Lecture 22

CIS 341: COMPILERS

Announcements / Plan

- HW5: OAT – typechecking, structs, function pointers
  - ***Due: TONIGHT!***

- HW6: LLVM Optimization: analysis and register allocation
  - Available soon
  - Due: Wednesday, April 26

- FINAL EXAM: Thursday, May 4th noon – 2:00p.m.

When to apply optimization

High level	AST	• Inlining • Function specialization • Constant folding
	IR	• Constant propagation • Value numbering • Dead code elimination
Mid level	Canonical IR	• Loop-invariant code motion • Common sub-expression elimination • Strength Reduction • Constant folding & propagation
Low level	Abstract assembly	• Branch prediction / optimization • Register allocation
	Assembly	• Loop unrolling • Cache optimization

Constant Propagation

- If the value is known to be a constant, replace the use of the variable by the constant
- Value of the variable must be propagated forward from the point of assignment
  - This is a substitution operation

- Example:

```
int x = 5;
int y = x * 2; ➔ int y = 5 * 2; ➔ int y = 10;  ➔
int z = a[y];     int z = a[y];     int z = a[y];  int z = a[10];
```

- To be most effective, constant propagation should be interleaved with constant folding

Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

- Example:

```
x = y;
if (x > 1) {
  x = x * f(x – 1);
}
```

➔

~~`x = y;`~~

```
if (y > 1) {
  x = y * f(y – 1);
}
```

- Can make the first assignment to x *dead* code (that can be eliminated).

Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x  = y * y  // x is dead!
…           // x never used  ➔       …
x = z * z                            x = z * z
```

- A variable is *dead* if it is never used after it is defined.
  - Computing such *definition* and *use* information is an important component of compiler

- Dead variables can be created by other optimizations…

Unreachable/Dead Code

- Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and can be deleted.
  - Performed at the IR or assembly level
  - Improves cache, TLB performance

- Dead code: similar to unreachable blocks.
  - A value might be computed but never subsequently used.
- Code for computing the value can be dropped
- But only if it's *pure*, i.e. it has *no externally visible side effects*
  - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  - Note: Pure functional languages (e.g. Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

Inlining

- Replace a call to a function with the body of the function itself with arguments rewritten to be local variables:
- Example in OAT code:

```
int g(int x) { return x + pow(x); }
int pow(int a) { var b = 1; var n = 0;
   while (n < a) {b = 2 * b}; return b; }
```

➔

```
int g(int x) { var a = x; var b = 1; var n = 0;
    while (n < a) {b = 2 * b}; var tmp = b;
    return x + tmp;
}
```

- May need to rename variable names to avoid *name capture*
  - Example of what can go wrong?
- Best done at the AST or relatively high-level IR.
- When is it profitable?
  - Eliminates the stack manipulation, jump, etc.
  - Can increase code size.
  - Enables further optimizations

Code Specialization

- Idea: create specialized versions of a function that is called from different places with different arguments.
- Example: specialize function `f` in:

```
class A implements I { int m() {…} }
class B implements I { int m() {…} }
int f(I x) { x.m(); }        // don't know which m
A a = new A(); f(a);         // know it's A.m
B b = new B(); f(b);         // know it's B.m
```

- `f_A` would have code specialized to dispatch to `A.m`
- `f_B` would have code specialized to dispatch to `B.m`
- You can also inline methods when the run-time type is known statically
  - Often just one class implements a method.

Common Subexpression Elimination

- In some sense it's the opposite of inlining: fold redundant computations together
- Example:

`a[i] = a[i] + 1` compiles to:

```
[a + i*4] = [a + i*4] + 1
```

Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*4; [t] = [t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!

Unsafe Common Subexpression Elimination

- Example: consider this OAT function:

```
unit f(int[] a, int[] b, int[] c) {
  var j = …; var i = …; var k = …;
  b[j] = a[i] + 1; c[k] = a[i]; return;
}
```

- The following optimization that shares the expression `a[i]` is unsafe… why?

```
unit f(int[] a, int[] b, int[] c) {
  var j = …; var i = …; var k = …;
  t = a[i];
  b[j] = t + 1; c[k] = t; return;
}
```

LOOP OPTIMIZATIONS

Loop Optimizations

- Program hot spots often occur in loops.
  - Especially inner loops
  - Not always: consider operating systems code or compilers vs. a computer game or word processor

- Most program execution time occurs in loops.
  - The 90/10 rule of thumb holds here too. (90% of the execution time is spent in 10% of the code)

- Loop optimizations are very important, effective, and numerous
  - Also, concentrating effort to improve loop body code is usually a win

Loop Invariant Code Motion

- Another form of redundancy elimination.
- If the result of a statement or expression does not change during the loop *and* it's pure, it can be hoisted outside the loop body.
- Often useful for array element addressing code
  - Invariant code not visible at the source level

```
for (i = 0; i < a.length; i++) {
    /* a not modified in the body */
}
```

```
t = a.length;
for (i =0; i < t; i++) {
  /* same body as above */
}
```

Hoisted loop-invariant expression

Strength Reduction (revisited)

- Strength reduction can work for loops too
- Idea: replace expensive operations (multiplies, divides) by cheap ones (adds and subtracts)
- For loops, create a *dependent induction variable*:

- Example:

```
for (int i = 0; i<n; i++) { a[i*3] = 1; }
  // stride by 3
```

```
int j = 0;
for (int i = 0; i<n; i++) {
  a[j] = 1;
  j = j + 3;    // replace multiply by add
}
```

Loop Unrolling (revisited)

- Branches can be expensive, unroll loops to avoid them.

```
for (int i=0; i<n; i++) { S }
```

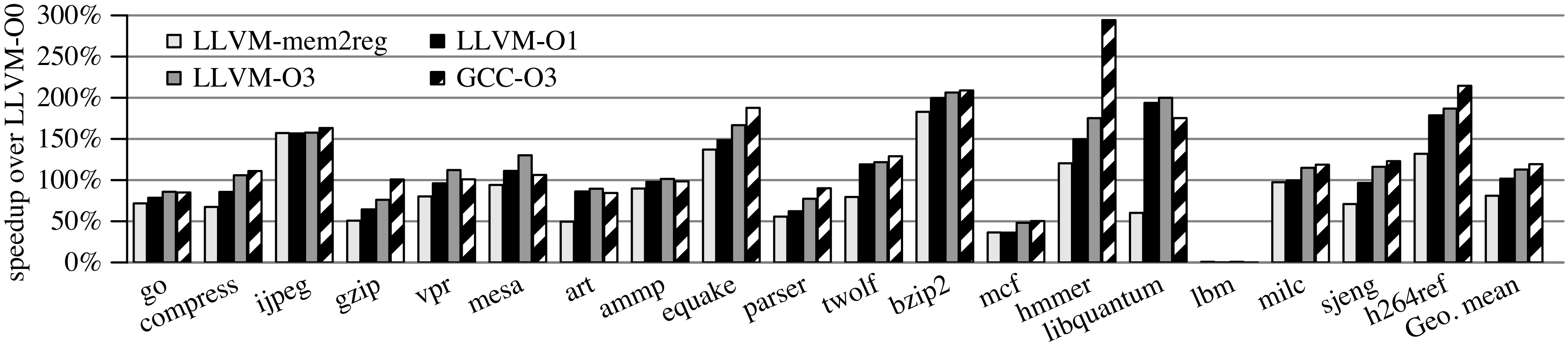
↓

```
for (int i=0; i<n-3; i+=4) {S;S;S;S};
for (         ; i<n; i++) { S } // left over iterations
```

- With k unrollings, eliminates (k-1)/k conditional branches
  - So for the above program, it eliminates ¾ of the branches
- Space-time tradeoff:
  - Not a good idea for large S or small n
- Interacts with instruction caching, branch prediction

EFFECTIVENESS?

Optimization Effectiveness?



$$\%\text{speedup} = \left[\frac{\text{base time}}{\text{optimized time}} - 1 \right] \times 100\%$$

Example:

base time = 2s

optimized time = 1s ⇒ 100% speedup

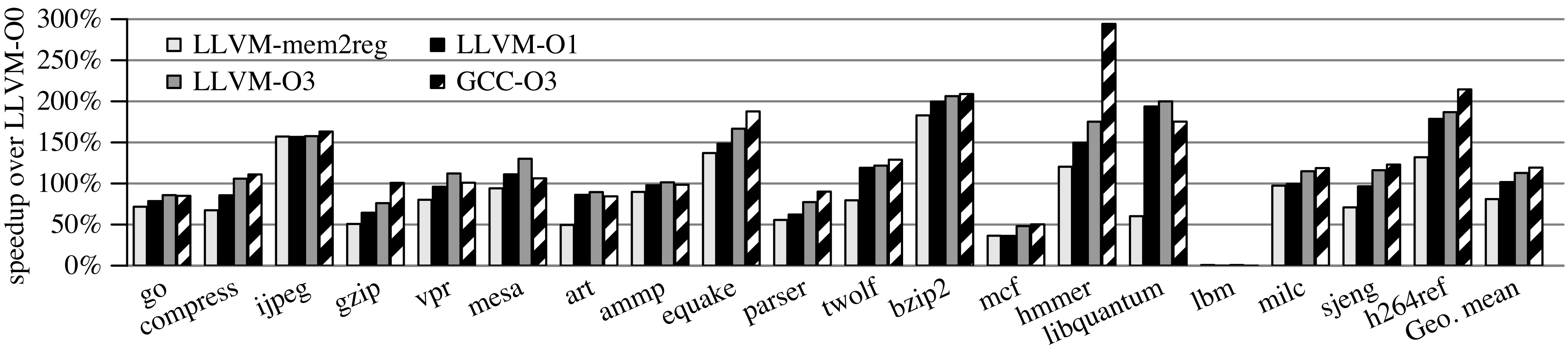
Example:

base time = 1.2s

optimized time = 0.87s ⇒ 38% speedup

Graph taken from:
Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic.
Formal Verification of SSA-Based Optimizations for LLVM.
In Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), 2013

Optimization Effectiveness?



- mem2reg: promotes alloca'ed stack slots to temporaries to enable register allocation
- Analysis:
  - mem2reg alone (+ back-end optimizations like register allocation) yields ~78% speedup on average
  - -O1 yields ~100% speedup
    (so all the rest of the optimizations combined account for ~22%)
  - -O3 yields ~120% speedup
- Hypothetical program that takes 10 sec. (base time):
  - Mem2reg alone: expect ~5.6 sec
  - -O1: expect ~5 sec
  - -O3: expect ~4.5 sec

CODE ANALYSIS

Motivating Code Analyses

- There are lots of things that might influence the safety/applicability of an optimization
  - What algorithms and data structures can help?

- How do you know what is a loop?
- How do you know an expression is invariant?
- How do you know if an expression has no side effects?
- How do you keep track of where a variable is defined?
- How do you know where a variable is used?
- How do you know if two reference values may be aliases of one another?

Moving Towards Register Allocation

- The OAT compiler currently generates as *many* temporary variables as it needs
  - These are the `%uids` you should be very familiar with by now.

- Current compilation strategy:
  - Each %`uid` maps to a stack location.
  - This yields programs with many loads/stores to memory.
  - Very inefficient.

- Ideally, we'd like to map as many `%uid`'s as possible into registers.
  - Eliminate the use of the `alloca` instruction?
  - Only 16 max registers available on 64-bit X86
  - %rsp and %rbp are reserved and some have special semantics, so really only 10 or 12 available
  - This means that a register must hold more than one slot
- When is this safe?

Liveness

- Observation: `%uid1` and `%uid2` can be assigned to the same register if their values will not be needed at the same time.
  - What does it mean for an `%uid` to be "needed"?
  - Ans: its contents will be used as a source operand in a later instruction.
- Such a variable is called "*live*"
- Two variables can share the same register if they are *not* live at the same time.

Scope vs. Liveness

- We can already get some coarse liveness information from variable scoping.
- Consider the following OAT program:

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

- Note that due to OAT's scoping rules, variables `b` and `c` can never be live at the same time.
  - `c`'s scope is disjoint from `b`'s scope
- So, we could assign `b` and `c` to the same alloca'ed slot and potentially to the same register.

But Scope is too Coarse

- Consider this program:

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

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

Live Variable Analysis

- A variable v is *live* at a program point if v is defined before the program point and used after it.
- Liveness is defined in terms of where variables are *defined* and where variables are *used*

- Liveness analysis: Compute the live variables between each statement.
  - May be *conservative* (i.e. it may claim a variable is live when it isn't) so because that's a safe approximation
  - To be useful, it should be more *precise* than simple scoping rules.

- Liveness analysis is one example of *dataflow analysis*
  - Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, …

Control-flow Graphs Revisited

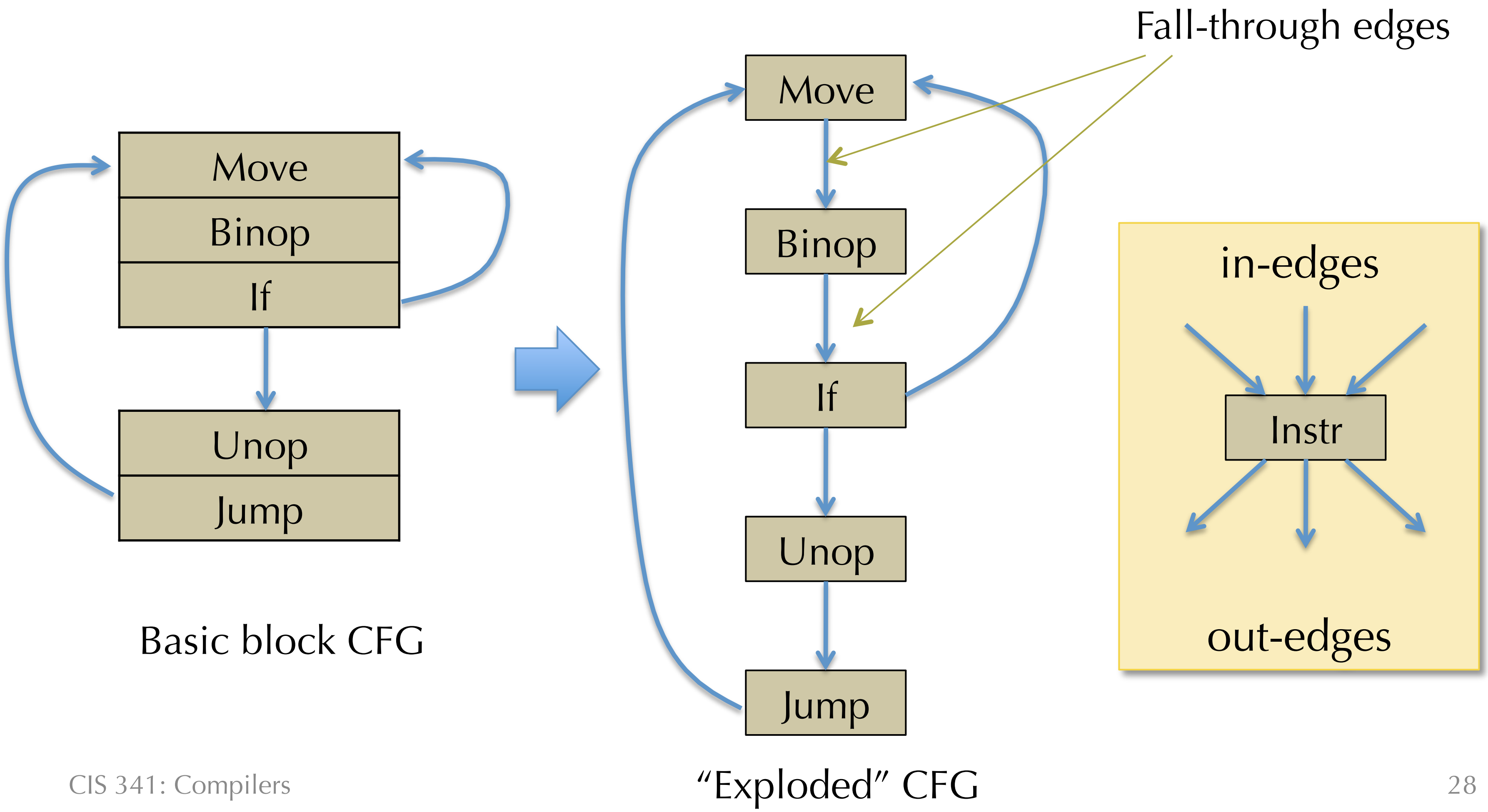
- For the purposes of dataflow analysis, we use the *control-flow graph* (CFG) intermediate form.
- Recall that a basic block is a sequence of instructions such that:
  - There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
  - There is a (possibly empty) sequence of non-control-flow instructions
  - The block ends with a single control-flow instruction (jump, conditional branch, return, etc.)

- A *control flow graph*
  - Nodes are blocks
  - There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  - There are no "dangling" edges – there is a block for every jump target.

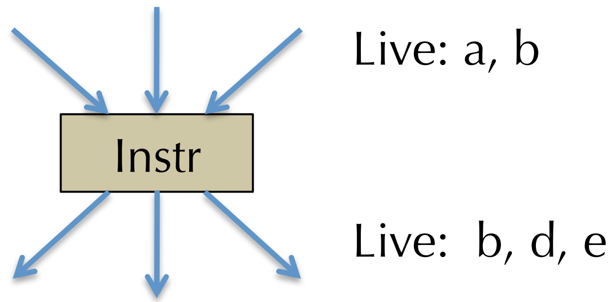
- Note: the following slides are intentionally a bit ambiguous about the exact nature of the code in the control flow graphs:
  - at the x86 assembly level
  - an "imperative" C-like source level
  - at the LLVM IR level
  - Same general idea, but the exact details will differ
    - e.g. LLVM IR doesn't have "imperative" update of %uid temporaries.
    - In fact, the SSA structure of the LLVM IR makes some of these analyses simpler.

Dataflow over CFGs

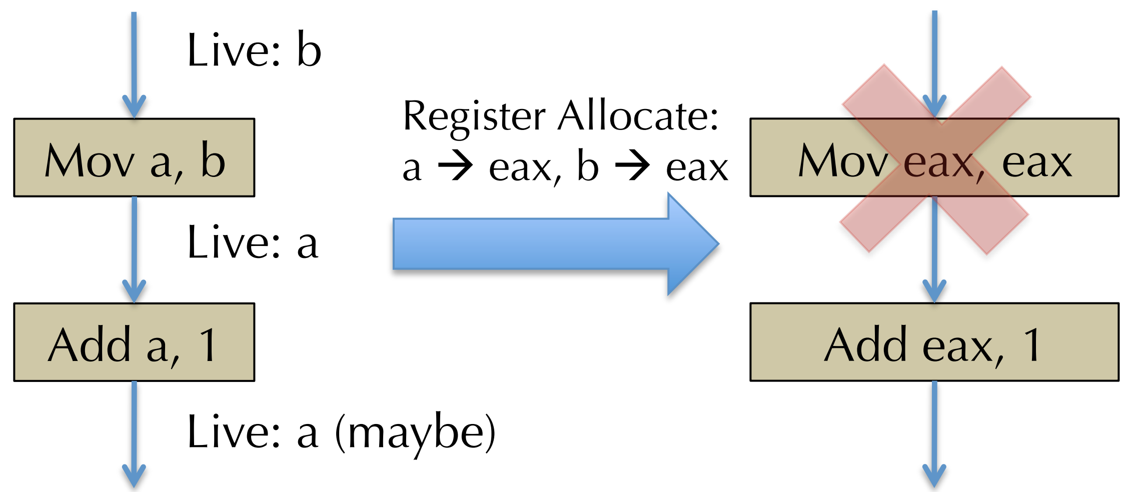
- For precision, it is helpful to think of the "fall through" between sequential instructions as an edge of the control-flow graph too.
  - Different implementation tradeoffs in practice…



Liveness is Associated with *Edges*



- This is useful so that the same register can be used for different temporaries in the same statement.
- Example: `a = b + 1`
- Compiles to:



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 node/statement s define:
  - use[s] : set of variables used by s
  - def[s] : set of variables defined 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

- The first clause says that v will be used on some path starting from edge e.
- The second clause 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? (e.g. LLVM IR uses SSA, so it doesn't allow redefinition ⇒ simplify liveness 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 pasing through no def of 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 across 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
- This is an instance of a general framework for computing program properties: dataflow analysis

Dataflow Value Sets for Liveness

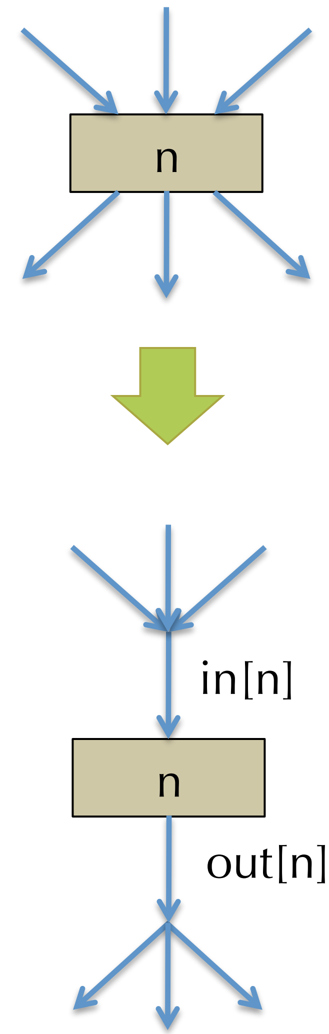
- Nodes are program statements, so:
- use[n] : set of variables used by n
- def[n] : set of variables defined by n
- in[n] : set of variables live on entry to n
- out[n] : set of variables live on exit from n

- Associate in[n] and out[n] with the "collected" information about incoming/outgoing edges

- For Liveness: what constraints are there among these sets?
- Clearly:

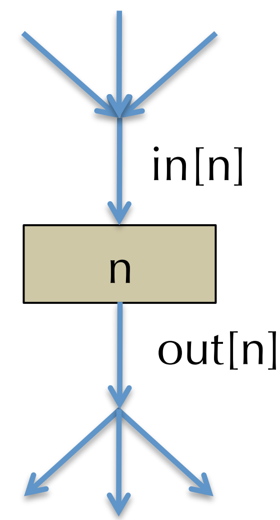
$$in[n] \supseteq use[n]$$

- What other constraints?



Other Dataflow Constraints

- 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, 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

- Find a solution to those constraints by starting from a rough guess.
- Start with: $in[n] = \emptyset$ and $out[n] = \emptyset$
- They don't satisfy the constraints:
  - $in[n] \supseteq use[n]$
  - $in[n] \supseteq out[n] - def[n]$
  - $out[n] \supseteq in[n']$ if $n' \in succ[n]$

- 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)
  - $in[n] = use[n] \cup (out[n] - def[n])$
  - $out[n] = \bigcup_{n' \in succ[n]} in[n']$

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

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

    end

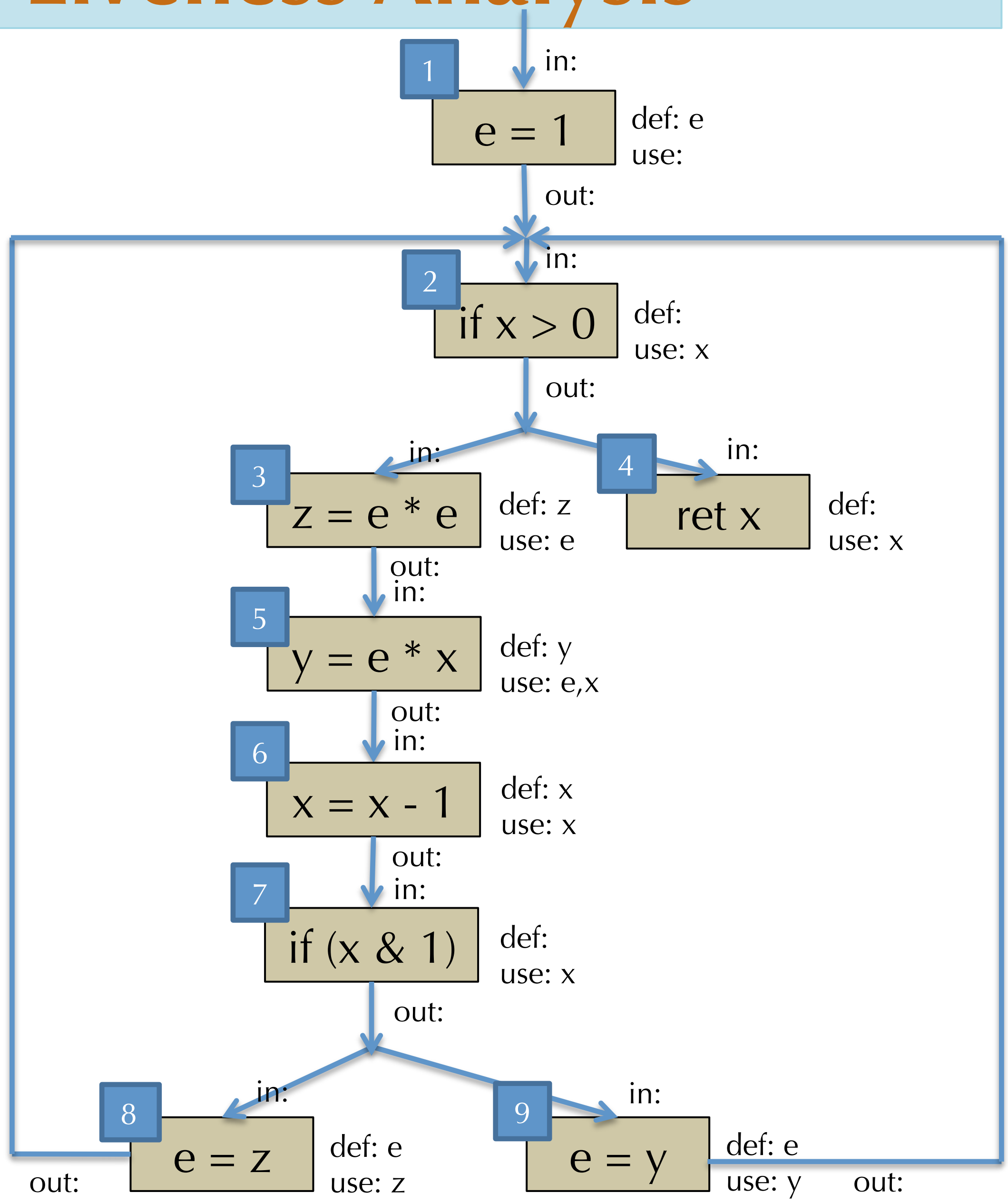
end

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

Example 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;
```



Example Liveness Analysis

Each iteration update:

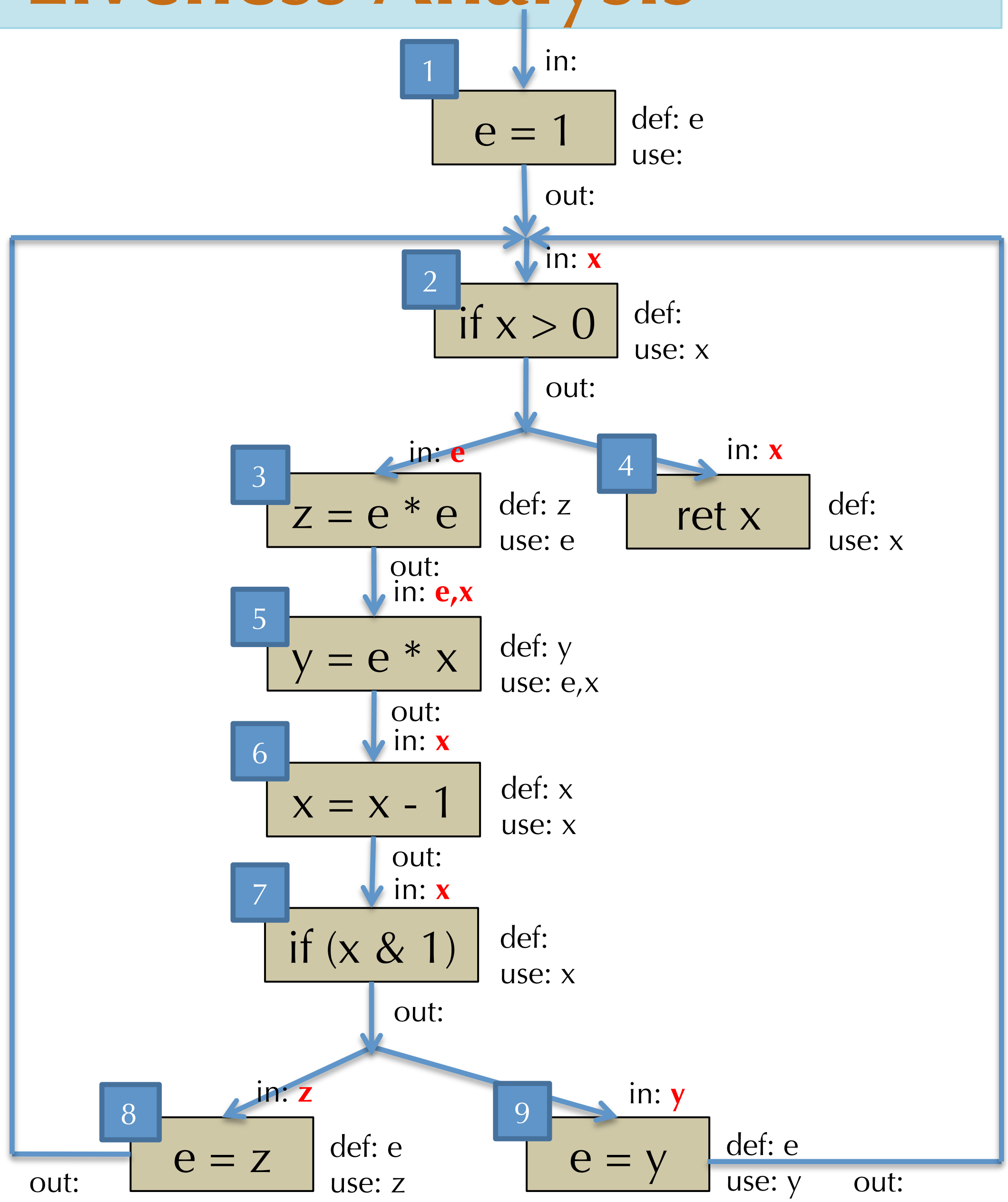
out[n] := $\cup_{n' \in succ[n]}$ in[n']

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

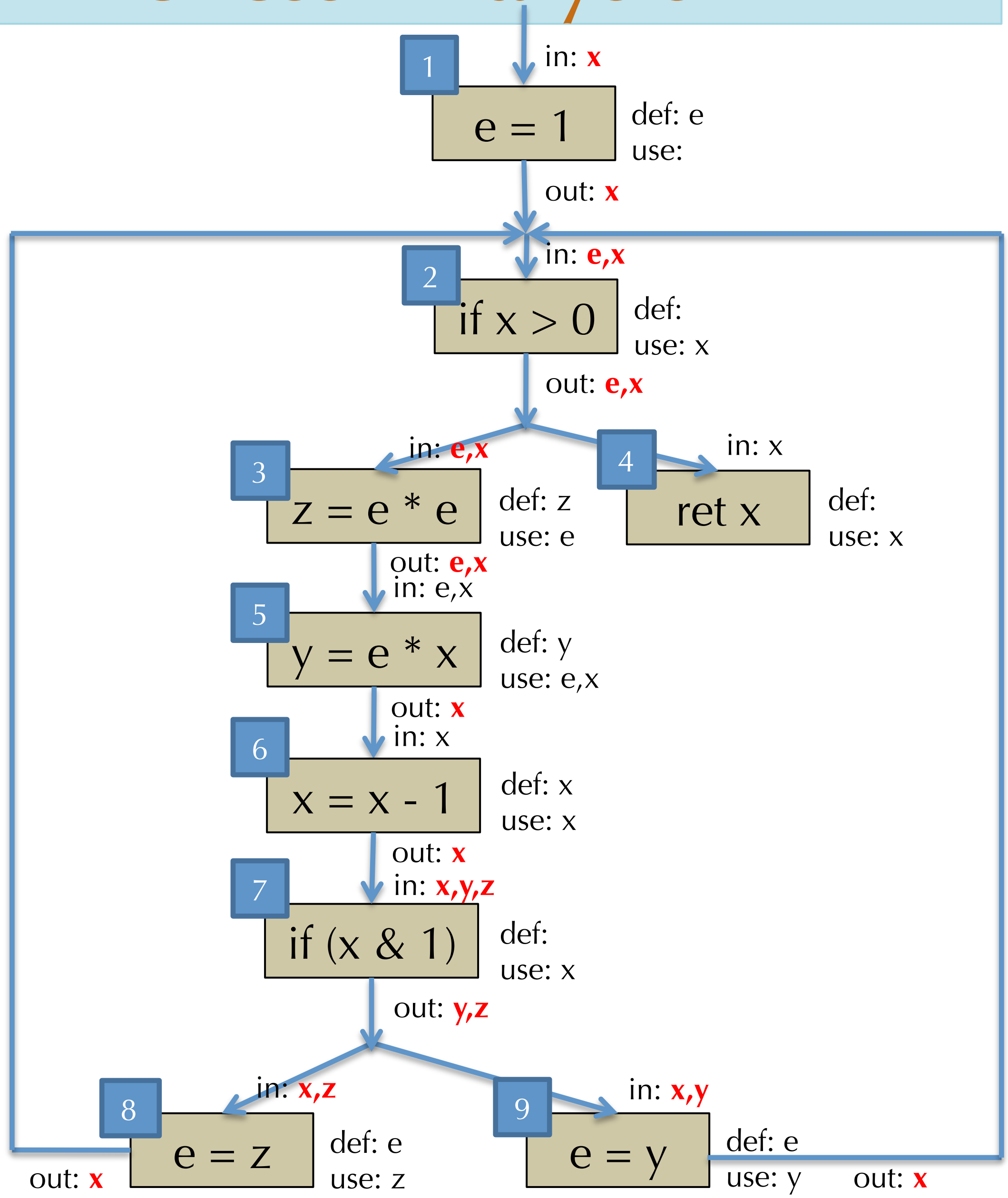
Each iteration update:

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

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

- Iteration 2:

out[1]= x
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



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 3:

out[1]= e,x

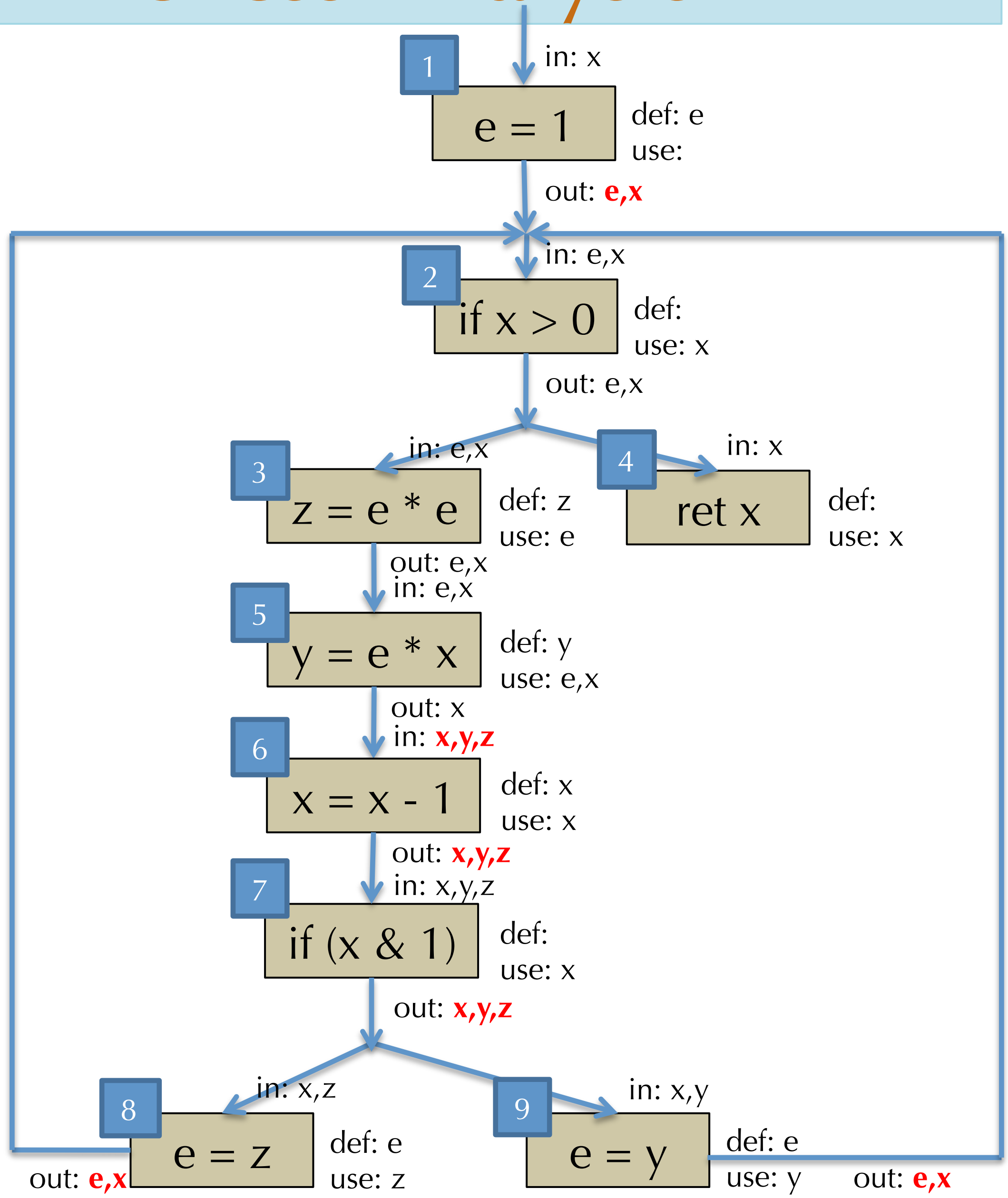
out[6]= x,y,z

in[6]= x,y,z

out[7]= x,y,z

out[8]= e,x

out[9]= e,x



Example Liveness Analysis

Each iteration update:

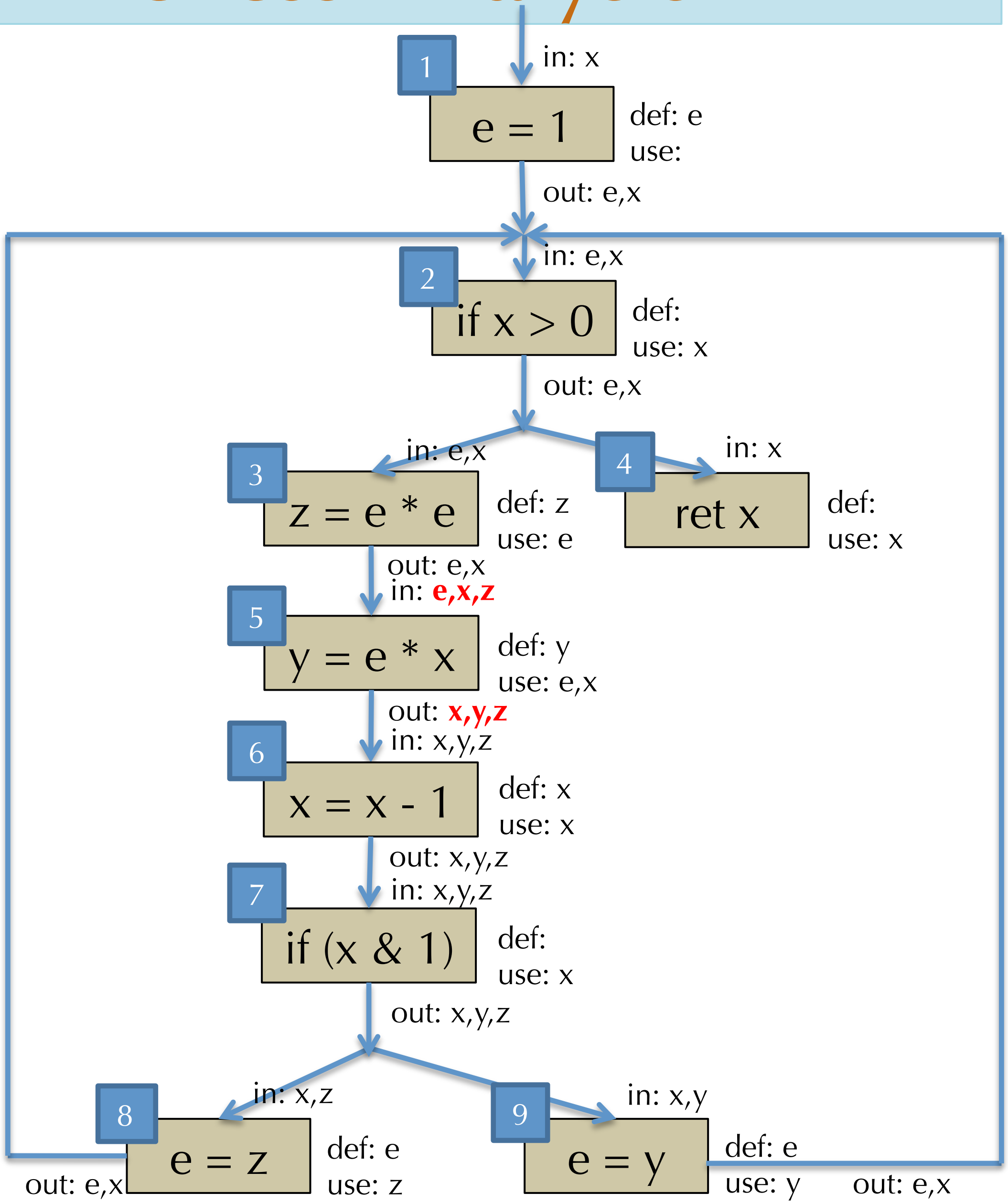
$out[n] := \bigcup_{n' \in succ[n]} in[n']$

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

- Iteration 4:

out[5]= x,y,z

in[5]= e,x,z



Example Liveness Analysis

Each iteration update:

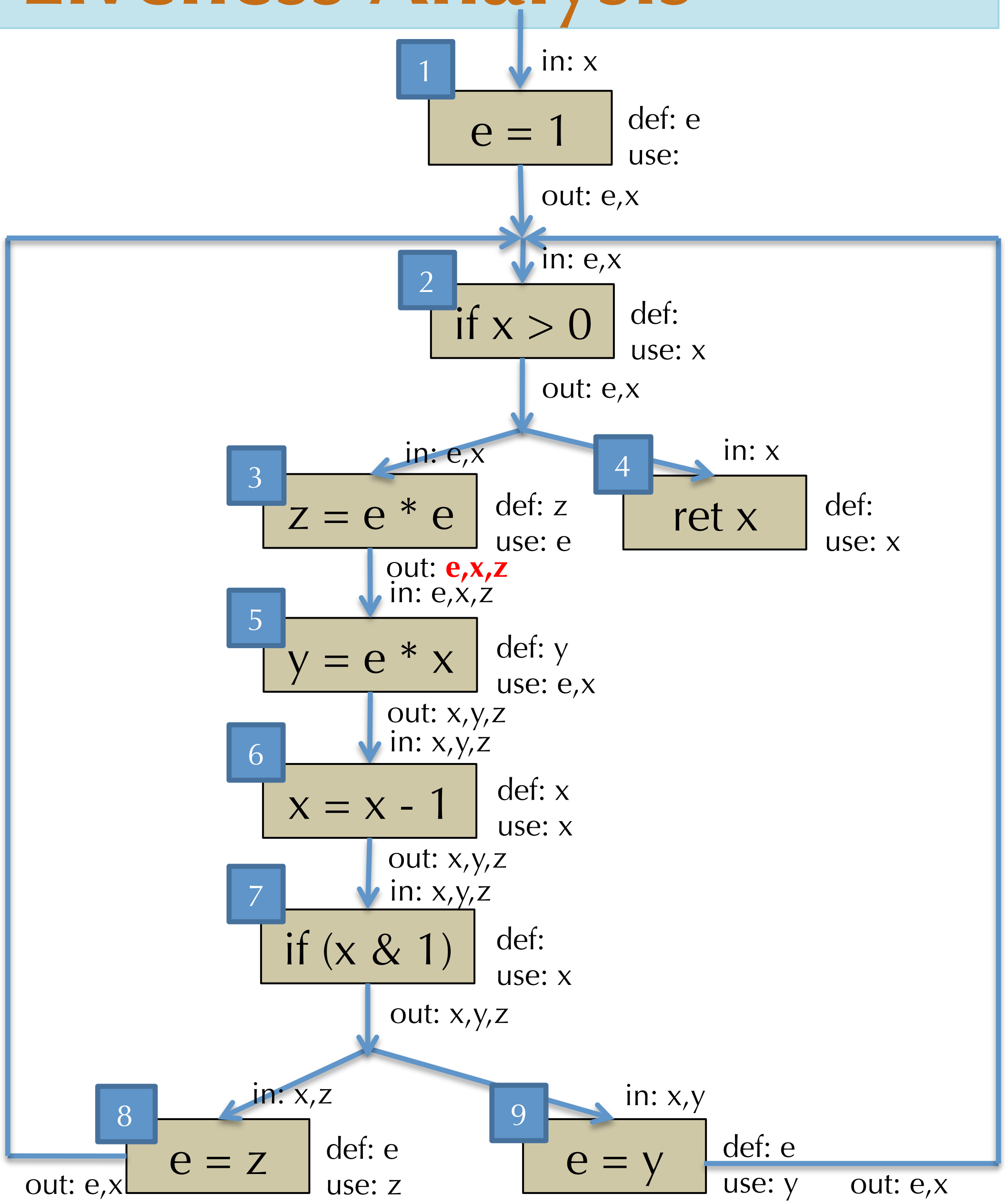
out[n] := $\cup_{n' \in succ[n]}$ in[n']

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

- Iteration 5:

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: $out[n] := \bigcup_{n' \in succ[n]} in[n']$
  - This is the only rule that involves more than one node

- If 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

- 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 to worklist
end
```