Spring 2022

CS 424/624 Reliable Software Systems

Lecture 3: Static Analysis

Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

(Slides based on Stephen Chong's lecture notes)

Static Analysis Basics

- **Goal: analyze all *possible* behaviors of a program without running it**
- **Some basic static analysis questions:**
  - Where does the source of a variable come from?
  - What program locations use the value of a variable?
  - How does a variable's value propagate throughout the program?
- **For reliability purpose: check some rules during the analysis**
  - Does the program dereference a null pointer?
  - Does the program free all allocations?
  - Is every file handle closed?

Static Analysis Basics

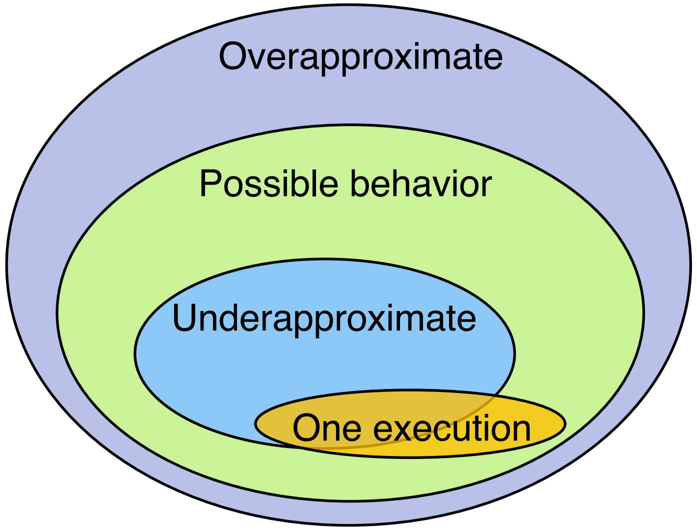
- **Different flavors:**
  - intra-procedural, inter-procedural
  - data flow, control flow
  - flow sensitive, path sensitive, context sensitive, field sensitive, etc.
- **Relies on compiler techniques**
  - Usually work on intermediate representation, e.g., static single assignment (SSA) form
    - Each variable is defined (assigned to) exactly once
    - But may be used multiple times
  - Popular tools: LLVM, Frama-C, Soot, FindBugs

Bad News: No Silver Bullet

- **No Perfect Static Analysis Method Exists**
  - Why?
  - the general problem of finding all possible run-time errors in an arbitrary program is undecidable: reducible to the halting problem
- **Each method makes trade-off between soundness and completeness**
  - Overapproximate or underapproximate the problem
  - Try to solve this simpler version

Sound, Complete Analysis?

- **A sound static analysis over-approximates the program behaviors**
  - guaranteed to identify all violations
  - but may report false positives
- **A complete static analysis under-approximates the program behaviors**
  - every reported violation is a true violation
  - but no guarantee all violations will be reported
- **Most existing bug detection static analyses are neither sound nor complete!**



Control-Flow Graph (CFG)

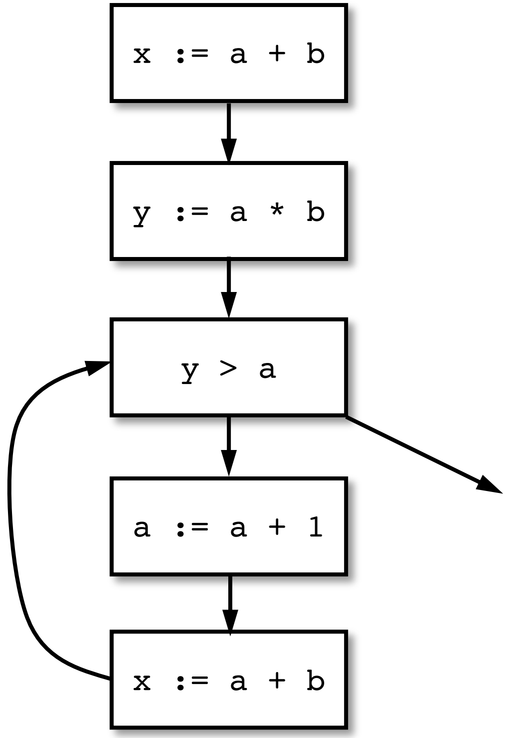
- **A control flow graph is a representation of a program that makes certain analyses (including dataflow analyses) easier**

- **A directed graph where**
  - Each node represents a statement
  - Edges represent control flow

- **Statements may be**
  - Assignments or `x := y op z` or `x := op z`
  - Copy statements `x := y`
  - Branches `goto L` or `if b then goto L`
  - Etc.

Control-flow Graph Example

```
x := a + b;

y := a * b;

while (y > a){

    a := a +1;

    x := a + b;

}
```

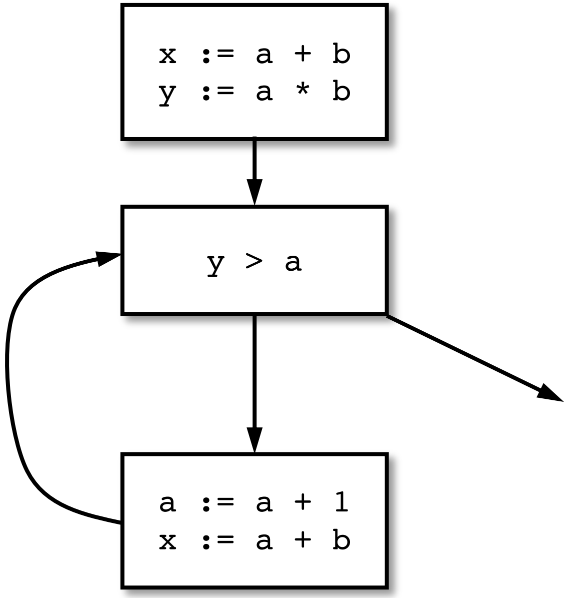


Variations on CFGs

- **Usually do not include declarations (*e.g.*, `int x;`) in the CFG**
  - but there is usually something in the implementation
- **May want a unique entry and exit point.**
- **May group statements into basic blocks**
  - A sequence of instructions with no branches into or out of the block.
    - i.e., execution starts only at the beginning of the block, and executes all of the block. Final statement in block may be a branch.

Control-Flow Graph with Basic Blocks

```
x := a + b;

y := a * b;

while (y > a){

    a := a +1;

    x := a + b;

}
```

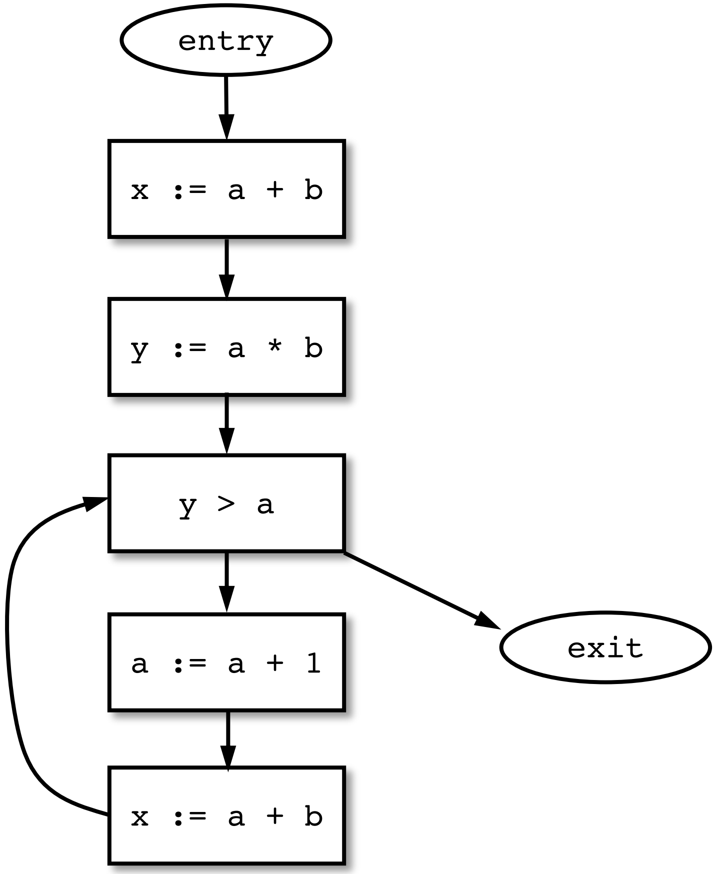


- **Can lead to more efficient implementations**
  - But more complicated to explain
- **We will use single-statement blocks in lecture**

CFG with Entry and Exit

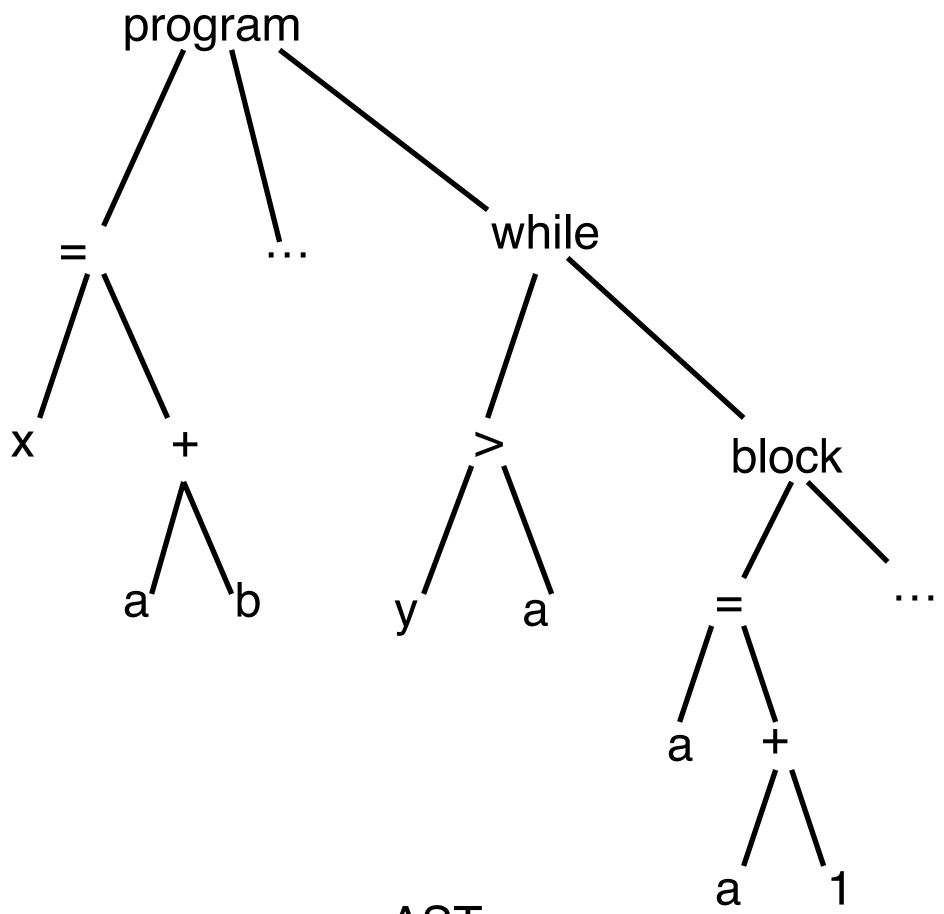
```
x := a + b;

y := a * b;

while (y > a){

    a := a +1;

    x := a + b;

}
```

- **All nodes without a normal predecessor should be pointed to by entry**
- **All nodes without a successor should point to exit**



CFG vs. AST (Abstract Syntax Tree)

- **CFGs are much simpler than ASTs**
  - fewer forms, less redundancy, only simple expressions
- **But, ASTs are a more faithful representation**
  - CFGs introduce temporaries
  - lose block structure of program
- **So for AST,**
  - easier to report error + other messages
  - easier to explain to programmer
  - easier to unparse to produce readable code



AST

Data Flow Analysis

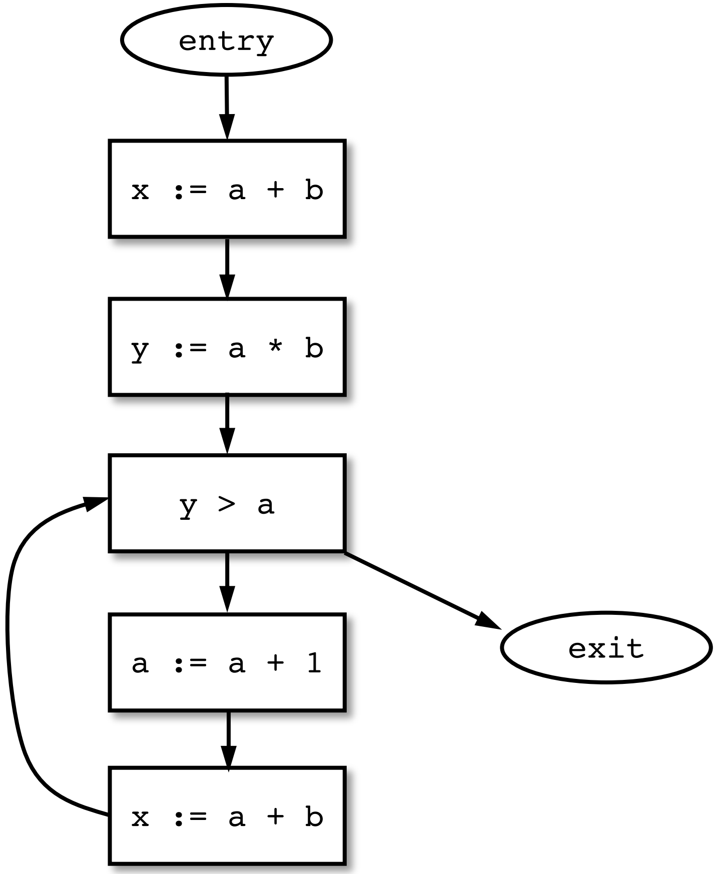
- **A framework for proving facts about program**
- **Reasons about lots of little facts**
- **Little or no interaction between facts**
  - works best on properties about how program computes
- **Based on all paths through program**
  - including infeasible paths
- **Let's consider some dataflow analyses**

Available Expressions

- **An expression `e = x op y` is available at a program point p, if**
  - `e` is computed on every path from the graph entry to node p, and
  - `e`*'s value* has not changed since the last time `e` was computed on the paths
    - i.e., there are no definitions of `x` or `y` since the most recent occurrence of `e` on the path
- **Available expression can be used to optimize code**
  - if an expression is available, it need not be recomputed
  - at least, if it is in a register somewhere

Data Flow Facts

- **Is expression e available?**
- **Facts:**
  - a + b is available
  - a * b is available
  - a + 1 is available
- **For each program point, we will compute which facts hold**

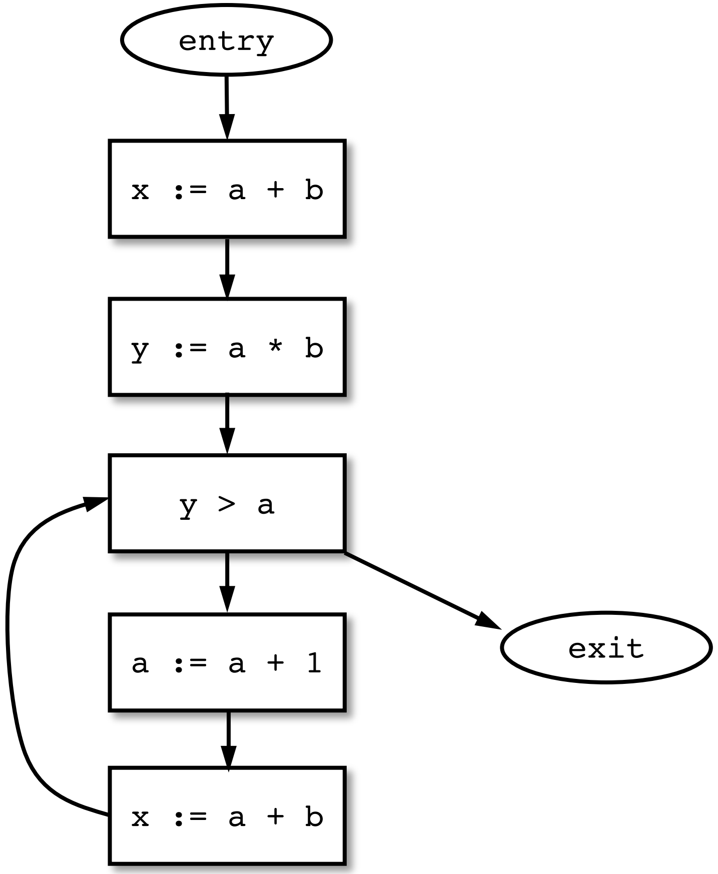


Gen and Kill

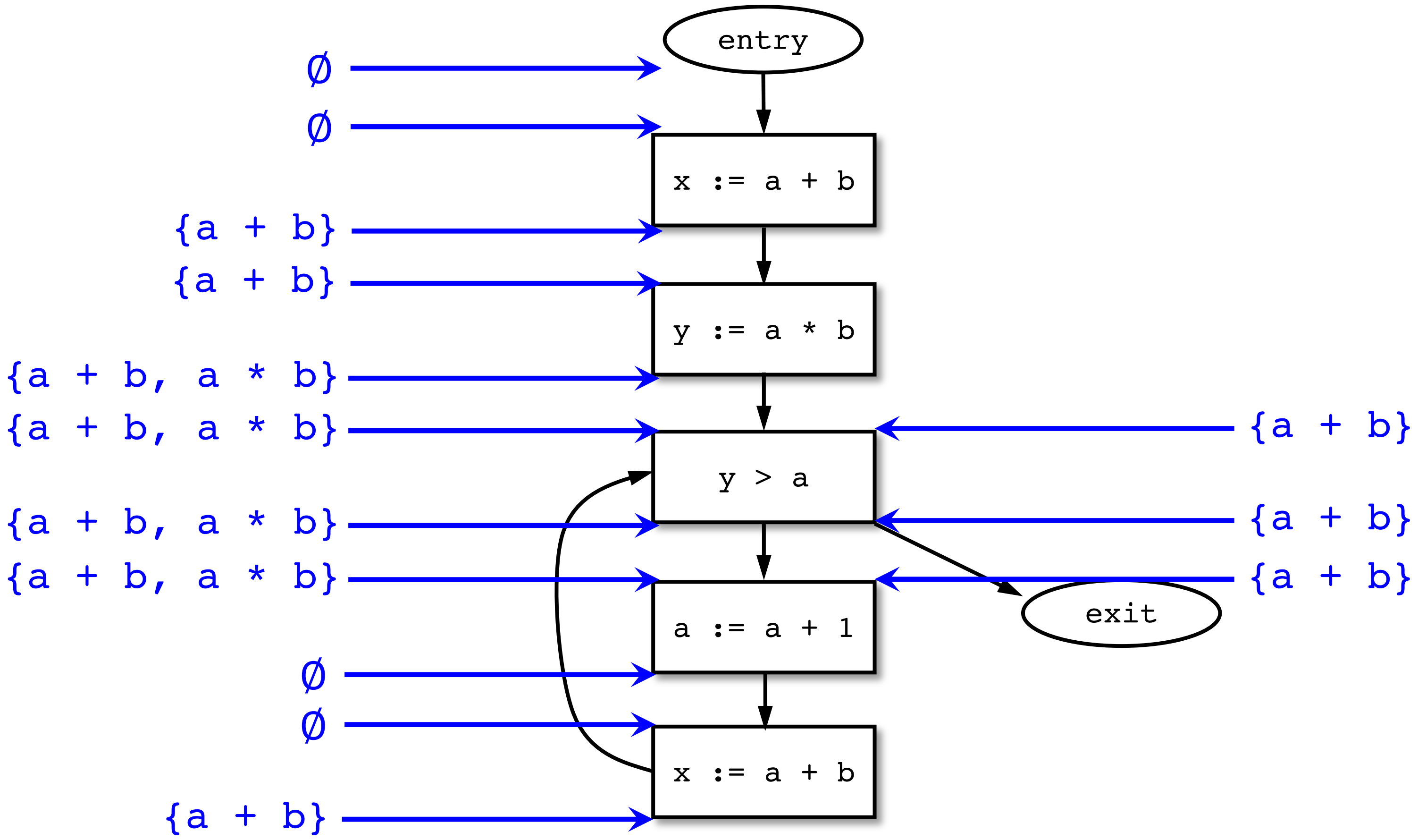
- **What is the effect of each statement on the facts?**

stmt	gen	kill
`x := a + b`	`a + b`	
`y := a * b`	`a * b`	
`y > a`		

`a := a + 1`		`a + 1`
		`a + b`
		`a * b`



Computing Available Expressions



stmt	gen	kill
x := a + b	a + b	
y := a * b	a * b	
y > a		
a := a + 1		a + 1 a + b a * b

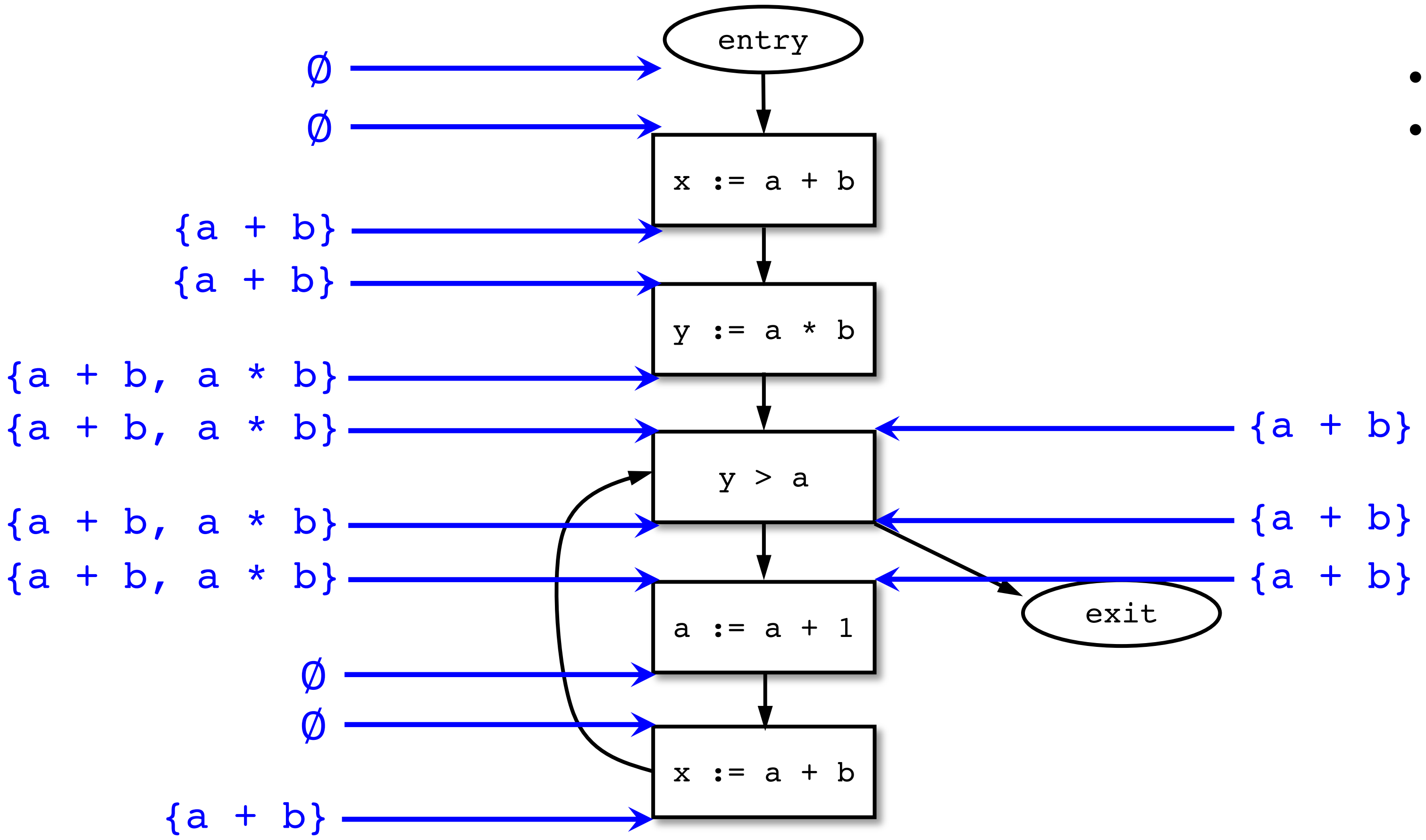
Terminology

- **A *join point* is a program point where two branches meet**
- **Available expressions is a *forward, must problem***
  - *Forward* = data Flow from in to out
  - *Must* = at join point, property must hold on all paths that are joined

Data Flow Equations

- **Let s be a statement**
  - succ(s) = {immediate successor statements of s}
  - pred(s) = {immediate predecessor statements of s}
  - In(s) = facts that hold just before executing s
  - Out(s) = facts that hold after executing s
- $\mathrm{In}(s) = \cap_{s' \in \mathrm{pred}(s)} \mathrm{Out}(s')$
- $\mathrm{Out}(s) = \mathrm{Gen}(s) \cup (\mathrm{In}(s) - \mathrm{Kill}(s))$
- **These are also called transfer functions**

Computing Available Expressions



- $\mathrm{In}(s) = \cap_{s' \in \mathrm{pred}(s)} \mathrm{Out}(s')$
- $\mathrm{Out}(s) = \mathrm{Gen}(s) \cup (\mathrm{In}(s) - \mathrm{Kill}(s))$

stmt	gen	kill
`x := a + b`	`a + b`	
`y := a * b`	`a * b`	
`y > a`		

`a := a + 1`		`a + 1`
		`a + b`
		`a * b`

Live Variables

- **A variable v is *live at a program* point p if**
  - v will be used on some execution path originating from p before v is overwritten
- **Optimization**
  - If a variable is not live, no need to keep it in a register
  - If a variable is dead at assignment, can eliminate assignment

Data Flow Equations

- **Available expressions is a forward must analysis**
  - propagate facts in same direction as control flow
  - expression is available if available on all paths

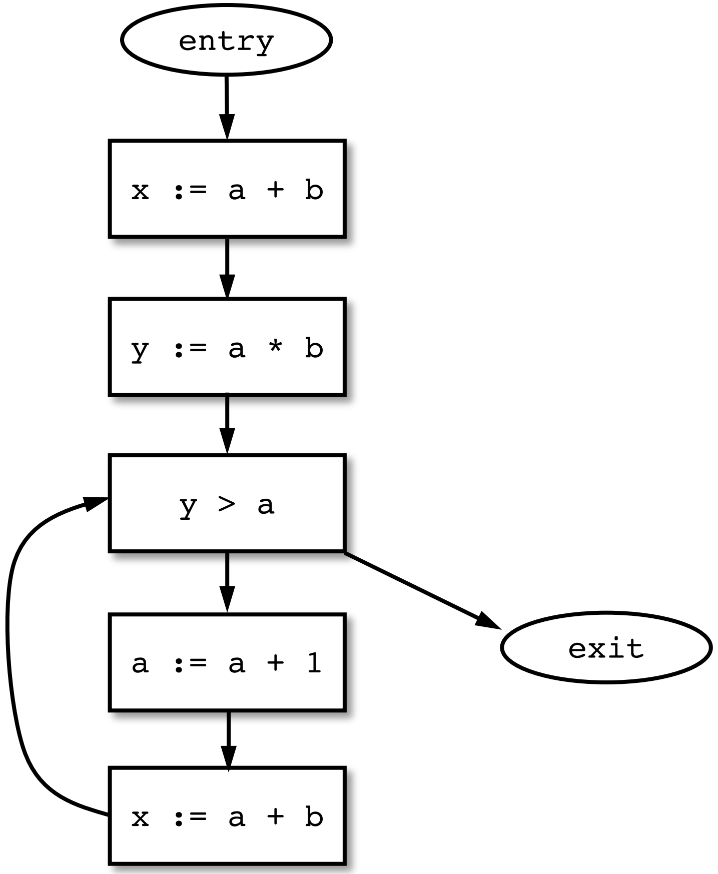
- $\mathrm{In}(s) = \cap_{s' \in \mathrm{pred}(s)} \mathrm{Out}(s')$
- $\mathrm{Out}(s) = \mathrm{Gen}(s) \cup (\mathrm{In}(s) - \mathrm{Kill}(s))$

- **Liveness is a backward may analysis**
  - to know if variable is live, need to look at future uses
  - variable is live if available on some path
- $\mathrm{Out}(s) = \cup_{s' \in \mathrm{succ}(s)} \mathrm{In}(s')$
- $\mathrm{In}(s) = \mathrm{Gen}(s) \cup (\mathrm{Out}(s) - \mathrm{Kill}(s))$

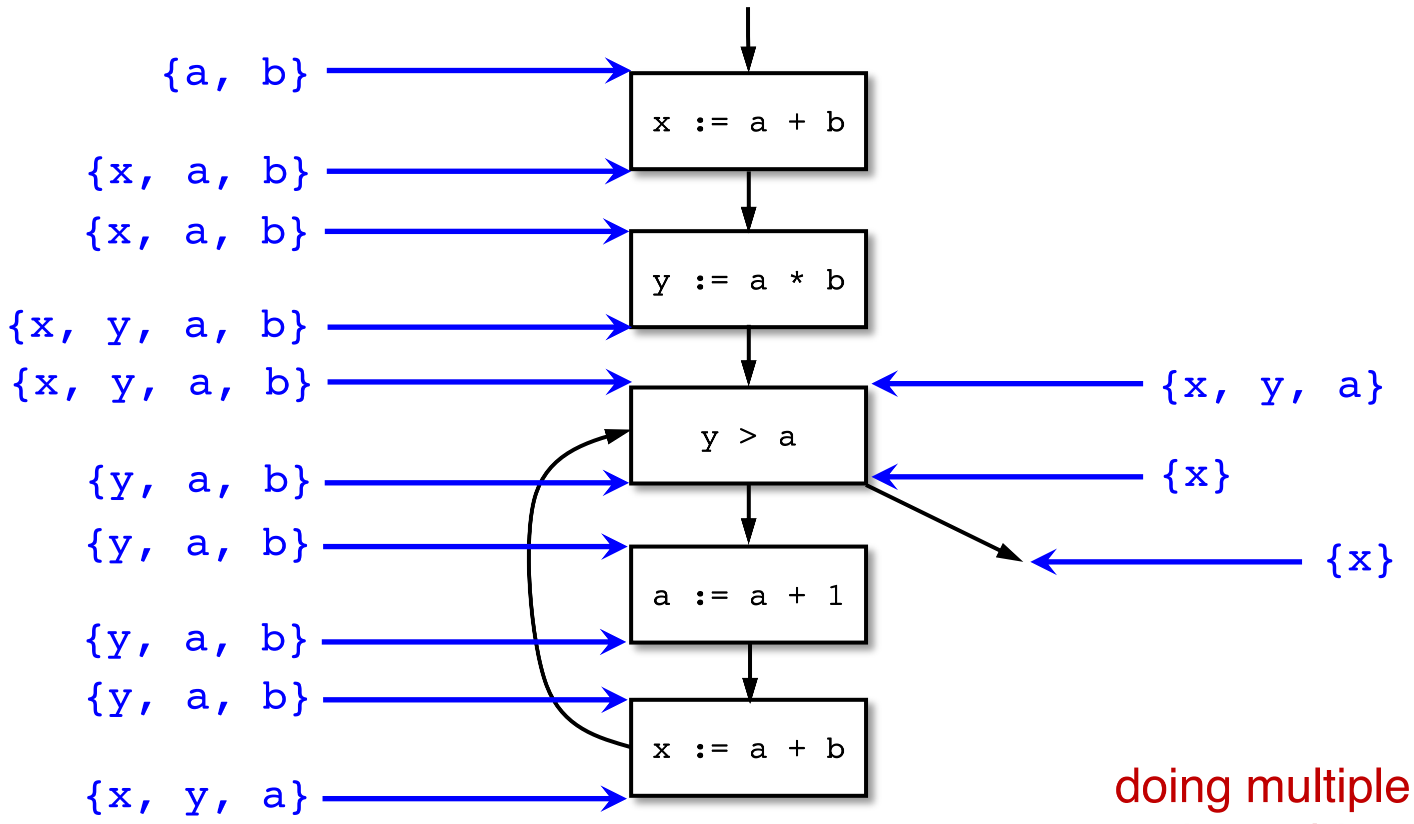
Gen and Kill

- **What is the effect of each statement on the facts?**

stmt	gen	kill
`x := a + b`	`a, b`	`x`
`y := a * b`	`a, b`	`y`
`y > a`	`a, y`	
`a := a + 1`	`a`	`a`



Computing Live Variables



- $\mathrm{Out}(s) = \bigcup_{s' \in \mathrm{succ}(s)} \mathrm{In}(s')$
- $\mathrm{In}(s) = \mathrm{Gen}(s) \cup (\mathrm{Out}(s) - \mathrm{Kill}(s))$

stmt	gen	kill
`x := a + b`	a, b	x
`y := a * b`	a, b	y
`y > a`	a, y	
`a := a + 1`	a	a

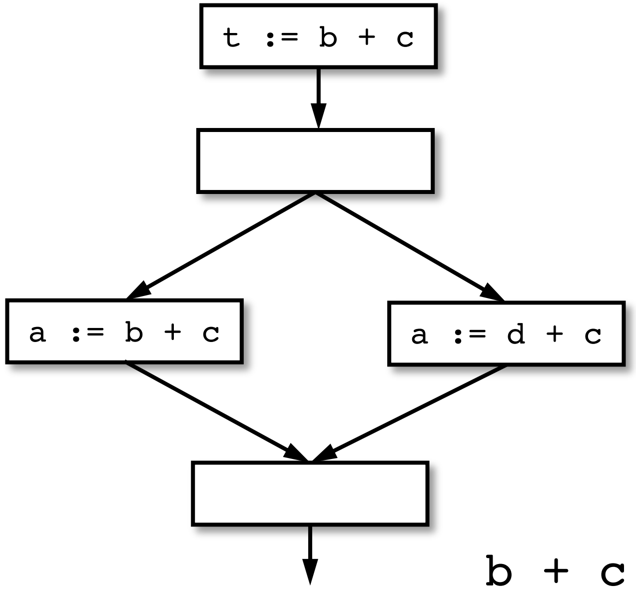
doing multiple iterations
until reaching fixed point

Very Busy Expressions

- **An expression `e` is very busy at point `p` if**
  - On every path from `p`, `e` is evaluated before the value of `e` is changed
  - *i.e.*, it is guaranteed that `e` will be computed at some time in the future
- **Optimization**
  - Can hoist very busy expression computation
  - Very busy expressions are ideal candidates for invariant loop motion
    - If an expression, invariant in a loop, is also very busy, we know it must be used in the future, and hence evaluation outside the loop must be worthwhile
- **What kind of problem?**
  - Forward or backward?
  - May or must?

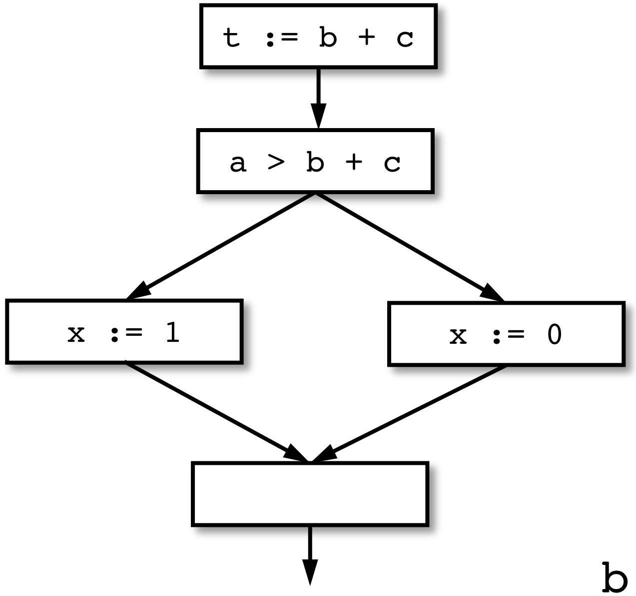
Examples

```
for (...) {
  if (...)
    a=b+c;
  else
    a=d+c;
}
```



`b + c` is *not* very busy at loop entrance

```
for (...) {
  if (a>b+c)
    x=1;
  else
    x=0;
}
```



`b + c` is very busy at loop entrance

Reaching Definitions

- **A *definition of a variable* `v` is an assignment to `v`**
- **A definition of variable `v` reaches point `p` if**
  - There is no intervening assignment to `v`
- **Also called *def-use* information**
- **What kind of problem?**
  - Forward or backward?
  - May or must?

Space of Data Flow Analyses

	May	**Must**
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

- **Most data flow analyses can be classified this way**
  - A few don't fit: bidirectional
- **Lots of literature on data flow analysis**

Forward Must Data Flow Algorithm

```
Out(s) := Gen(s) for all statements s
W := {all statements}                              (worklist)
repeat {
    take s from W
    In(s) := $\cap_{s' \in pred(s)}$ Out(s')
    temp := Gen(s) ∪ (In(s) – Kill(s))
    if (temp != Out(s)) {
        Out(s) := temp
        W := W ∪ succ(s)
    }
} until W = ∅
```

Will the algorithm terminate?

Practical Implementation

- **Data flow facts are assertions that are true or false at a program point**

- **Can represent set of facts as bit vector**
  - Fact i represented by bit i
  - Intersection=bitwise and, union=bitwise or, etc

- **“Only” a constant factor speedup**

- **But very useful in practice**

Basic Blocks

- **A basic block is a sequence of statements such that**
  - No branches to any statement except the first
  - No statement in the block branches except the last
- **In practical data flow implementations**
  - Compute Gen/Kill for each basic block
    - Compose transfer functions
  - Store only In/Out for each basic block
  - Typical basic block is about 5 statements

Flow Sensitivity

- **Data flow analysis is flow sensitive**
  - The order of statements is taken into account
  - I.e., we keep track of facts per program point
- **Alternative: Flow-insensitive analysis**
  - Analysis the same regardless of statement order
  - Standard example: types describe facts that are true at all program points
    - /*x:int*/ x:=… /*x:int*/

A Problem...

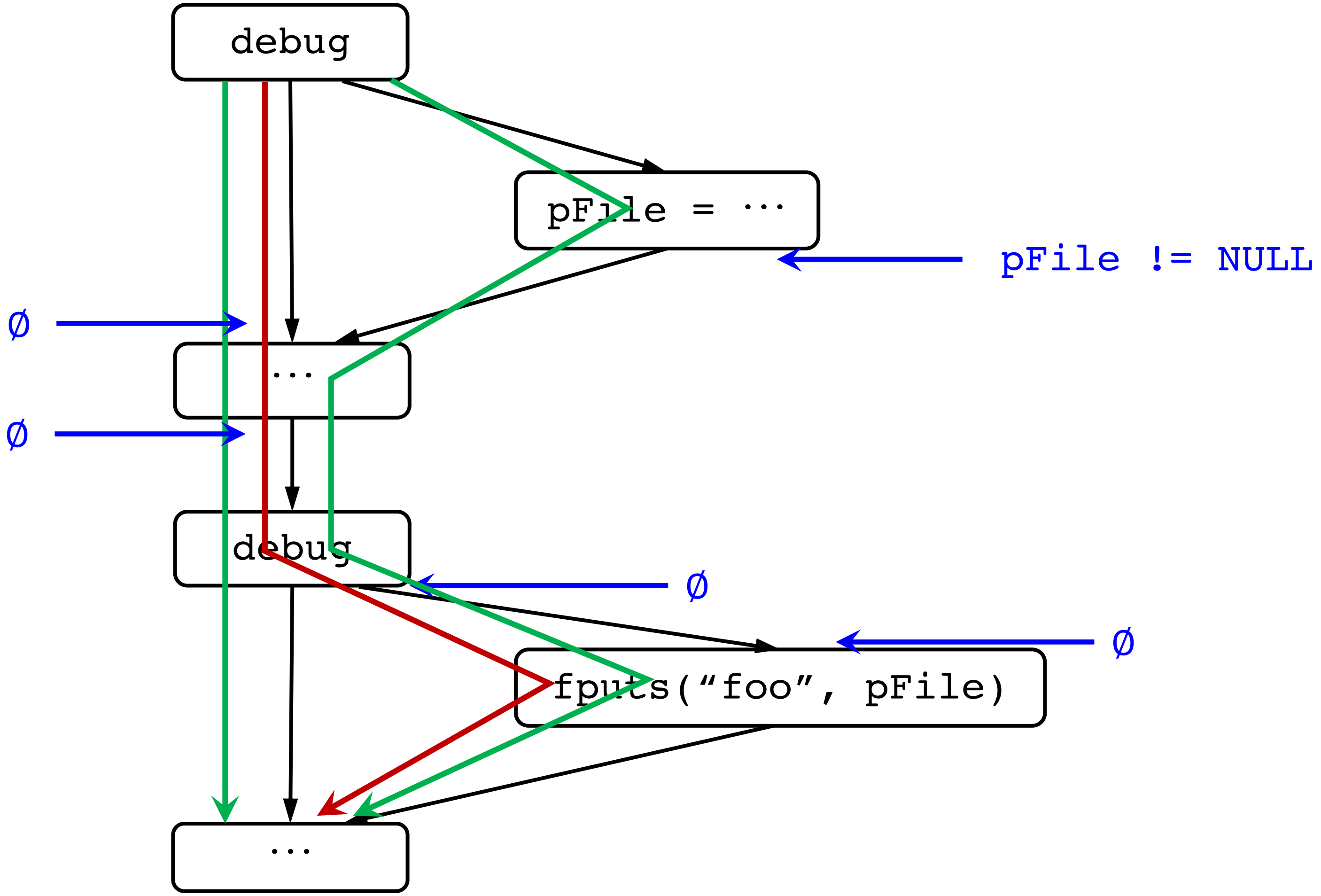
- **Consider following program**

```
FILE *pFile = NULL;
if (debug) {
  pFile = fopen(“debuglog.txt”, “a”)
}
…
if (debug) {
  fputs(“foo”, pFile);
}
```

- **Can `pFile` be `NULL` when used for `fputs`?**
- **What dataflow analysis could we use to determine if it is?**

Path Sensitivity

```
FILE *pFile = NULL;
if (debug) {
  pFile = fopen(…)
}
…
if (debug) {
  fputs("foo", pFile);
}
```



Path Sensitivity

- **A path-sensitive analysis tracks data flow facts depending on the path taken**
  - Path often represented by which branches of conditionals taken
- **Can reason more accurately about correlated conditionals (or dependent conditionals) such as in previous example**
- **How can we make a path sensitive analysis**
  - Could do a dataflow analysis where we track facts for each possible path
  - But exponentially many paths make it difficult to scale

Data Flow Analysis and Heap

- **Data Flow is good at analyzing local variables**
  - But what about values stored in the heap?
- **Not modeled in traditional data flow**
- **In practice: *x := e**
  - Assume all data flow facts killed (!)
  - Or, assume write through x may affect any variable whose address has been taken
- **In general, hard to analyze pointers**

Terminology Review

- Must vs. May
- (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Path-sensitive vs Path-insensitive

More Terminology

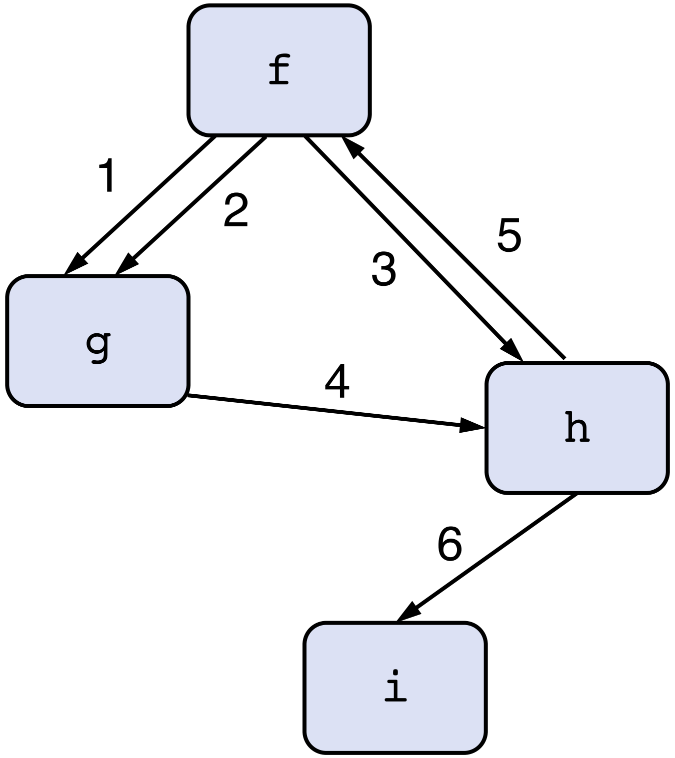
- **An analysis that models only a single function at a time is intra-procedural**
- **An analysis that takes multiple functions into account is inter-procedural**
- **An analysis that takes the whole program into account is whole program**
- **An inter-procedural analysis that considers the calling context when analyzing the target of a function call is context-sensitive**
  - otherwise, it is context-insensitive

Call Graph

- **Inter-procedural analysis uses calling relationships among multiple procedures**
  - Enables more precise analysis information
- **First problem: how do we know what procedures are called from where?**
  - Especially difficult in higher-order languages, languages where functions are values
- **Let’s assume we have a (static) call graph**
  - Indicates which procedures can call which other procedures, and from which program points.

Call Graph Example

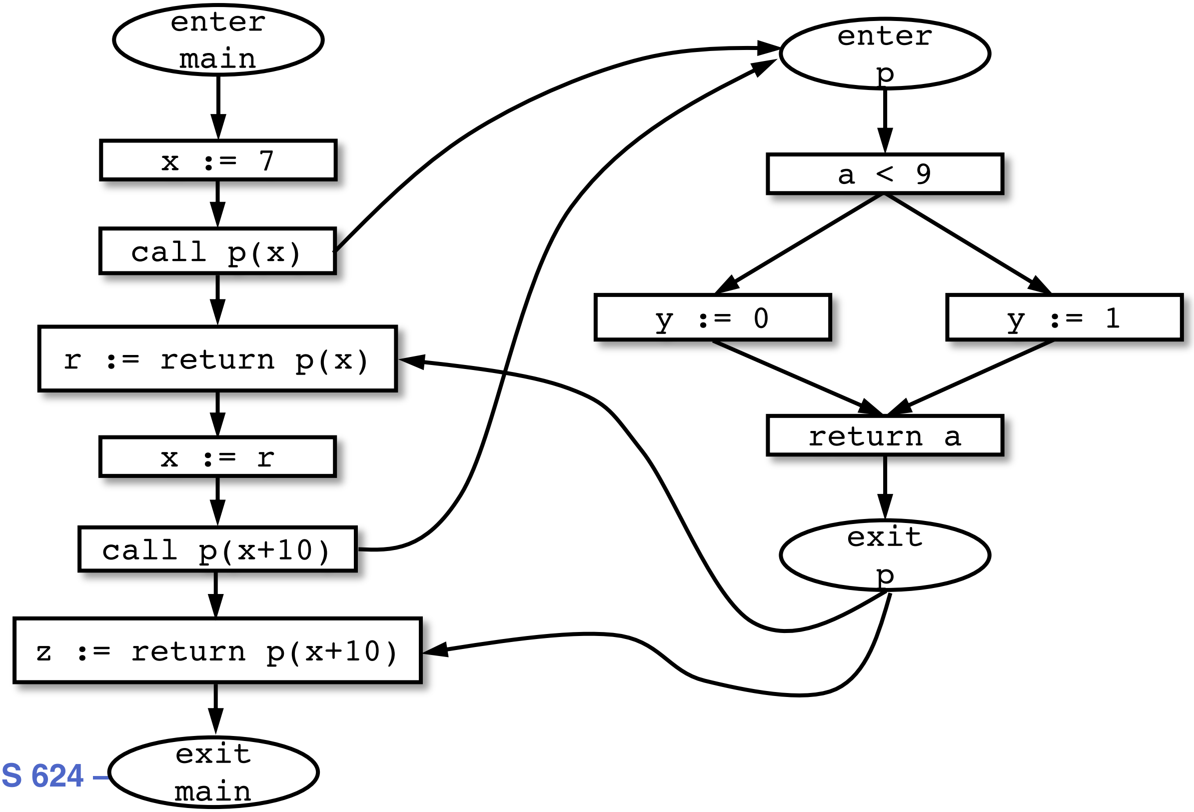
```
f() {
 1: g();
 2: g();
 3: h();
}

g() {
 4: h();
}

h() {
 5: f();
 6: i();
}

i() { … }
```



Inter-procedural Dataflow Analysis

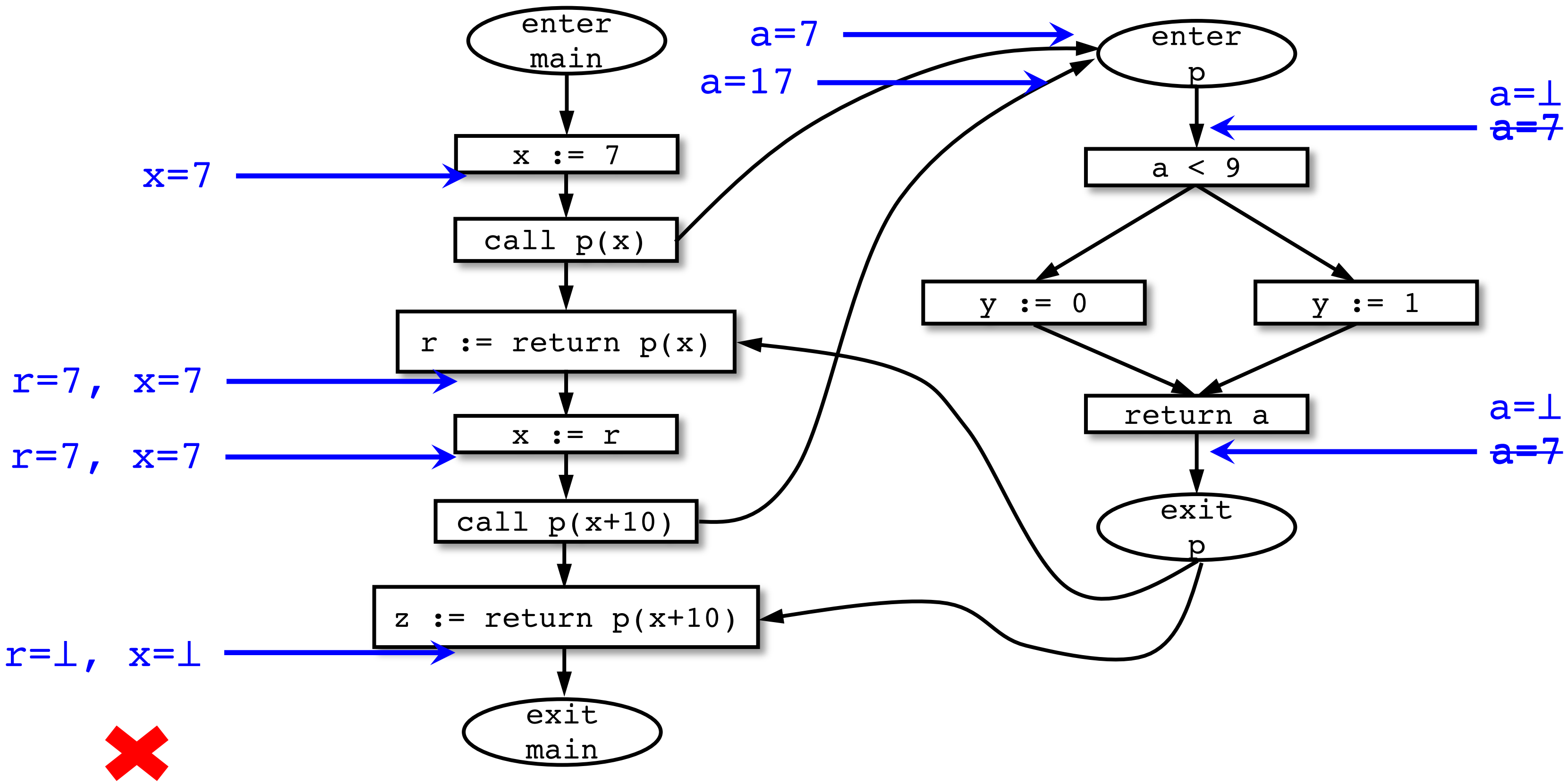
- **How do we deal with procedure calls?**
- **Obvious idea: make one big CFG**

```
main() {
 x := 7;
 r := p(x);
 x := r;
 z := p(x + 10);
}

p(int a) {
 if (a < 9)
   y := 0;
 else
   y := 1;
 return a;
}
```



Example

```
main() {
 x := 7;
 r := p(x);
 x := r;
 z := p(x + 10);
}

p(int a) {
 if (a < 9)
   y := 0;
 else
   y := 1;
 return a;
}
```

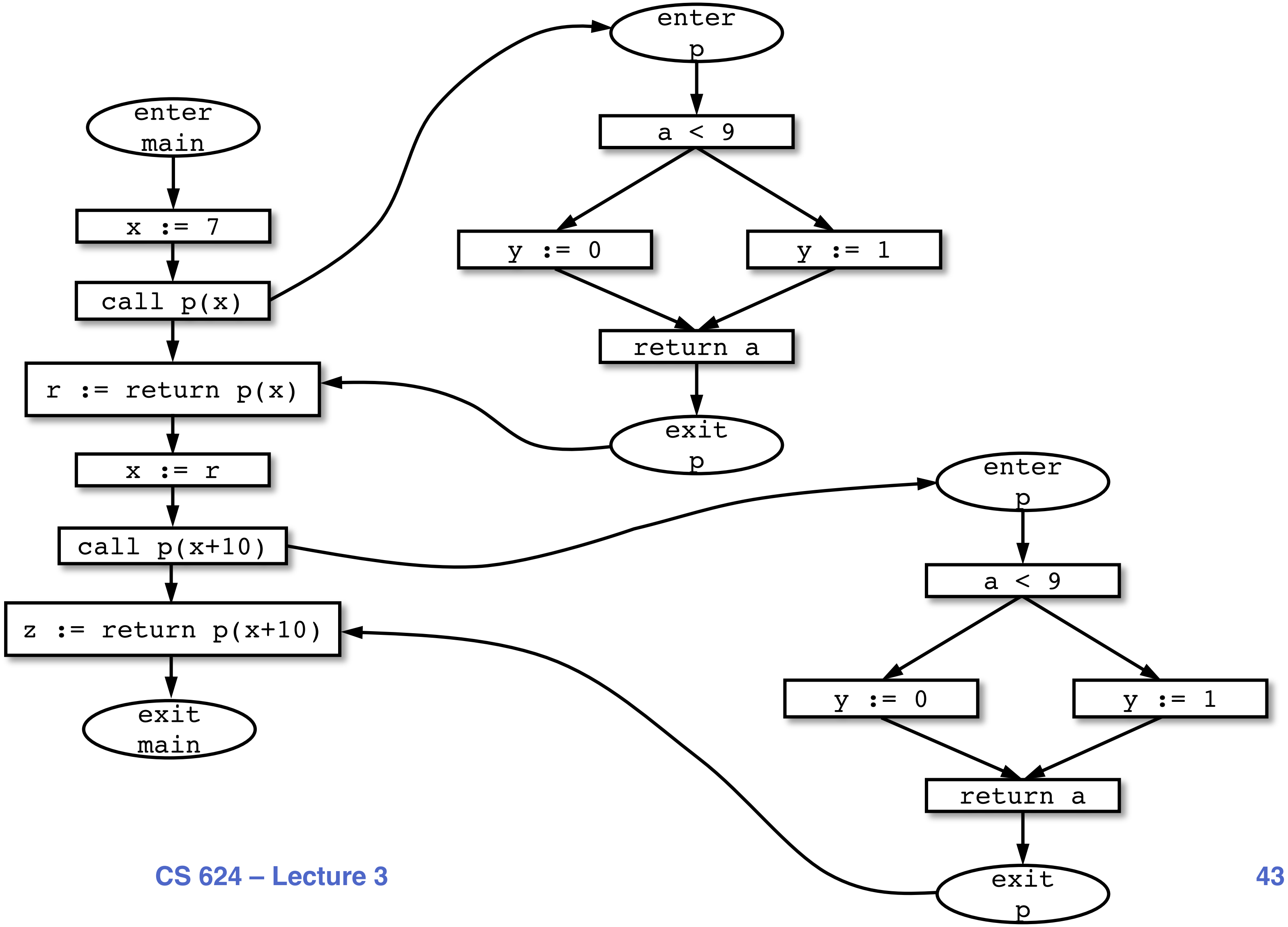


Context Sensitivity

- **Problem: dataflow facts from one call site “tainting” results at other call site**
  - p analyzed with merge of dataflow facts from all call sites
- **How to address?**

Inlining

- **Inlining**
  - Use a new copy of a procedure's CFG at each call site
- **Concerns?**
  - May be expensive! Exponential increase in size of CFG
    - p() { q(); q(); } q() { r(); r() } r() { … }
  - What about recursive procedures?



Context Sensitivity

- **Solution: make a finite number of copies**
- **Use context information to determine when to share a copy**
  - Results in a context-sensitive analysis
- **Choice of what to use for context will produce different tradeoffs between precision and scalability**
- **Common choice: approximation of call stack**