Lecture 21

CIS 341: COMPILERS

Announcements

- Project 5 Compiling objects in full Oat
  - Available from the course web pages
  - Updated oat.pdf fixes a few typos (mentioned on Piazza)
  - Due April 8th

- Final Exam:
  - Tuesday, April 30th noon-2:00 pm
  - Moore 216

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 8 max registers available on 32-bit X86
  - ESP and (often) EBP are reserved, so only 6-7 really 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) {
  int a = 0; {int b = x + x; a = b * b;}
  int 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) {
  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,x all overlap – they're all in scope.
- 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, …

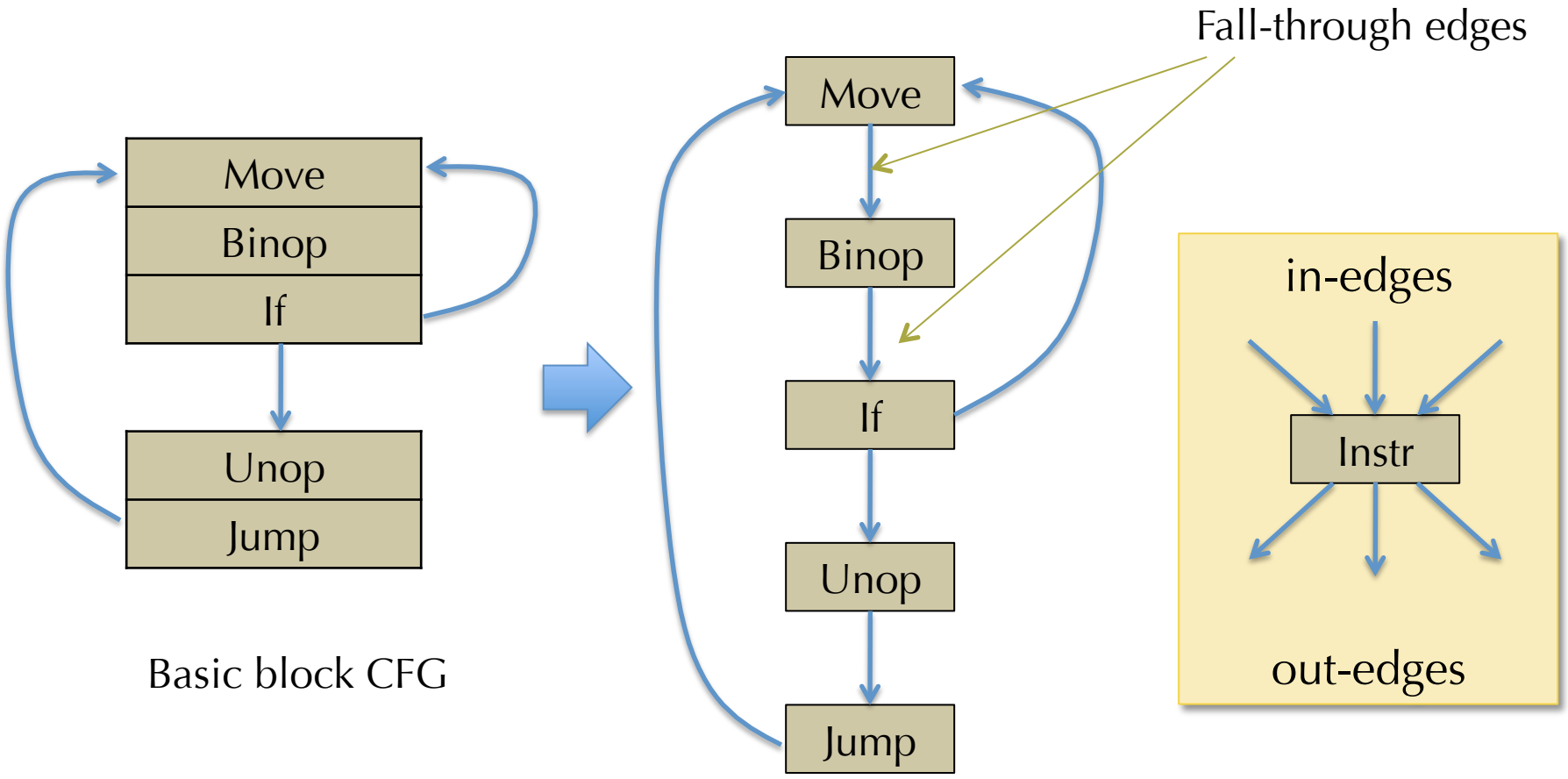
Control-flow Graphs Revisited

- 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.)

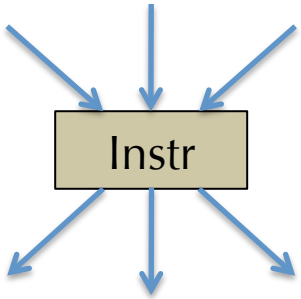
- A *control flow graph*
  - Nodes are blocks
  - There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  - There are no "dangling" edges – there is a block for every jump target.

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.
  - In practice, identify instructions by offsets within basic blocks



Liveness is Associated with *Edges*

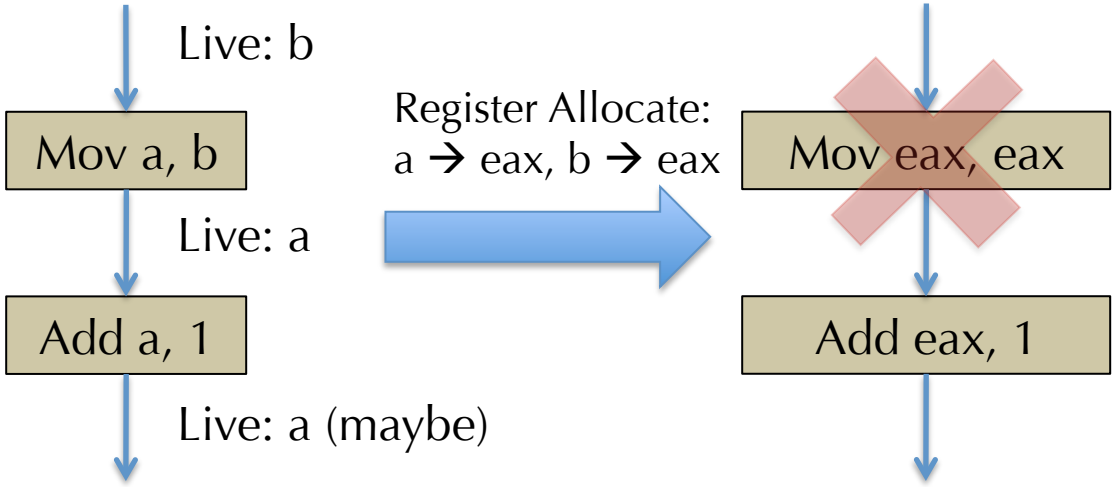


Live: a, b

Live: b, d, e

- 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

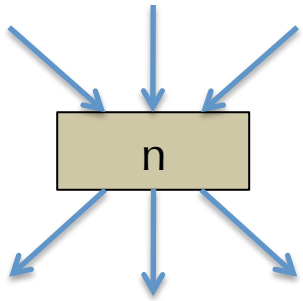
- Approach: define *equations* that must be satisfied by any liveness determination.
  - Equations based on "obvious" constraints.

- 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



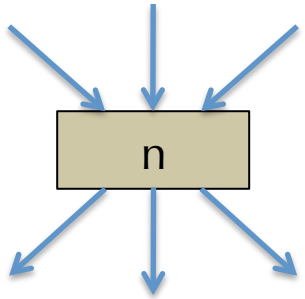
- What constraints are there among these sets?
- Clearly:

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

- What other constraints?

Other Dataflow Constraints

- We have: in[n] ⊇ use[n]
  - "A variable must be live on entry to n if it is used by n"



- Also: in[n] ⊇ 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] ⊇ in[n'] if n' ∈ 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:
  - $in[n] = use[n] \cup (out[n] - def[n])$
  - $out[n] = \bigcup_{n' \in succ[n]} in[n']$

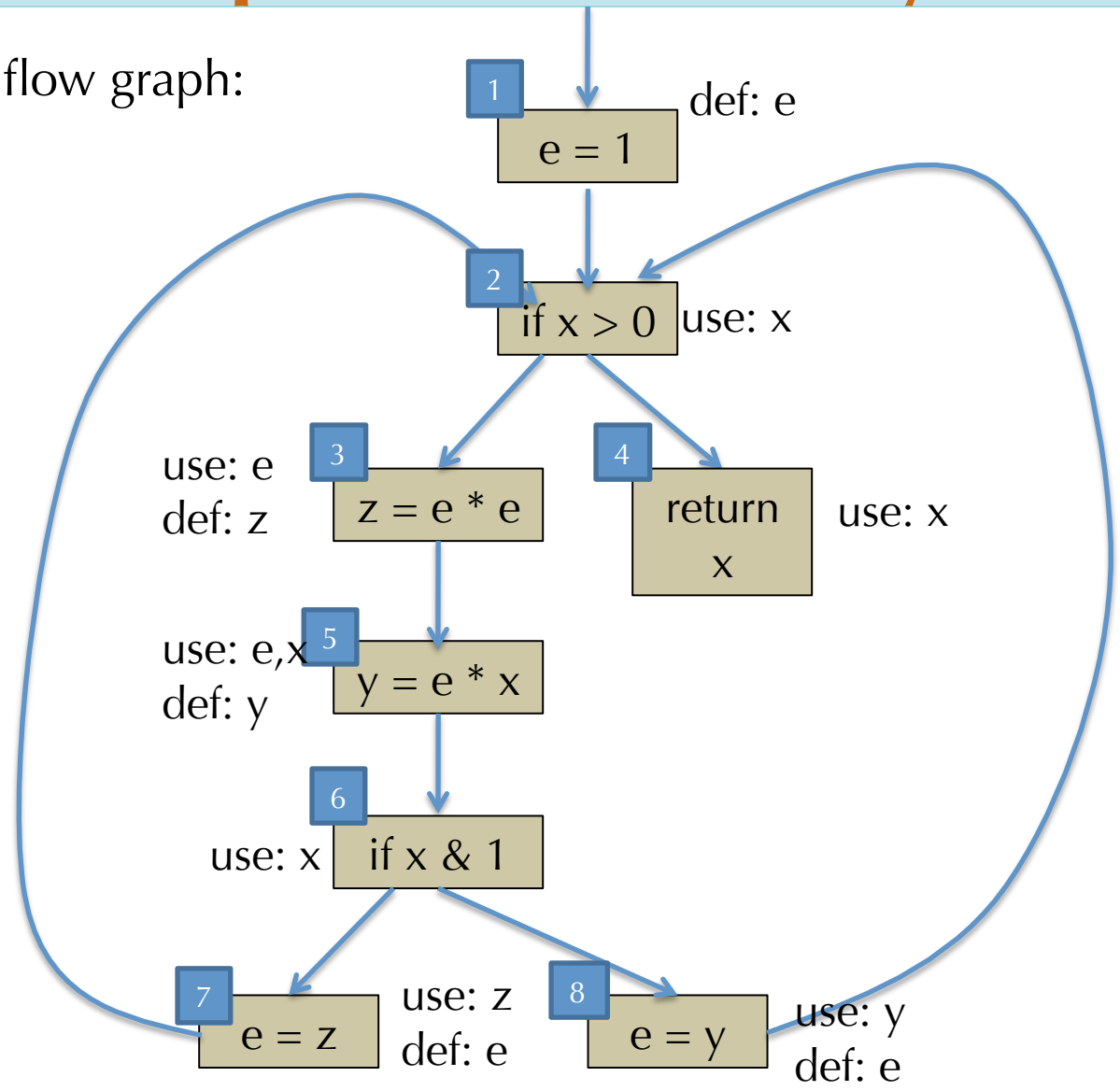
Complete Liveness Analysis Algorithm

for all n, in[n] := ∅, out[n] := ∅
repeat until no change in 'in' and 'out'
    for all n
        out[n] := $\bigcup_{n' \in succ[n]}$ in[n']
        in[n] := use[n] ∪ (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 ∅?

Example Liveness Analysis

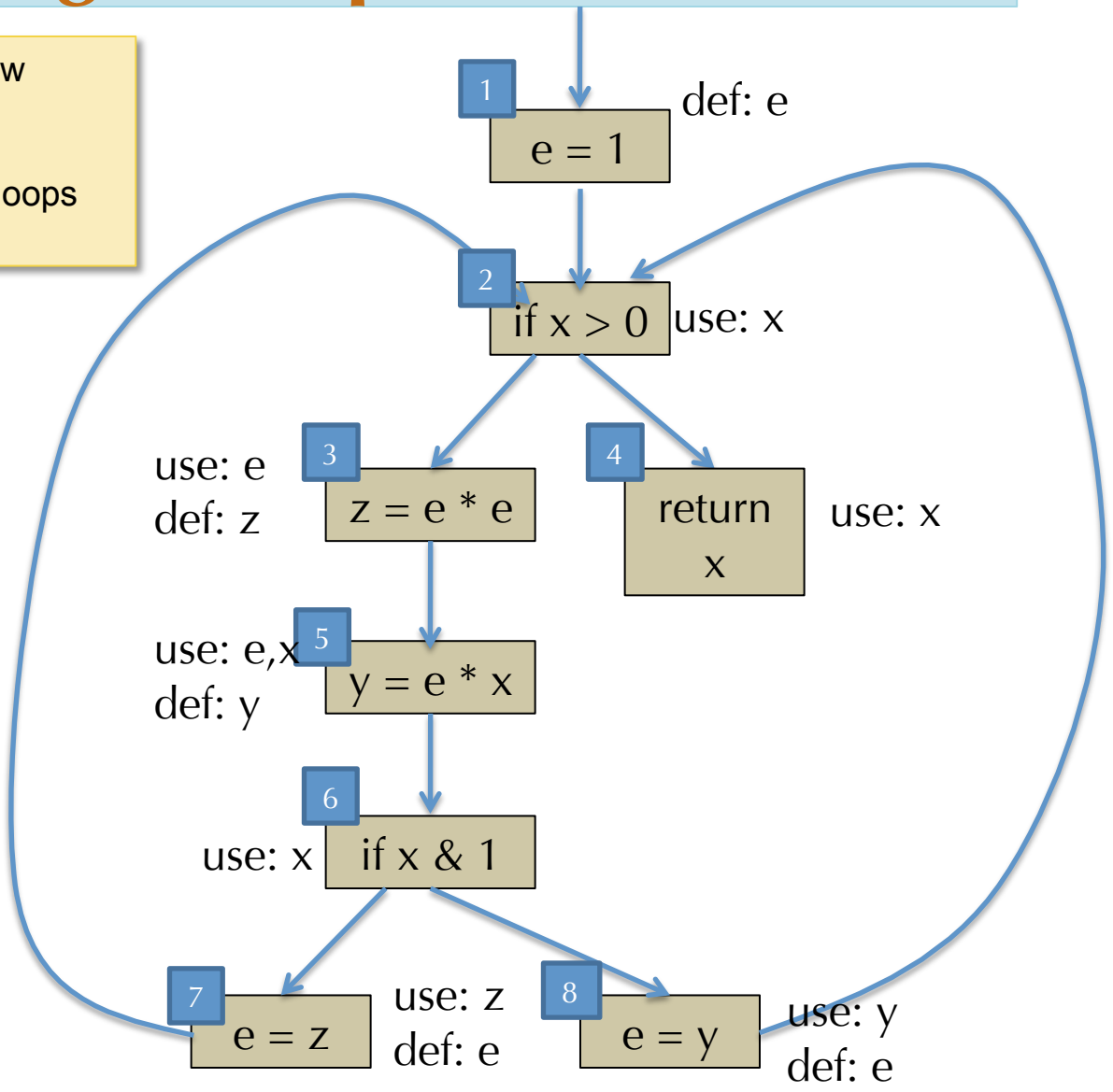
- Example flow graph:



Iterating the Equations

2: in = {x}
3: in = {e}
4: in = {x}
5: in = {e,x}
6: in = {x}
7: out = {x}, in = {x,z}
8: out = {x}, in = {x,y}
1: out = {x}, in = {x}
2: out = {e,x}, in = {e,x}
3: out = {e,x}, in = {e,x}
5: out = {x}, in = {e,x}
6: out = {x,y,z}, in = {x,y,z}
7: out = {e,x}, in = {x,z}
8: out = {e,x}, in = {x,y}
1: out = {e,x}, in = {x}
5: out = {x,y,z}, in = {e,x,z}
3: out = {e,x,z}, in = {e,x}
done!

Steps at left show which sets have changed.
Brackets group loops over "for all n".



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 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] := ∅, out[n] := ∅
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
```

OTHER DATAFLOW ANALYSES

Generalizing Dataflow Analyses

- The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well.
  - Reaching definitions analysis
  - Available expressions analysis
  - These analyses follow the same 3-step approach as for liveness.

- To see these as an instance of the same kind of algorithm, it is useful to work over a canonical intermediate instruction representation called *quadruples*
  - Allows easy definition of def[n] and use[n]
  - A "looser" variant of LLVM's IR that doesn't require the "static single assignment"
  - Easier for dataflow analyses (SSA form is

Quadruple Format

- A Quadruple sequence is just a control-flow graph (flowgraph) where each node is a quadruple:
- Quadruple forms n:

Quadruple forms n:	def[n]	use[n]	description
a = b op c	{a}	{b,c}	arithmetic
a = [b]	{a}	{b}	load
[a] = b	Ø	{b}	store
jump L	Ø	Ø	jump
if a goto L1 else L2	Ø	{a}	branch
L:	Ø	Ø	label
a = f(b_1,…,b_n)	{a}	{b_1,…,b_n}	call w/return
f(b_1,…,b_n)	Ø	{b_1,…,b_n}	call no return
return a	Ø	{a}	return

- The LLVM IR we've been using is already in this format…
  - In fact the SSA property is a refinement of the quadruples approach: it further restricts the use of variables.
  - For the time being, we'll ignore the alloca instruction and assume that all storage variables are allocated "before" the part of the CFG we're analyzing.
  - We'll see more about the connection between SSA and dataflow analysis later.

REACHING DEFINITIONS

Reaching Definition Analysis

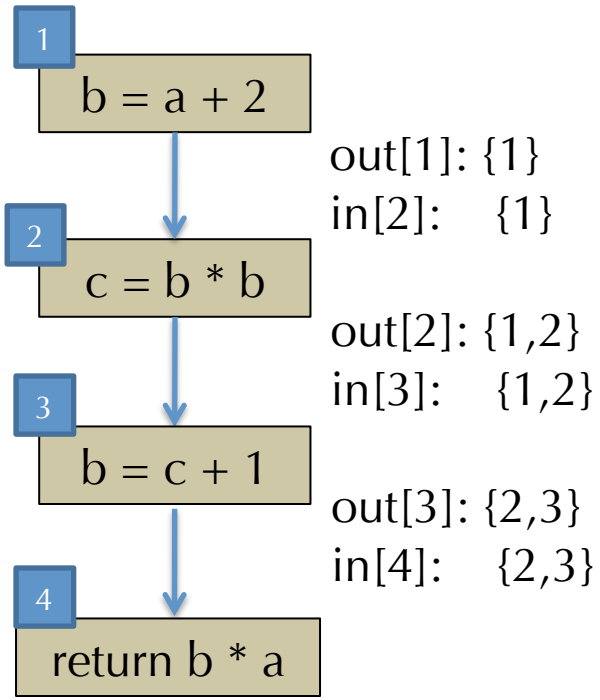
- Question: what uses in a program does a given variable definition reach?

- This analysis is used for constant propagation & copy prop.
  - If only one definition reaches a particular use, can replace use by the definition (for constant propagation).
  - Copy propagation additionally requires that the copied value still has its same value – computed using an *available expressions* analysis (next)

- Input: Quadruple CFG
- Output: in[n] (resp. out[n]) is the set of nodes defining some variable such that the definition may reach the beginning (resp. end) of node n

Example of Reaching Definitions

- Results of computing reaching definitions on this simple CFG:



Reaching Definitions Step 1

- Define the sets of interest for the analysis
- Let defs[a] be the set of *nodes* that define the variable a
- Define gen[n] and kill[n] as follows:
- Quadruple forms n: gen[n] kill[n]

a = b op c	{n}	defs[a] - {n}
a = [b]	{n}	defs[a] - {n}
[a] = b	∅	∅
jump L	∅	∅
if a goto L1 else L2	∅	∅
L:	∅	∅
$a = f(b_1,\ldots,b_n)$	{n}	defs[a] - {n}
$f(b_1,\ldots,b_n)$	∅	∅
return a	∅	∅

Reaching Definitions Step 2

- Define the constraints that a reaching definitions solution must satisfy.
- out[n] ⊇ gen[n]
  "The definitions that reach the end of a node at least include the definitions generated by the node"

- in[n] ⊇ out[n'] if n' is in pred[n]
  "The definitions that reach the beginning of a node include those that reach the exit of *any* predecessor"

- out[n] ∪ kill[n] ⊇ in[n]
  "The definitions that come in to a node either reach the end of the node or are killed by it."
  - Equivalently: out[n] ⊇ in[n] - kill[n]

Reaching Definitions Step 3

- Convert constraints to iterated update equations:
- $in[n] := \bigcup_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] - kill[n])$

- Algorithm: initialize $in[n]$ and $out[n]$ to $\emptyset$
  - Iterate the update equations until a fixed point is reached

- The algorithm terminates because in[n] and out[n] increase only *monotonically*
  - At most to a maximum set that includes all variables in the program
- The algorithm is precise because it finds the *smallest* sets that satisfy the constraints.

AVAILABLE EXPRESSIONS