Dataflow Analysis - II

(Constant Propagation and Reaching Definitions)

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Content

- Dataflow Analysis
  - Constant Propagation (background for lattice theory)
  - Reaching Definitions
  - General Framework
  - Implementation task in LLVM

Constant Propagation

- Bigger problem size:
  - Which lines using X could be replaced with a constant value? (apply only constant propagation)
  - How can we automate to find an answer to this question?

```
1. X := 2
2. Label1:
3. Y := X + 1
4. if Z > 8 goto Label2
5. X := 3
6. X := X + 5
7. Y := X + 5
8. X := 2
9. if Z > 10 goto Label1
10.X := 3
11.Label2:
12.Y := X + 2
13.X := 0
14.goto Label3
15.X := 10
16.X := X + X
17.Label3:
18.Y := X + 1
```

Constant Propagation

- Problem statement:
  - Replace use of a variable X by a constant K
- Requirement:
  - **property**: on every path to the use of X, the last assignment to X is: X=K

    Same as: “is X=K at a program point?”

    At any program point where the above property holds, we can apply constant propagation.

How can we find constants?

Symbolic Execution

- Ideal: run program and see which variables are constant
  - Problem: variables can be constant with some inputs, not others – need an approach that works for all inputs!
  - Problem: program can run forever (infinite loops?) – need an approach that we know will finish
- Idea: run program *symbolically*
  - Essentially, keep track of whether a variable is constant or not constant (but nothing else)

Slide courtesy: Prof. Milind Kulkarni

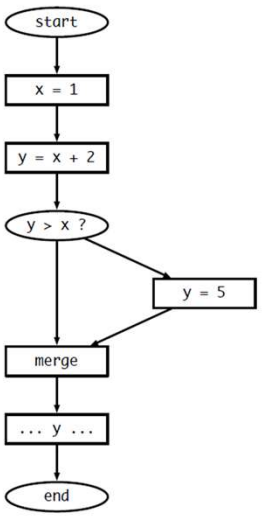
Overview of algorithm

- Build control flow graph
  - We'll use statement-level CFG (with merge nodes) for this
- Perform symbolic evaluation
  - Keep track of whether variables are constant or not
- Replace constant-valued variable uses with their values, try to simplify expressions and control flow

Slide courtesy: Prof. Milind Kulkarni

Build CFG

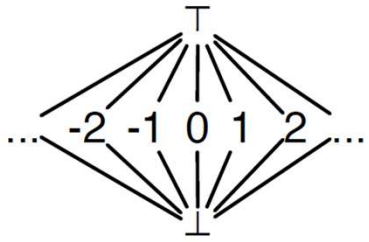
```
x = 1;
y = x + 2;
if (y > x) then y = 5;
... y ...
```



Slide courtesy: Prof. Milind Kulkarni

Symbolic evaluation

- Idea: replace each value with a symbol
  - constant (specify which), no information, definitely not constant
- Can organize these possible values in a *lattice*
  - Set of possible values, arranged from least information to most information



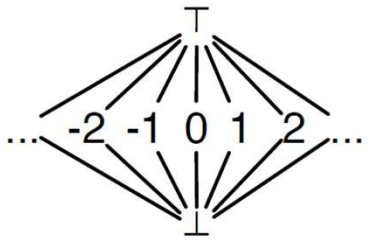
5/14/2025 Slide courtesy: Prof. Milind Kulkarni 8

In the next step, we need to symbolically evaluate the code. What this means that in a step-by-step execution through the program, the variables of a program don't take specific values. Instead, they take symbols: e.g. for constant propagation, whenever we encounter the three possible scenarios for variables --- may or maynot be a const / don't know, definitely not a constant, constant ---, we resp. use the symbols bottom (inverted T), top (T), and one of the infinitely many constant values.

These symbols are shown in a lattice arranged from least information (don't know) to most information (definitely not a constant).

Symbolic evaluation

- Evaluate expressions symbolically: eval(e, V_{in})
  - If e evaluates to a constant, return that value. If any input is ⊤ (or ⊥), return ⊤ (or ⊥)
    - Why?
- Two special operations on lattice
  - meet(a, b) – highest value less than or equal to both a and b
  - join(a, b) – lowest value greater than or equal to both a and b



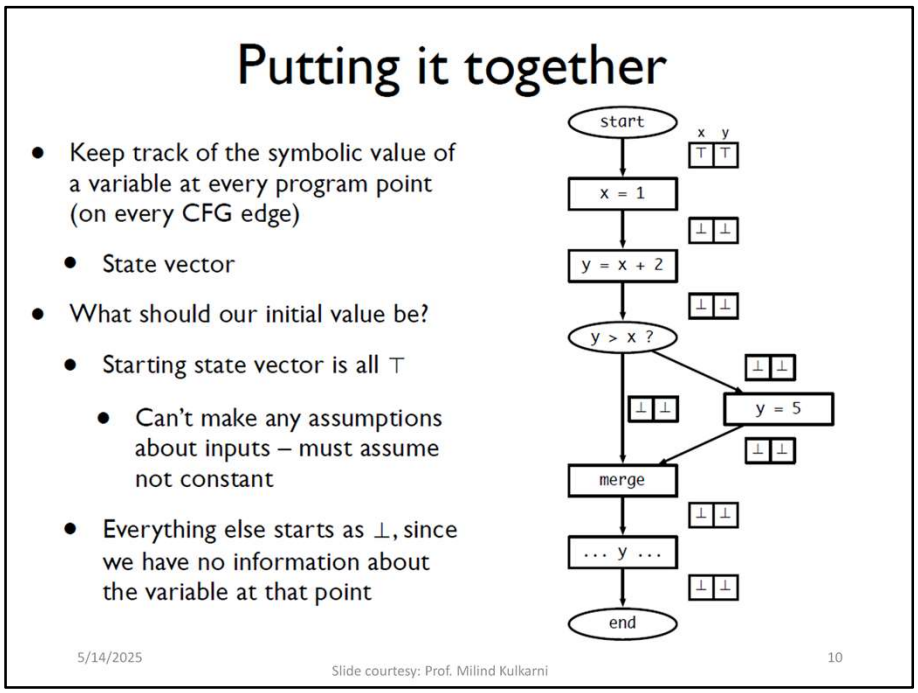
Join often written as a ⊔ b
Meet often written as a ⊓ b

5/14/2025 Slide courtesy: Prof. Milind Kulkarni 9

In a step-by-step execution of the program statements, we call the eval function on each statement. Ignore the inputs to the eval function for now. We'll see it soon.

In a statement, if a variable is computed as a result of two or more other variables and any of those variables is not a constant (or don't know), then we definitely know that the variable being computed is definitely not a constant (or don't know).

Two special operations example:
meet(don't know, constant value K) = constant value K
join(don't know, const value K) = don't know

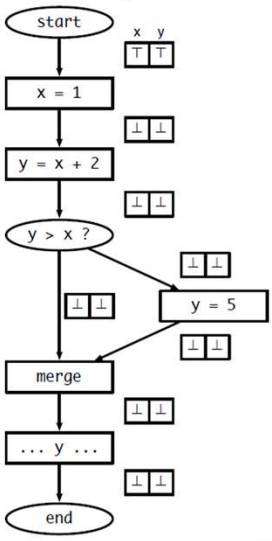


So, in our automatic computation of determining whether a variable is constant or not, we have two steps: 1) build a statement-level cfg and 2) symbolically evaluate For the running example introduced earlier, if we are analyzing the const property of variables x and y at any point in the program, then we need to keep track of a state vector for these two variables.

The state vector is shown as a label on the edges of a CFG. To begin with, we initialize the label on the starting edge to (top, top)=(definitely not a constant, definitely not a constant). All other edges have labels (bottom, bottom), indicating that we don't know anything about the variables (makes sense because we haven't yet analyzed those nodes/statements).

Executing symbolically

- For each statement `t = e` evaluate `e` using V_{in}, update value for `t` and propagate state vector to next statement
- What about switches?
  - If `e` is true or false, propagate V_{in} to appropriate branch
  - What if we can't tell?
    - Propagate V_{in} to both branches, and symbolically execute both sides
- What do we do at merges?

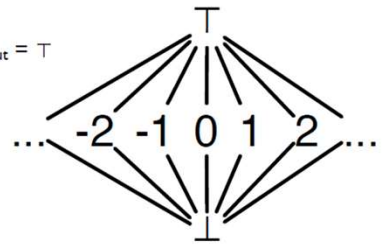


5/14/2025 Slide courtesy: Prof. Milind Kulkarni 11

When we visit a node / evaluate a statement of the form " t = e" what should happen? We use the incoming state vector, Vin, and depending upon whether e is a constant or not (e may be an expression of two or more variables), we update the value of t in Vin and propagate the updated vector on outgoing edge.

Handling merges

- Have two different V_{in}s coming from two different paths
- Goal: want new value for V_{in} to be *safe* (shouldn't generate wrong information), and we don't know which path we actually took
- Consider a single variable. Several situations:
  - $V_1 = \bot, V_2 = * \rightarrow V_{out} = *$
  - V_1 = constant x, V_2 = x $\rightarrow V_{out}$ = x
  - V_1 = constant x, V_2 = constant y $\rightarrow V_{out} = \top$
  - $V_1 = \top, V_2 = * \rightarrow V_{out} = \top$
- Generalization:
  - $V_{out} = V_1 \sqcup V_2$



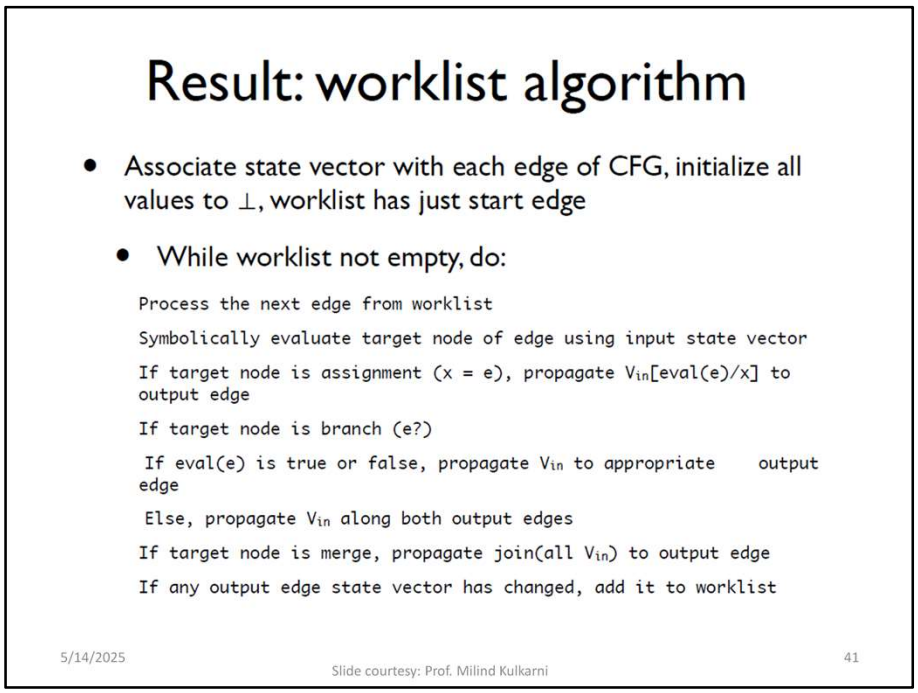
Slide courtesy: Prof. Milind Kulkarni

Let's say that on one path we know x is a constant with value 3 and on another path we know that x is a constant with value 2, what can we say about the place where both paths merge? --- we know that x is definitely not a constant (can be 2 or 3)

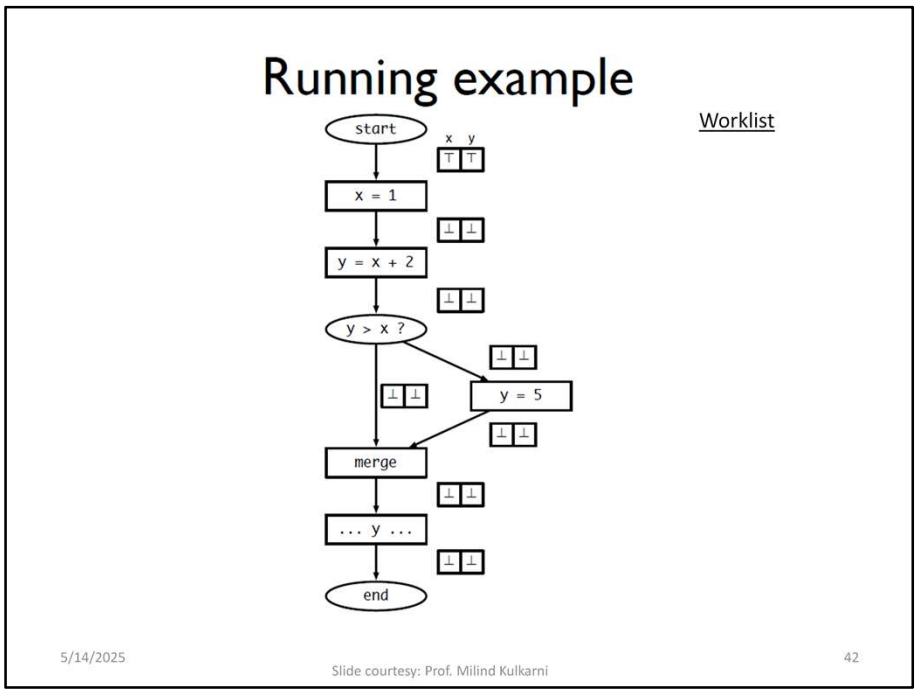
Similarly, if we know that on path A we know that x is definitely not a constant and on another path, we know that x is a constant with value 2 (or don't know anything), at the merge point we try to be conservative and say that x is definitely not a constant.

If we don't know anything about x on a path and on another path we know that x is definitely not a constant (or a constant with value K), we say at the merge point, that x is definitely not a constant (or a constant with value K).

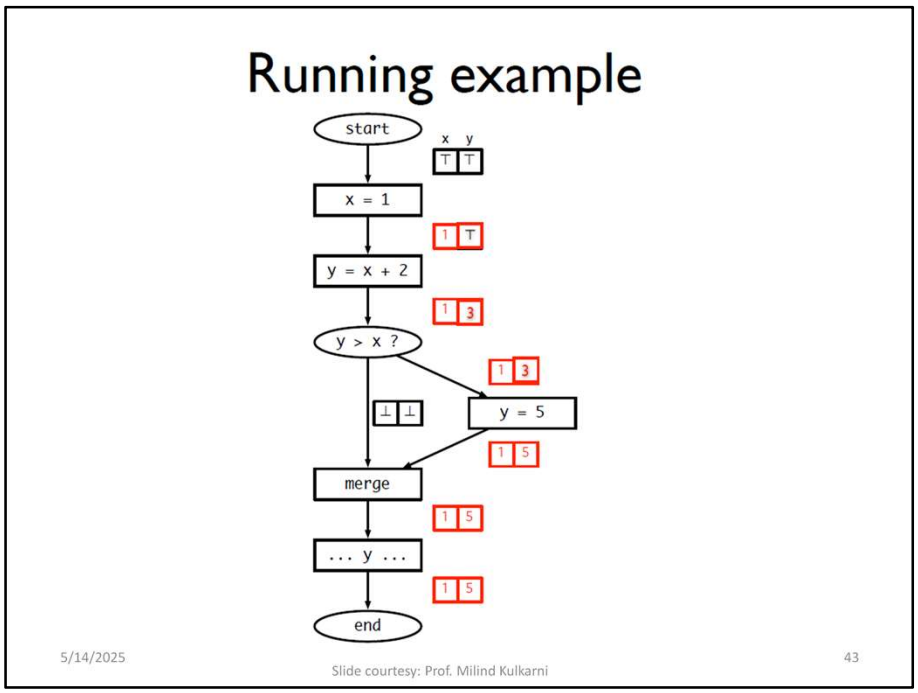
Translating these possibilities in terms of the lattice for constant propagation, we see that at the merge point, we are choosing a symbol that is at least as high as the higher of the two symbols corresponding to the two paths. This exactly refers to the definition of the join operation on lattice from the previous slide. So, we write Vout= V1 U V2 at merge points.



Here is the worklist algorithm for systematically computing/updating the state vectors.



Initial state.

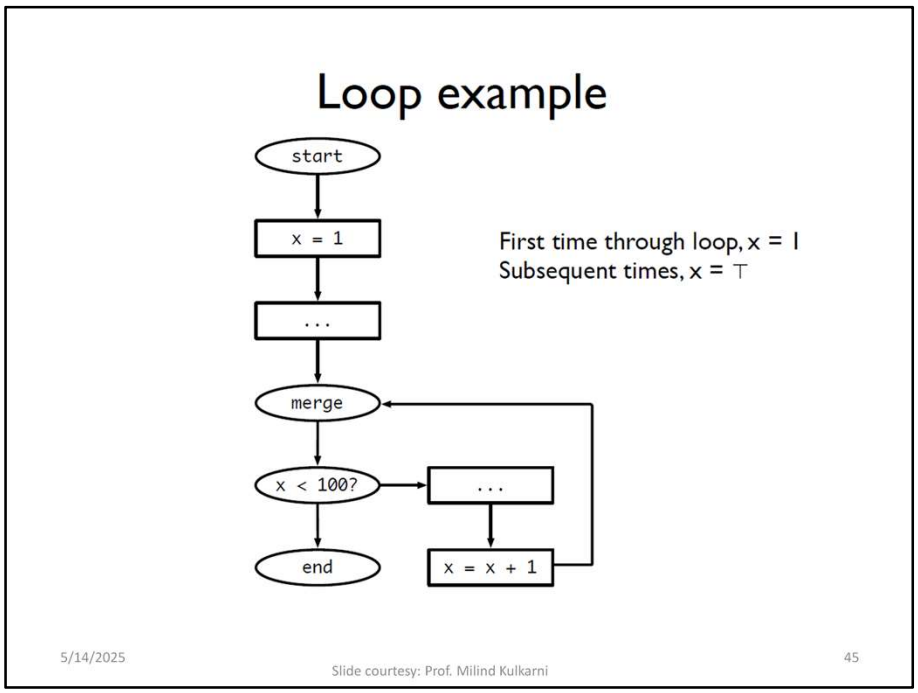


At “x=1”: Vin=(T,T) Vout=(1,T) //eval(1,Vin) returns 1.
Now, since the outedge changed from (bottom, bottom) to (1, top) (1,T), we add (1,T) to worklist and process it.
At “y=x+2”: we know that x is a constant with value 2 (from Vin). So, y is a constant with value 3. We update Vout as (1,3)
At “y>x?”: we know that y is a constant with value 3 and x is a constant with value 1. This always evaluates to true. Hence, we propagate Vout along only the true direction.
At merge point, we apply the join operation and get the resulting vector as (1,5).
Continuing in this way, we get the state vectors shown.

What do we do about loops?

- Unless a loop never executes, symbolic execution looks like it will keep going around to the same nodes over and over again
- Insight: if the input state vector(s) for a node don't change, then its output doesn't change
  - If input stops changing, then we are done!
- Claim: input will eventually stop changing. Why?

Slide courtesy: Prof. Milind Kulkarni



In this example, when the loop is entered for the first time, the edge between “merge” and “x<100?” (lets call this edge E) is computed after applying join on state vectors(1, bottom) --- the edge coming from “x=x+1” to merge (lets call this edge G) is initialized to bottom. The edge coming from “…” to merge has 1 (because we know that x is a constant with value 1). Join(1, bottom) = 1.

At “x<100?” when Vin is 1, the condition is true. So, Vout (1) is propagated only on the edge between “x<100” and the “…” within the loop (lets call this edge F). Continuing, the label on edge G is updated as (2). “x=x+1” with Vin=(1) computes Vout as (2).

Now at merge, we need to compute join(2, 1). This gives Vout and the updated label on edge E as (T). Why T? refer the slide on handling merges.

All subsequent times, we get x=T.

Complexity of algorithm

- V = # of variables, E = # of edges
- Height of lattice = 2 → each state vector can be updated at most 2 *V times.
- So each edge is processed at most 2 *V times, so we process at most 2 * E *V elements in the worklist.
- Cost to process a node: O(V)
- Overall, algorithm takes $O(EV^2)$ time

Slide courtesy: Prof. Milind Kulkarni

Question

- Can we generalize this algorithm and use it for more analyses?

Slide courtesy: Prof. Milind Kulkarni

Constant propagation

- Step 1: choose lattice (which values are you going to track during symbolic execution)?
  - Use constant lattice
- Step 2: choose direction of dataflow (if executing symbolically, can run program backwards!)
  - Run forward through program
- Step 3: create *transfer functions*
  - How does executing a statement change the symbolic state?
- Step 4: choose *confluence operator*
  - What do do at merges? For constant propagation, use join

Slide courtesy: Prof. Milind Kulkarni

Reaching Definitions

- Recall – Terminology
  - Use: an instruction uses all its operands / arguments
  - Definition: an instruction defines the variable that it writes to
  - Available: definitions that reach a given program point are available there
  - Kill: any definition currently kills all the available definitions

Reaching definitions:

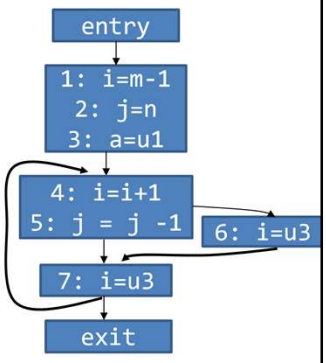
For every definition and every use, determine whether the definition reaches the use.

Reaching definitions

- What definitions of a variable *reach* a particular program point
  - A definition of variable x from statement s reaches a statement t if there is a path from s to t where x is not redefined
- Especially important if x is used in t
  - Used to build *def-use* chains and *use-def* chains, which are key building blocks of other analyses
    - Used to determine dependences: if x is defined in s and that definition reaches t then there is a flow dependence from s to t
  - We used this to determine if statements were loop invaraint
    - All definitions that reach an expression must originate from outside the loop, or themselves be invariant

Reaching Definitions - Example

- What are In, Out, Gen, and Kill sets?



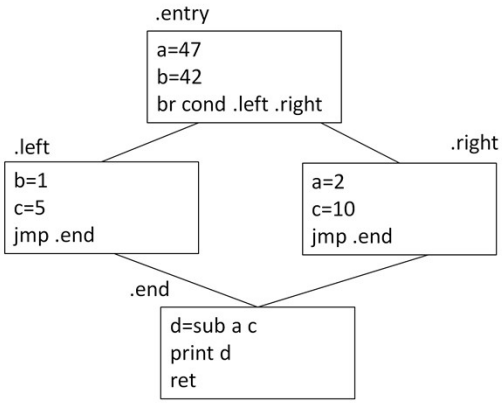
$In(b) = \cup_{i \in Pred(b)} Out(i)$

$Out(b) = gen(b) \cup (In(b) - kill(b))$

//set that contains all statements that may define some variable x in b. E.g. gen(1:a=3;2:a=4)={2}

//set that contains all statements that define a variable x that is also defined in b. E.g. kill(1:a=3; 2:a=4)={1,2}

Reaching Definitions - Example



How many uses are here?

Creating a reaching-def analysis

- Can we use a powerset lattice?
- At each program point, we want to know which definitions have reached a particular point
  - Can use powerset of set of definitions in the program
    - V is set of variables, S is set of program statements
    - Definition: $d \in V \times S$
      - Use a tuple, <v, s>
  - How big is this set?
    - At most $|V \times S|$ definitions

Slide courtesy: Prof. Milind Kulkarni

Forward or backward?

- What do you think?

Slide courtesy: Prof. Milind Kulkarni

Choose confluence operator

- Remember: we want to know if a definition *may* reach a program point
- What happens if we are at a merge point and a definition reaches from one branch but not the other?
  - We don't know which branch is taken!
  - We should union the two sets – any of those definitions can reach
- We want to avoid getting too many reaching definitions → should start sets at $\perp$

Slide courtesy: Prof. Milind Kulkarni

Transfer functions for RD

- Forward analysis, so need a slightly different formulation
  - Merged data flowing into a statement

$$IN(s) = \bigcup_{t \in pred(s)} OUT(t)$$
$$OUT(s) = \mathbf{gen}(s) \cup (IN(s) - \mathbf{kill}(s))$$

- What are gen and kill?
  - gen(s): the set of definitions that *may* occur at s
    - e.g., gen(s_1: x = e) is <x, s_1>
  - kill(s): all previous definitions of variables that are *definitely* redefined by s
    - e.g., kill(s_1: x = e) is <x, *>

Slide courtesy: Prof. Milind Kulkarni

Generalization (Recap)

- **Direction of the analysis:**
  - How does information flow w.r.t. control flow?
- **Join operator:**
  - What happens at merge points? E.g. what operator to use Union or Intersection?
- **Transfer function:**
  - Define sets gen(b), kill(b), IN(b), OUT(b)
- **Initializations?**

Liveness Analysis and Reaching Definitions as LLVM Passes - Try it yourself

- Refer to code examples