v0 v3 v5 v6
sw v5, (v0)	v3 v6
sw v6, (v3)	

Scope vs. Liveness

❑ We can already get some coarse liveness information from variable scoping

```
int f(int x) {  
    var a = 0;  
    if (x > 0) {  
        var b = x * x; a = b + b;  
    }  
    var c = a * x;  
    return c;  
}
```

Since **b** and **c** have disjoint scopes, they can never be live simultaneously and can share the same physical register.

❑ But scope is too coarse

```
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

- The scopes of **a**, **b**, **c**, and **x** all overlap; they're all in scope at the end of the block.
- But **a**, **b**, and **c** are never live at the same time.
- So, they can share the same stack slot/register

❑ Computing liveness is more complicated in the presence of control flow

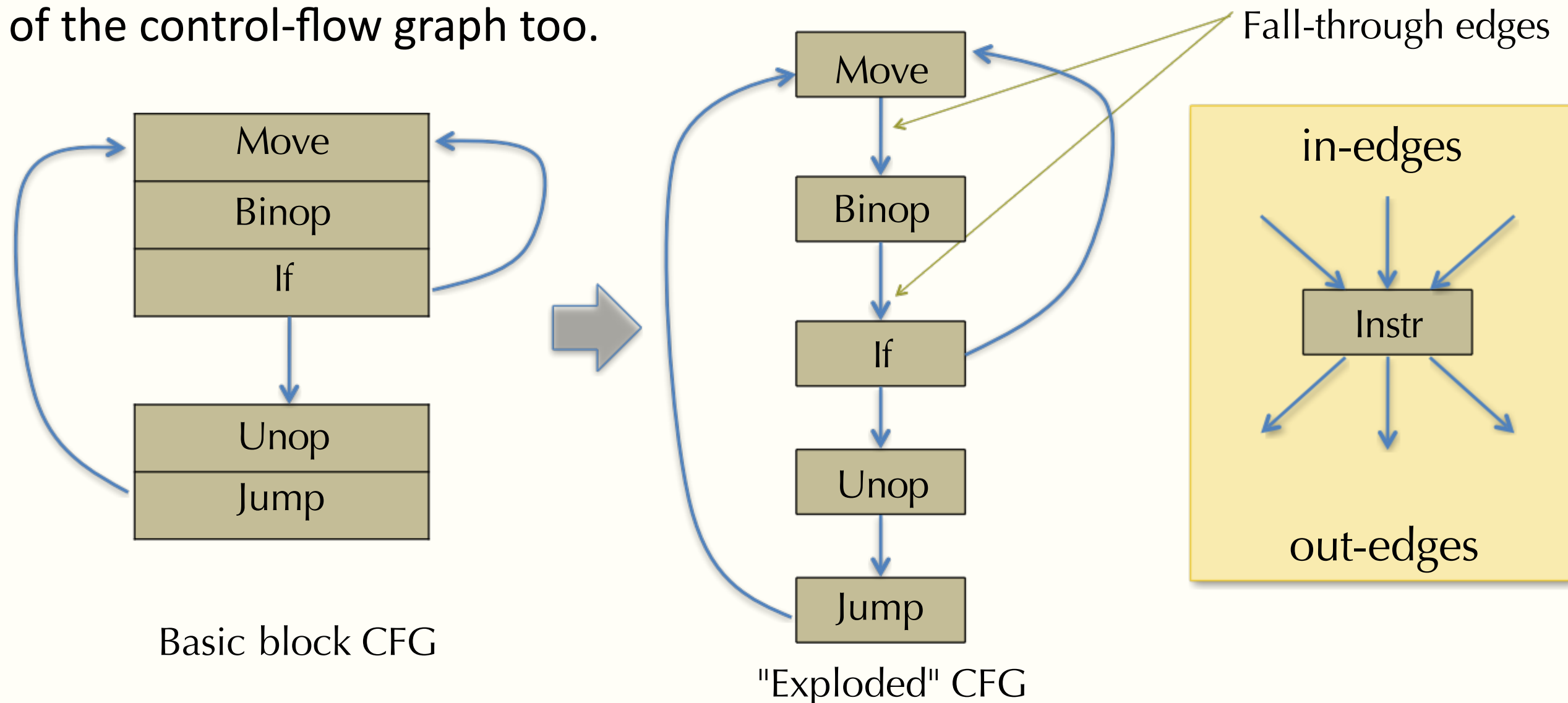
Revisited Control-flow Graphs (CFGs)

```
type block = {  
  insns : (uid * insn) list;  
  term: terminator }  
  
type cfg = block * (lbl * block) list
```

□ A CFG is represented as a list of labelled basic blocks with these invariants:

- No two blocks have the same label, there is a distinguished, unlabeled entry block
- All terminators mention only labels that are defined among the set of basic blocks

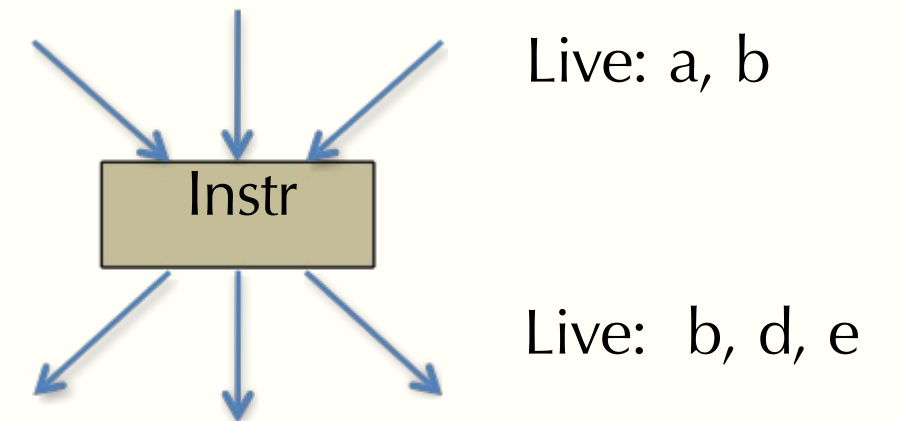
□ For precision, it is helpful to think of the “fall through” between sequential instructions as an edge of the control-flow graph too.



Liveness analysis

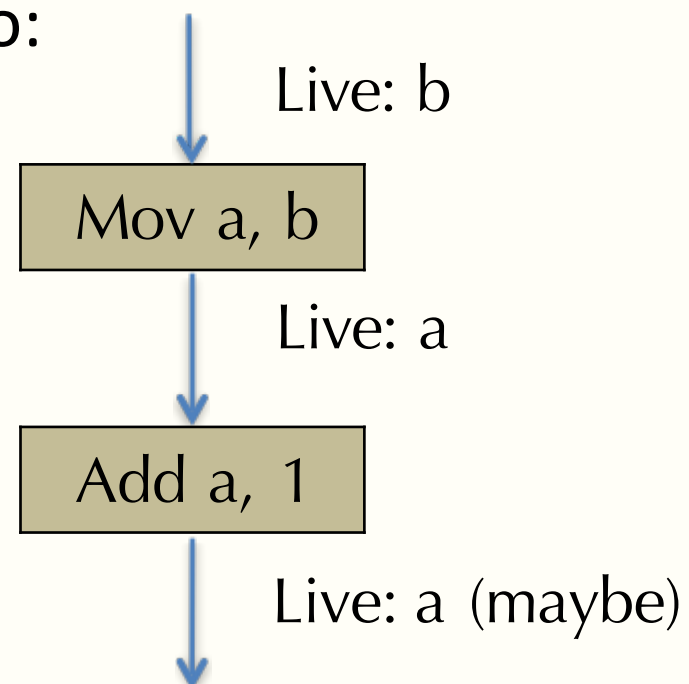
□ Liveness analysis: Compute the live variables between each statement

- Liveness analysis is one example of dataflow analysis
- May be conservative (i.e., may claim a variable is live when it isn't)
 - Safe approximation!
- To be useful, it should be more precise than simple scoping rules.

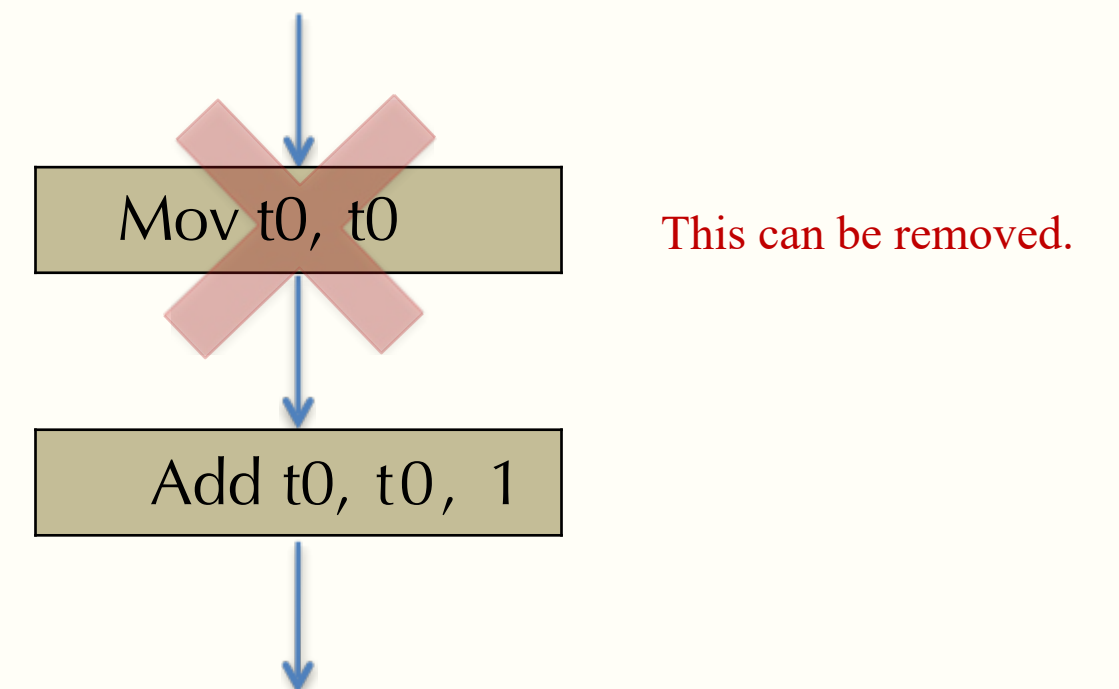


□ Liveness is associated with edges

- This is useful for reusing the same register for different temporaries in one statement.
- Example: $a = b + 1$
- Compiles to:



Register Allocate:
 $a \rightarrow t0, b \rightarrow t0$



Uses and Definitions

- ❑ Every instruction/statement **uses** some set of variables
 - i.e., reads from them
- ❑ Every instruction/statement **defines** some set of variables
 - i.e., writes to them
- ❑ For a CFG node/statement **s**, define:
 - $\text{use}[s]$: set of variables used by s
 - $\text{def}[s]$: set of variables defined by s
- ❑ Examples:
 - $a = b + c$ $\text{use}[s] = \{b, c\}$ $\text{def}[s] = \{a\}$
 - $a = a + 1$ $\text{use}[s] = \{a\}$ $\text{def}[s] = \{a\}$

Liveness, Formally

□ Variable v is live on edge e if:

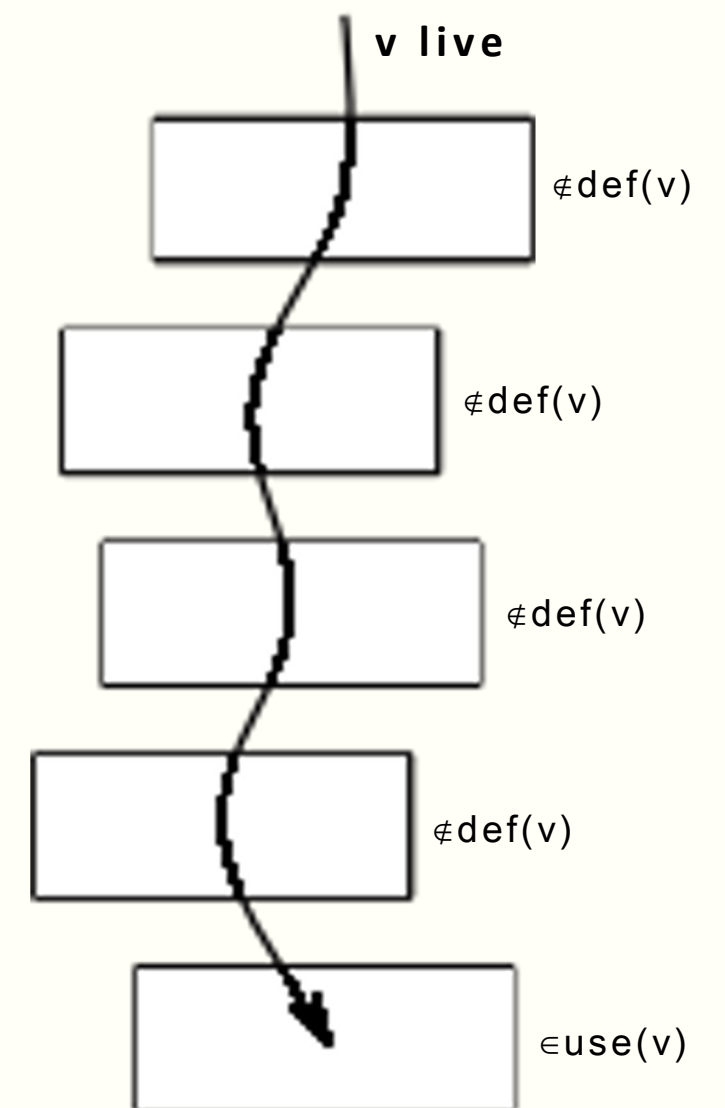
- (1) there is a node n in the CFG such that $\text{use}[n]$ contains v , and
- (2) there is a directed path from e to n such that for every statement s' on the path, $v \notin \text{def}[s']$

□ Clause (1) says that v will be used on some path starting from edge e

□ Clause (2) says that v won't be redefined on that path before the use

□ Questions:

- How to compute this efficiently?
- How to use this information (e.g., for register allocation)?
- How does the choice of IR affect this?
 - LLVM IR uses SSA, so it doesn't allow redefinition, which simplifies the analysis



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 that does not define v ”

□ Backtracking Algorithm:

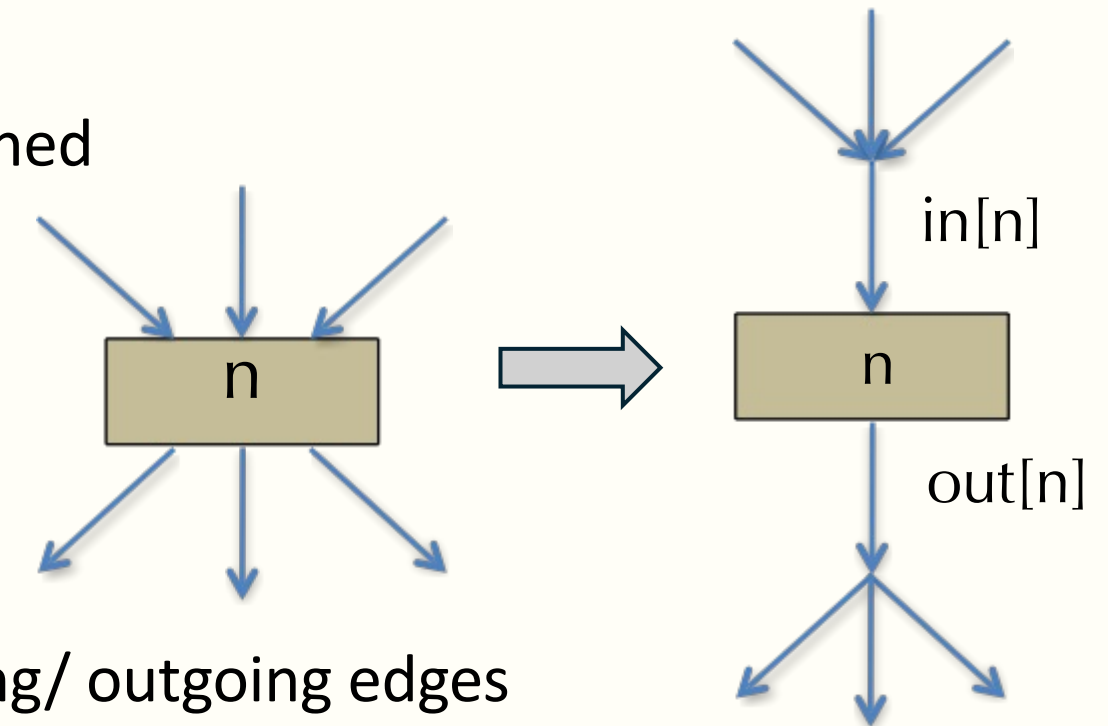
- For each variable v ,
- Try all paths from each use of v , tracing backwards through the control-flow graph until either v is defined or a previously visited node has been reached.
- Mark the variable v live on each edge traversed.

□ **Inefficient** because it explores the same paths many times

- for different uses and different variables

Dataflow Analysis

- ❑ **Idea:** compute liveness information for all variables simultaneously
 - Keep track of sets of information about each node
- ❑ **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 no more refinement is possible: a **fixpoint** has been reached
- ❑ **Dataflow Value Sets for Liveness**
 - Nodes are program statements, so, for each n , define the following sets:
 - **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
 - Associate $\text{in}[n]$ and $\text{out}[n]$ with the “collected” information about incoming/ outgoing edges
 - i.e., $\text{out}[n]$ is the union of all liveness information on outgoing edges of n



Liveness Dataflow Constraints

□ For liveness, what constraints are there among these sets?

- $use[n]$, $def[n]$, $in[n]$, $out[n]$

□ We have: $in[n] \supseteq use[n]$

- “A variable must be live on entry to n if it is used by n ”

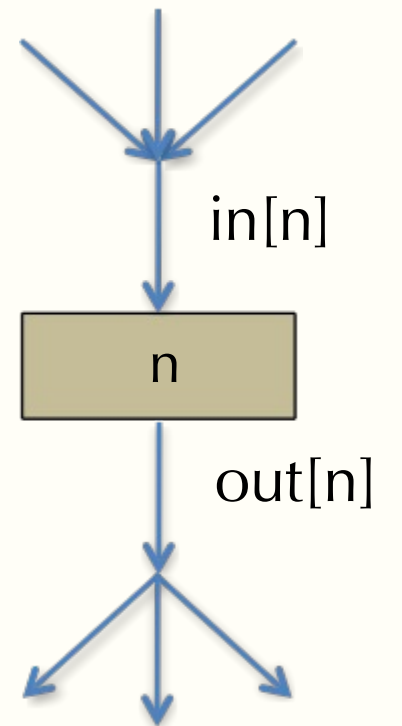
□ Also: $in[n] \supseteq out[n] - def[n]$

- “If a variable is live on exit from n , and n doesn’t define it, then it is live on entry to n ”

- Note: here ‘ $-$ ’ means “set difference”

□ And: $out[n] \supseteq in[n']$ if $n' \in succ[n]$

- “If a variable is live on entry to a successor node of n , it must be live on exit from n .”



Iterative Dataflow Analysis for Liveness Analysis

□ Start with: $\text{in}[n] = \emptyset$ and $\text{out}[n] = \emptyset$

● $\text{in}[n] \supseteq \text{use}[n]; \text{in}[n] \supseteq \text{out}[n] - \text{def}[n]; \text{out}[n] \supseteq \text{in}[n']$ if $n' \in \text{succ}[n]$

□ iteratively re-compute $\text{in}[n]$ and $\text{out}[n]$ where forced to by the constraints

● Each iteration will add variables to the sets $\text{in}[n]$ and $\text{out}[n]$

● (i.e. the live variable sets will increase monotonically)

□ We stop when $\text{in}[n]$ and $\text{out}[n]$ satisfy following equations:

● (which are derived from the constraints above)

$$\begin{aligned}\text{in}[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\ \text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n']\end{aligned}$$

● We reach a **fixed point** of the equations

□ Complete Liveness Analysis Algorithm

● Why do we start with $\emptyset$?

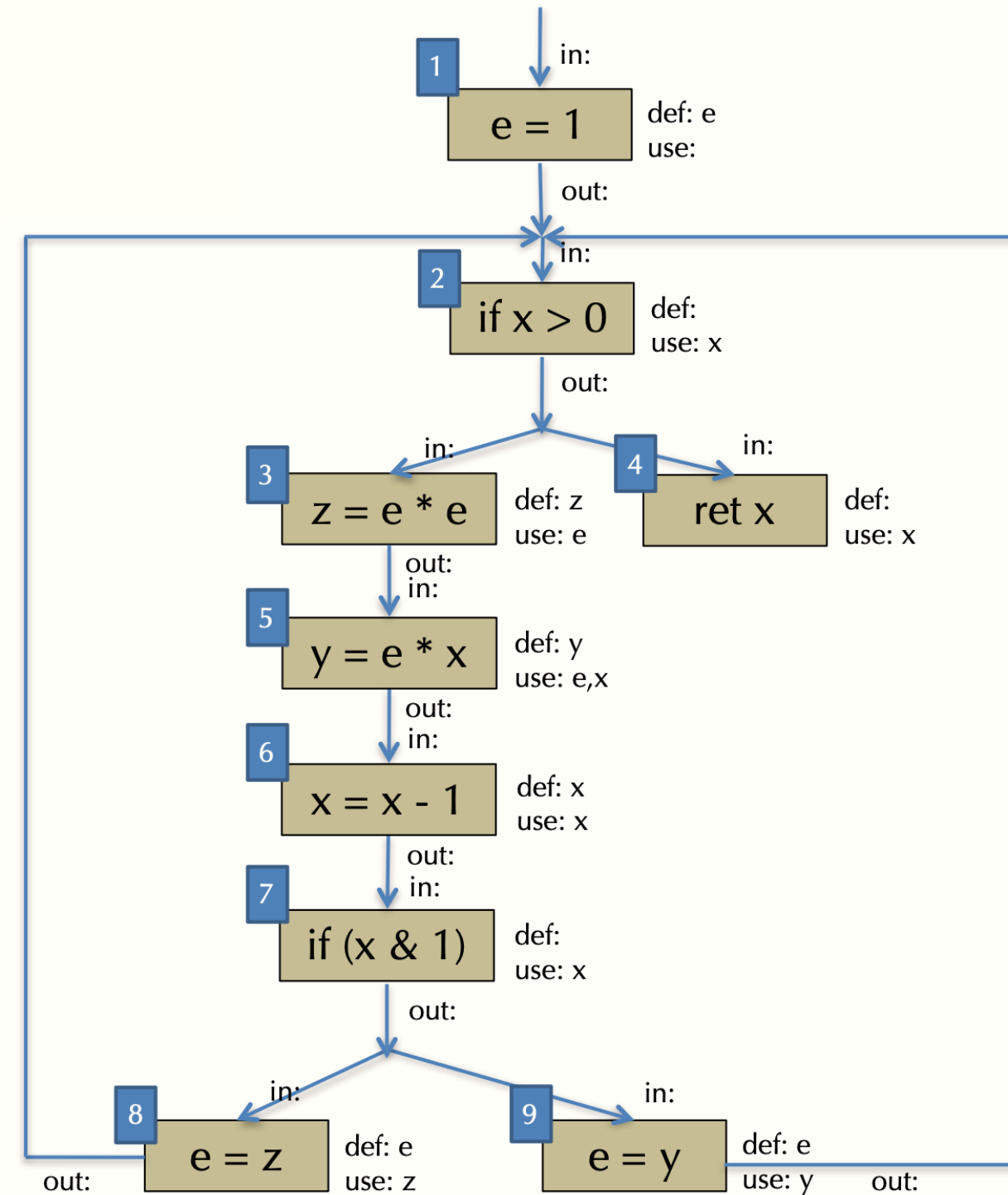
● The algorithm is guaranteed to terminate... Why?

```
for all n, in[n] := g, out[n] := g
repeat until no change in 'in' and 'out'
  for all n
    out[n] :=  $\bigcup_{n' \in \text{succ}[n]} \text{in}[n']$ 
    in[n] :=  $\text{use}[n] \cup (\text{out}[n] - \text{def}[n])$ 
  end
end
```

An Example of Liveness Analysis

□ Example flow graph

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



An Example of Liveness Analysis

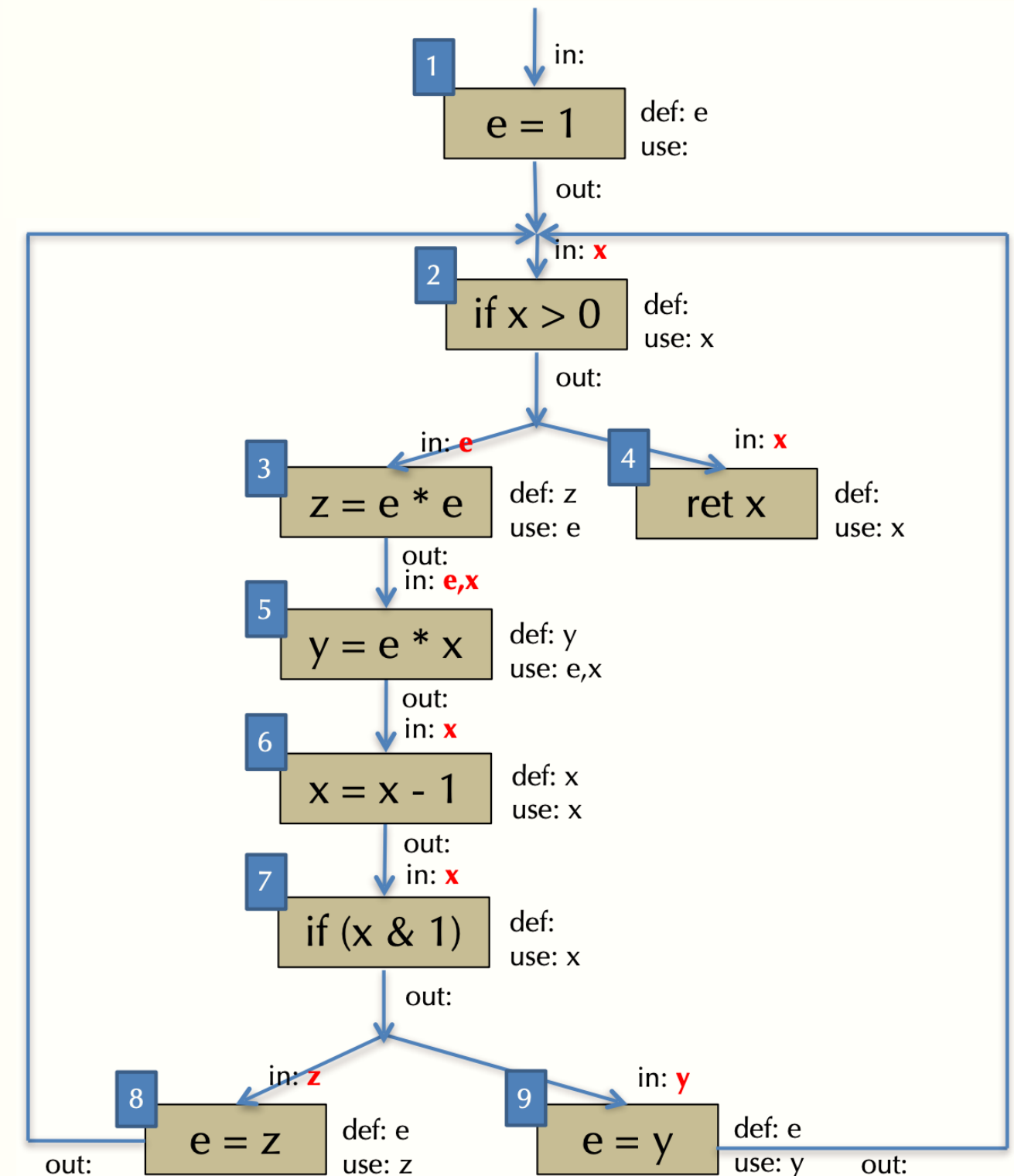
□ Each iteration update:

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\ \text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \end{aligned}$$

□ 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)



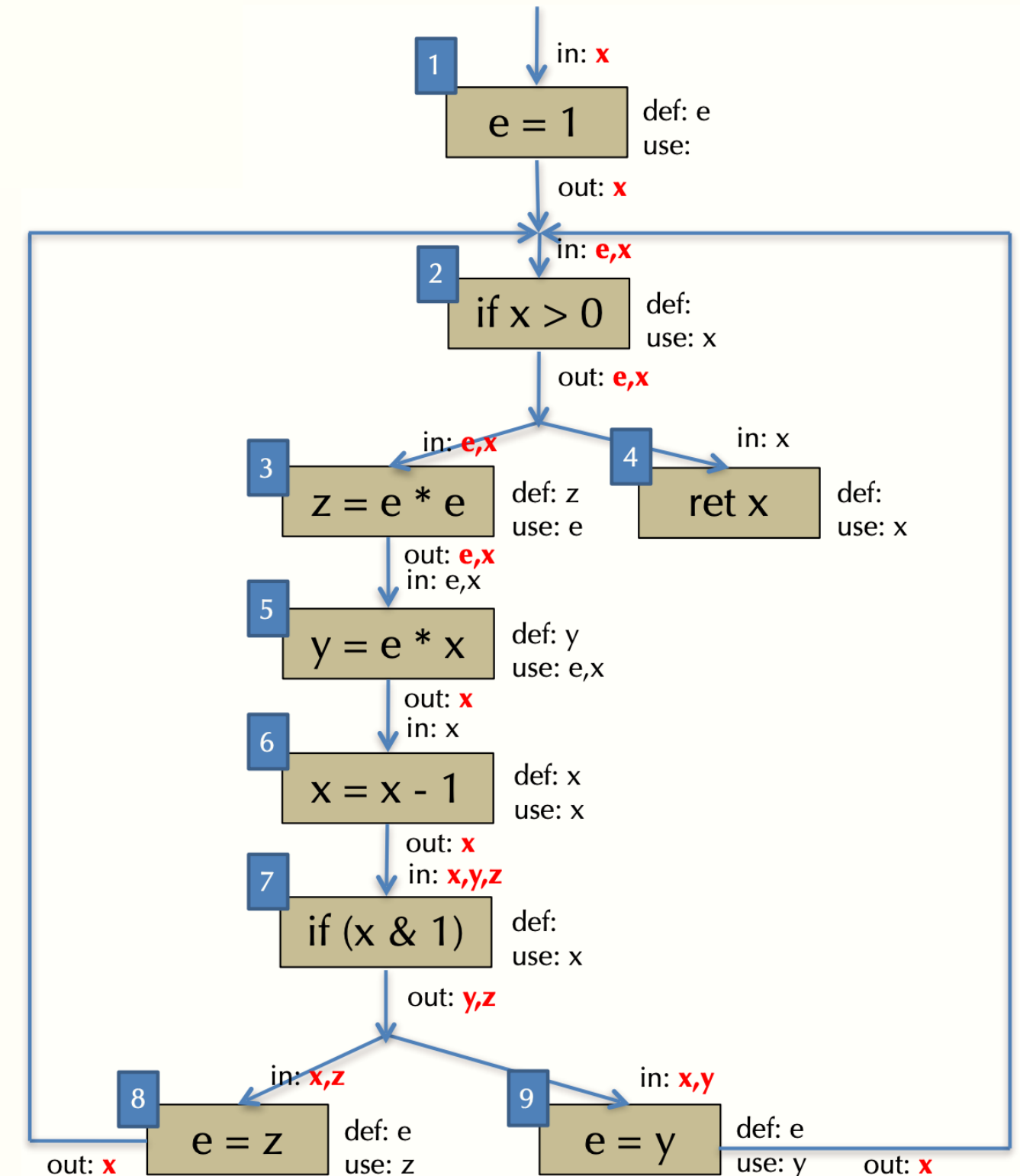
An Example of Liveness Analysis

□ Each iteration update:

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\ \text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \end{aligned}$$

□ Iteration 2:

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



An Example of Liveness Analysis

□ Each iteration update:

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\ \text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \end{aligned}$$

□ Iteration 3:

$\text{out}[1] = e, x$

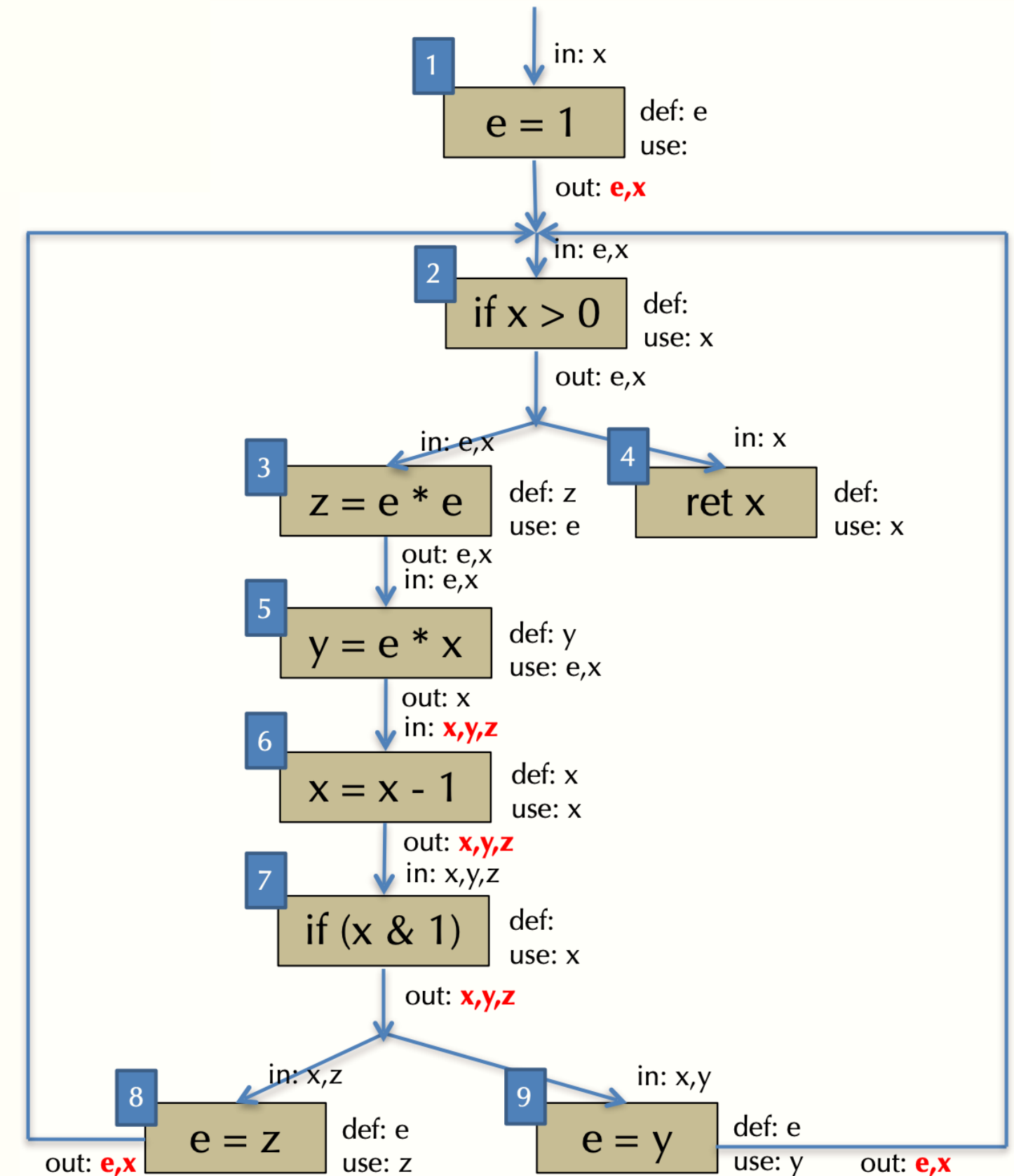
$\text{out}[6] = x, y, z$

$\text{in}[6] = x, y, z$

$\text{out}[7] = x, y, z$

$\text{out}[8] = e, x$

$\text{out}[9] = e, x$



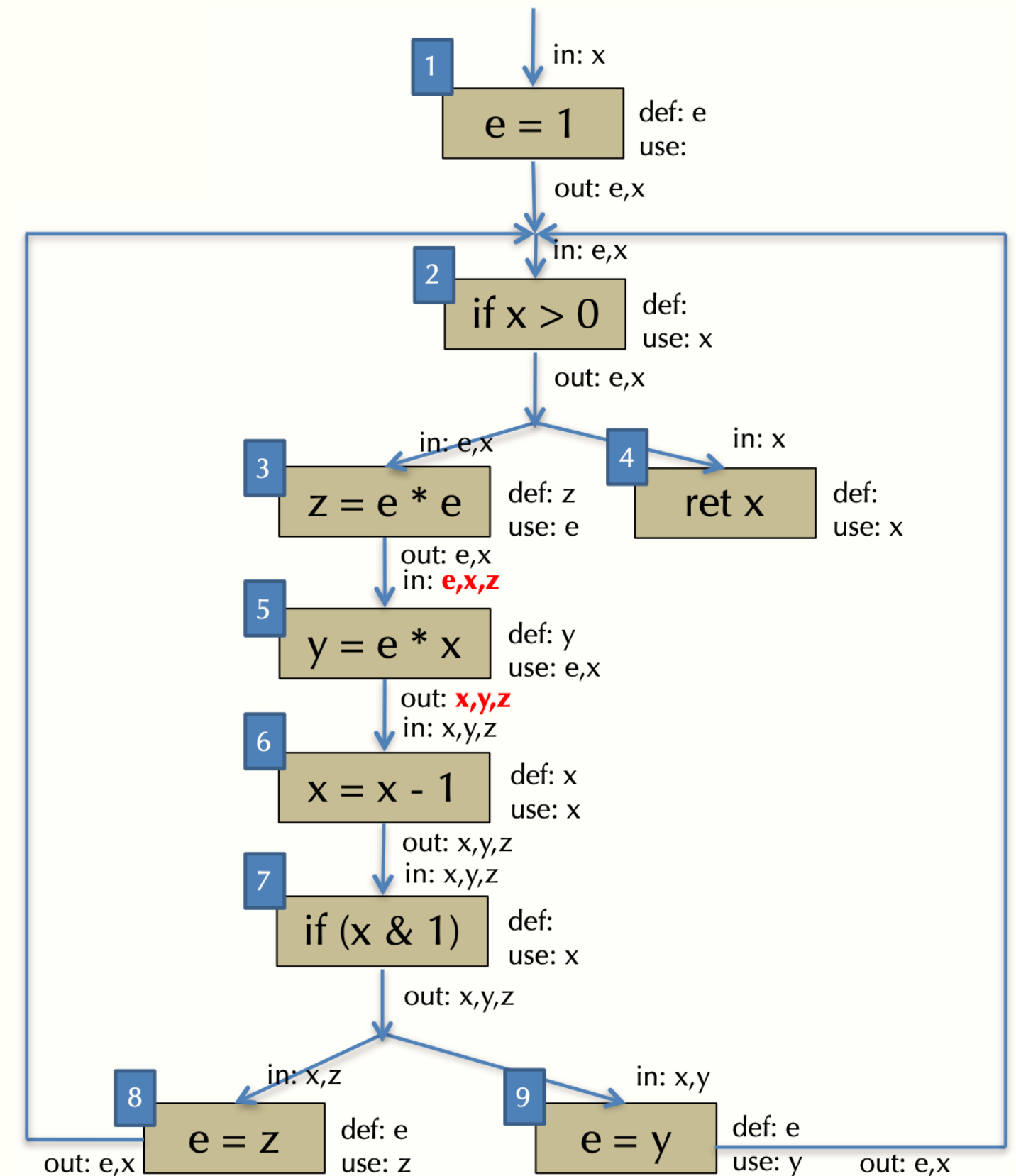
An Example of Liveness Analysis

□ Each iteration update:

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\ \text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \end{aligned}$$

□ Iteration 4:

$\text{out}[5] = x, y, z$
 $\text{in}[5] = e, x, z$



An Example of Liveness Analysis

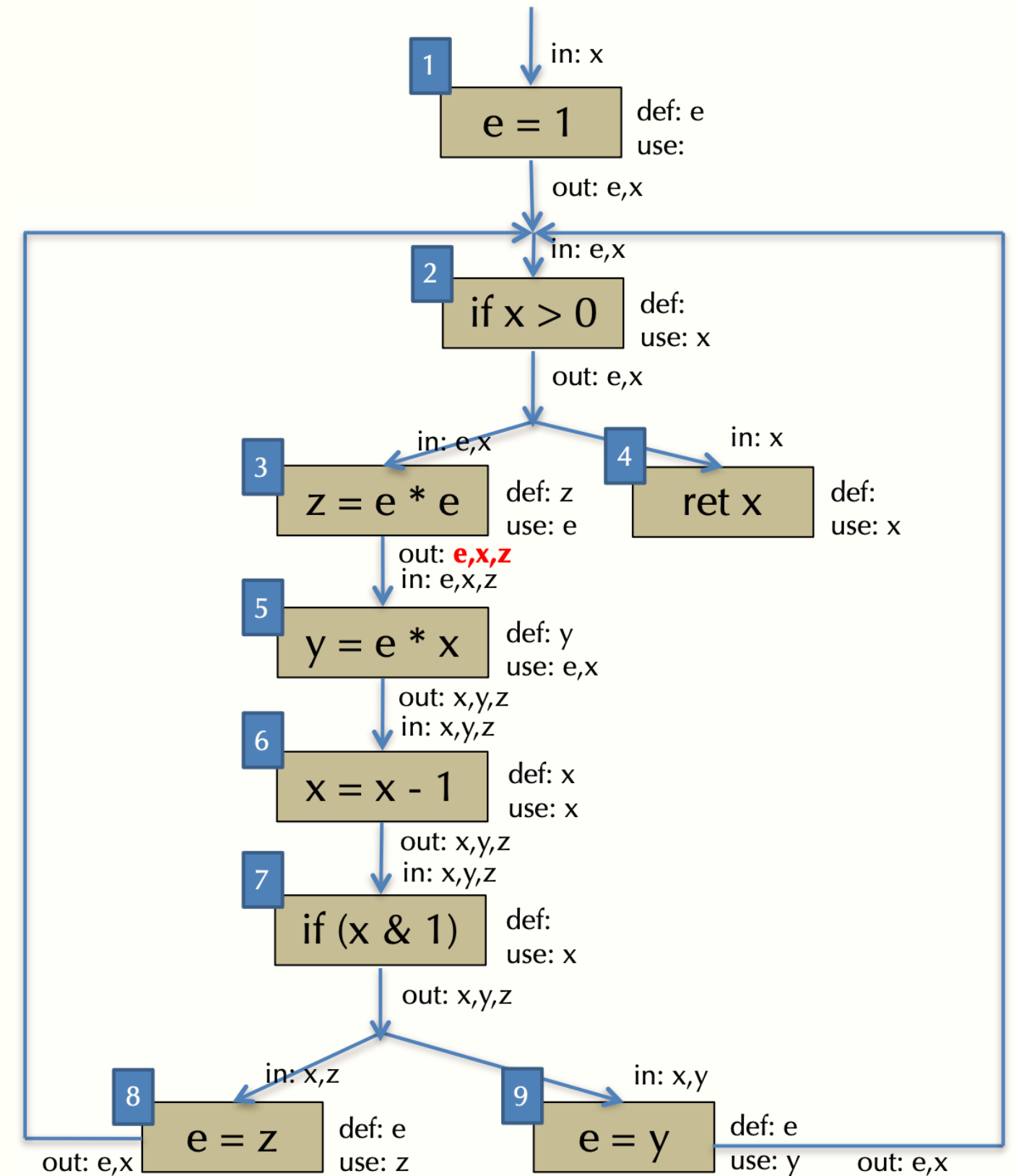
□ Each iteration update:

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\ \text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \end{aligned}$$

□ Iteration 5:

$\text{out}[3] = e, x, z$

Done!



Improving the Algorithm

❑ Can we do better?

❑ Observe: the only way information propagates from one node to another is using:

- $\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
- This is the only rule that involves more than one node
- If the **in** sets of a node's successors haven't changed, then the node itself won't change!

❑ **Idea** for an improved version of the algorithm:

- Keep track of which node's successors have changed

❑ A **worklist** algorithm:

```
for all n, in[n] :=  $\emptyset$ , out[n] :=  $\emptyset$ 
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] :=  $\bigcup_{n' \in \text{succ}[n]} \text{in}[n']$ 
    in[n] := use[n]  $\cup$  (out[n] – def[n])
    if (old_in  $\neq$  in[n]),                          // if in[n] has changed
        for all m in pred[n], w.push(m)             // add to worklist
end
```

❑ More about DFA:

- <https://plc-cqu.github.io/teaching/SA/Spring2026/06DFA.pdf>

作业：基于数据流分析计算变量liveness

□练习1: 利用数据流分析，计算右侧CFG中每个变量的live range

- Live range: 可以是CFG边的集合，表示变量在这些边上是live的
- 需要给出每轮迭代，每个节点的In和Out值

□练习2: 实现GraphColoring.ml中基于worklist的活性分析算法

- 基于图染色的寄存器分配算法，一会介绍！

