HARVARD
School of Engineering
and Applied Sciences

Interprocedural Analysis

CS252r Spring 2011

Procedures

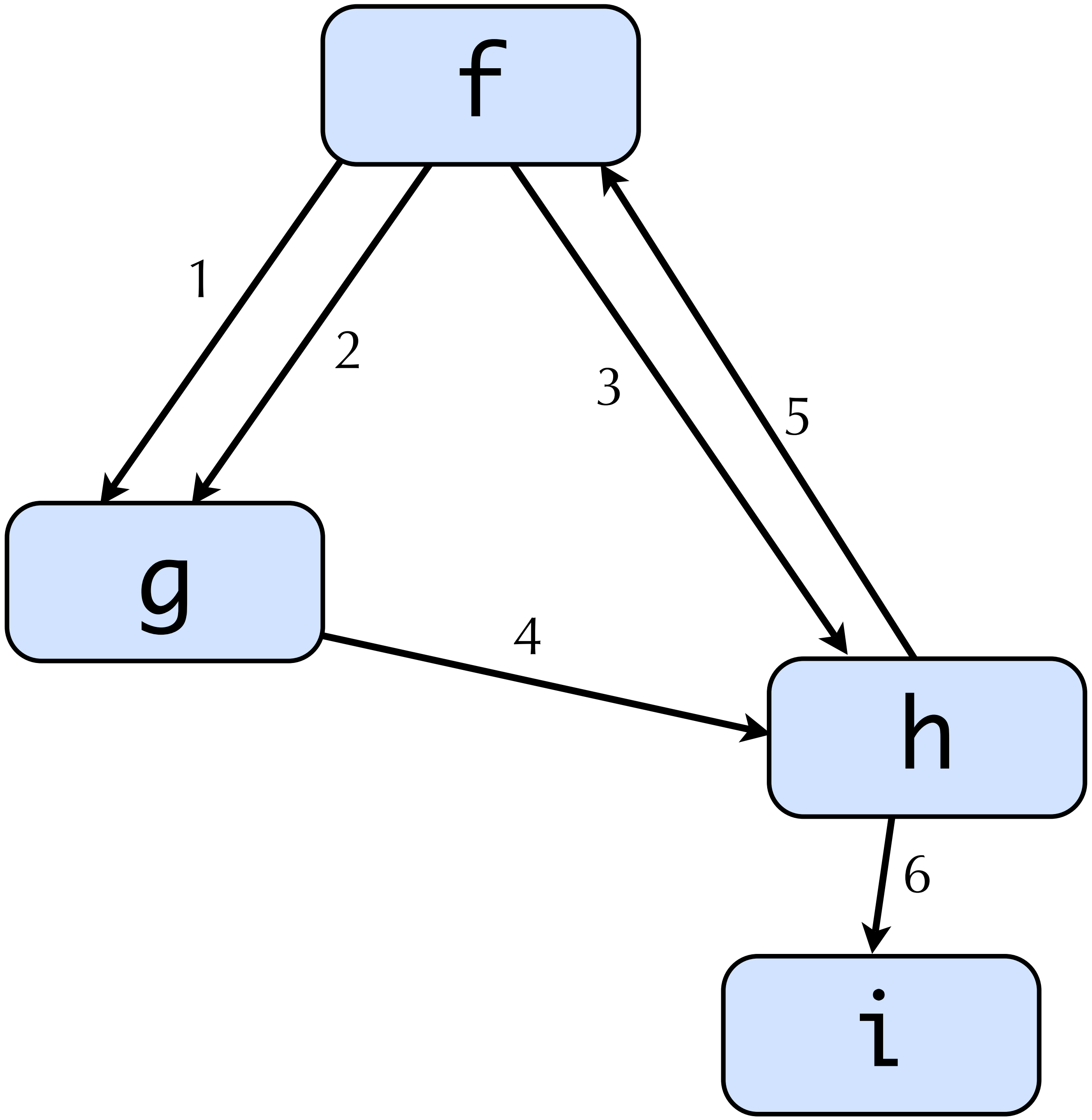
- So far looked at **intraprocedural** analysis: analyzing a single procedure
- **Interprocedural analysis** uses calling relationships among procedures
    - Enables more precise analysis information

Call graph

- First problem: how do we know what procedures are called from where?
  - Especially difficult in higher-order languages, languages where functions are values
  - We'll ignore this for now, and return to it later in course…
- 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

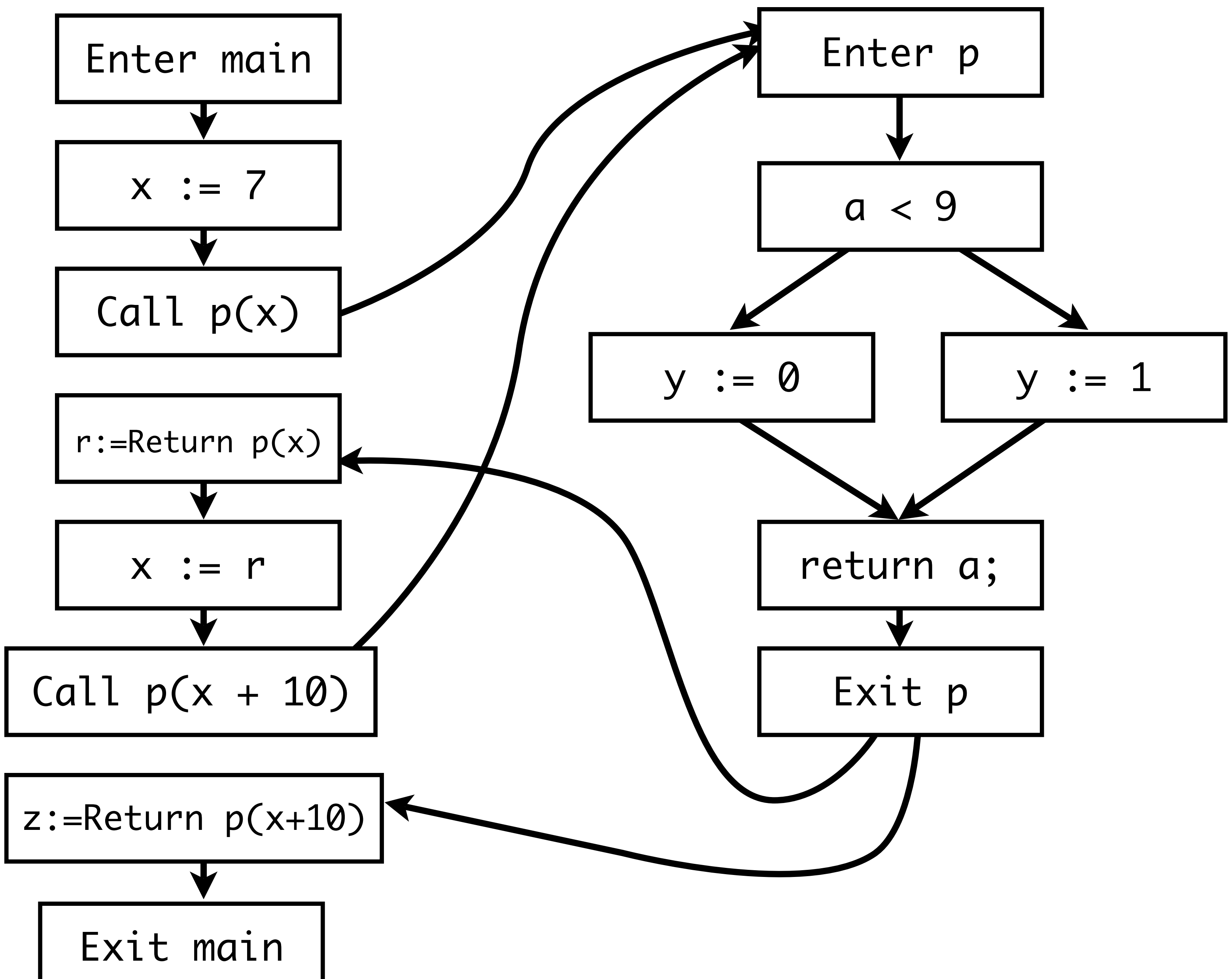
```
f() {
    1:  g();
    2:  g();
    3:  h();
}

g() {
    4: h();
}

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

i() { … }
```



Interprocedural dataflow analysis

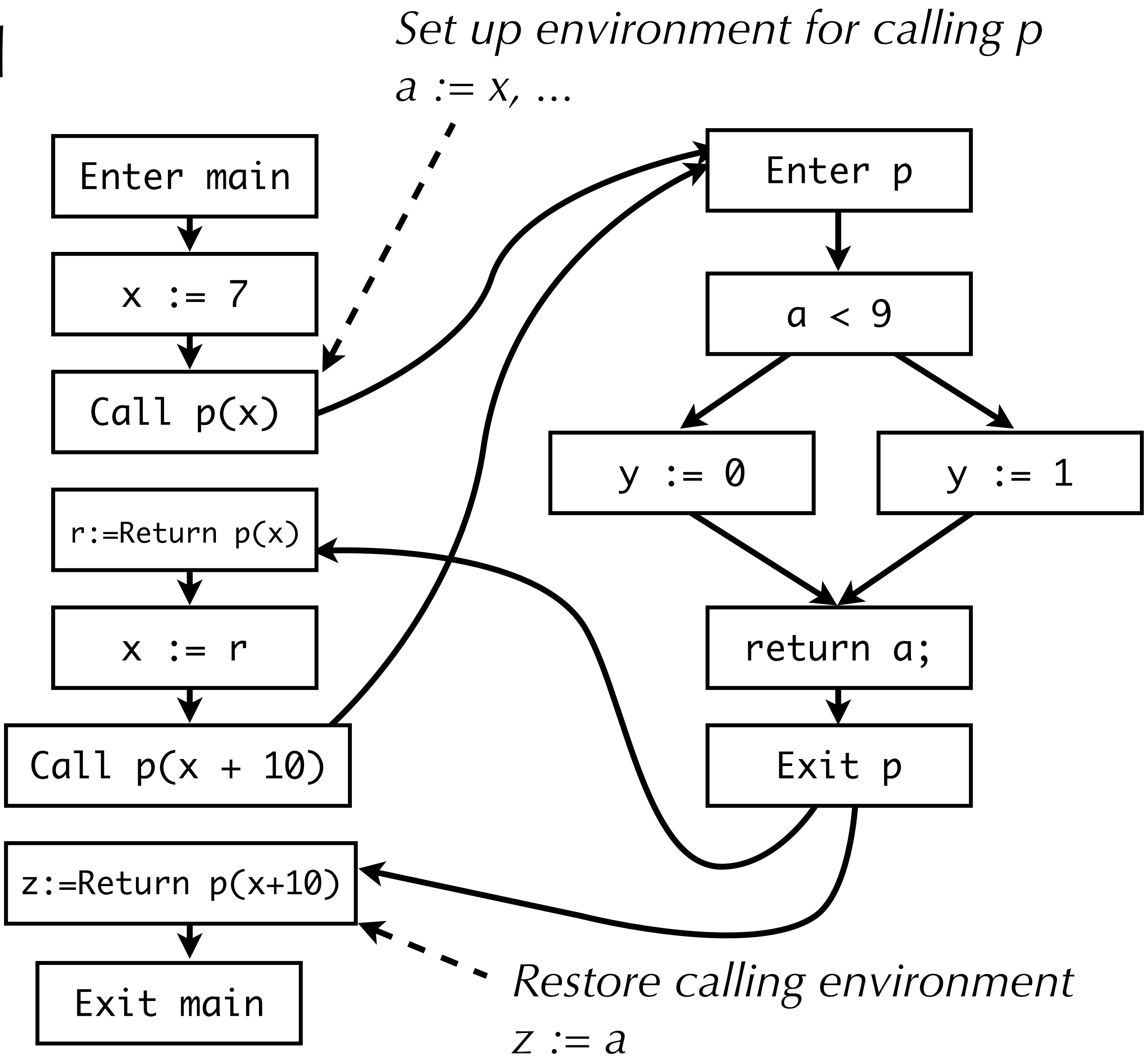
- How do we deal with procedure calls?
- Obvious idea: make one big CFG

```
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;
}
```

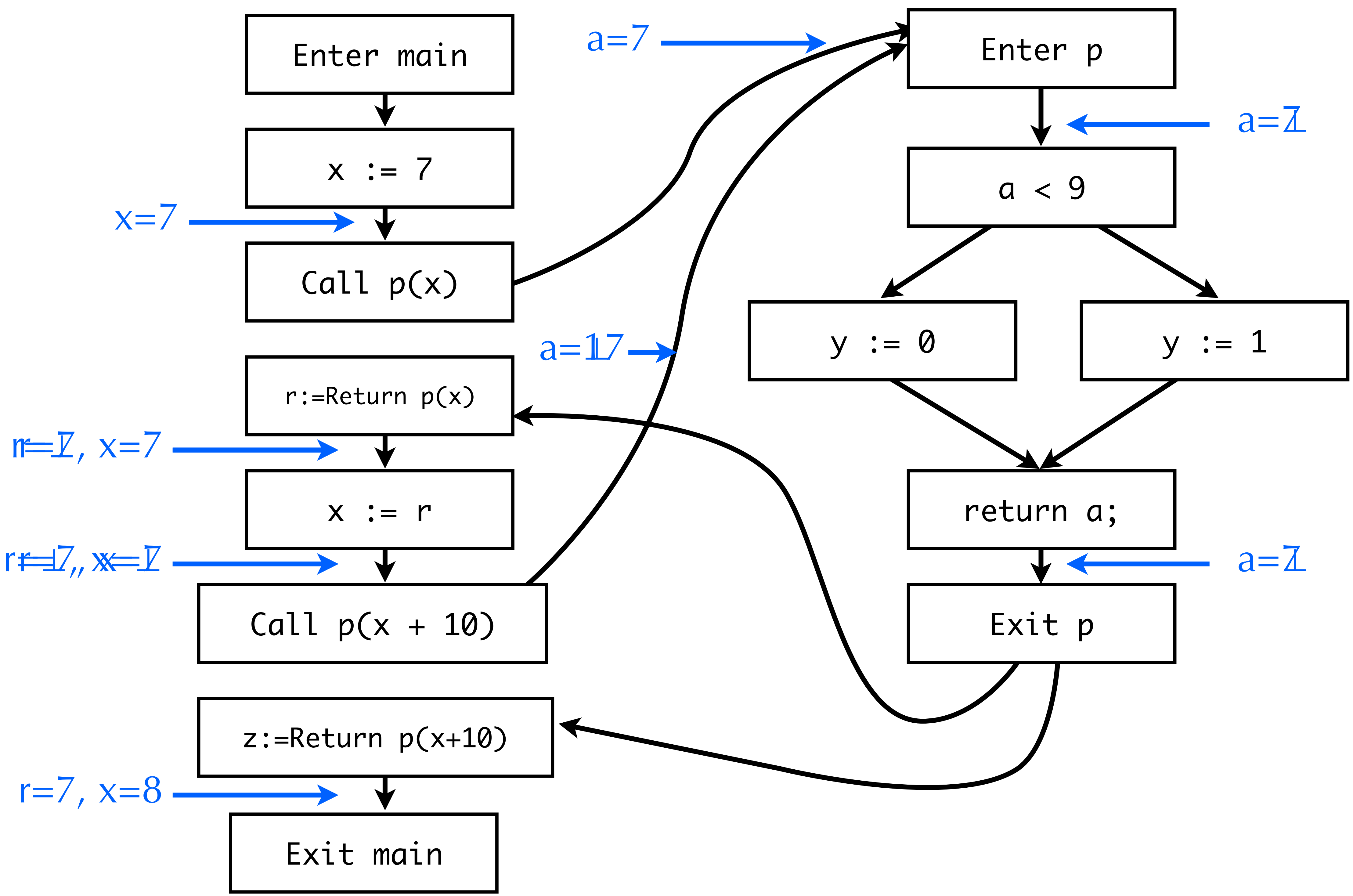


Interprocedural CFG

- CFG may have additional nodes to handle call and returns
  - Treat arguments, return values as assignments
- Note: a local program variable represents multiple locations



Example



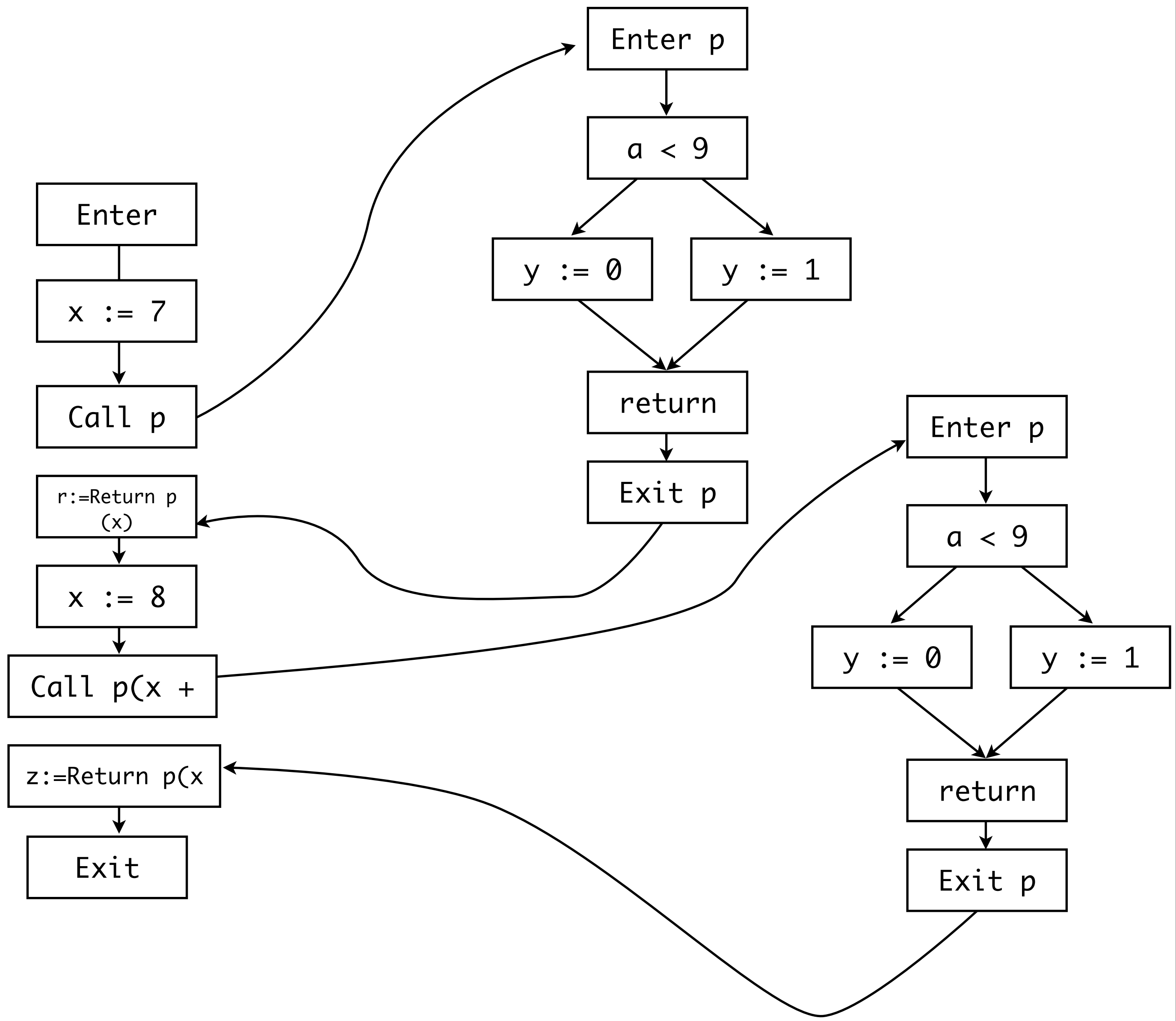
Enter main
x := 7
x=7
Call p(x)
r:=Return p(x)
x := r
Call p(x + 10)
z:=Return p(x+10)
r=7, x=8
Exit main
a=7
Enter p
a < 9
y := 0
y := 1
return a;
Exit p

Invalid paths

- 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
- Problems? Concerns?
  - May be expensive! Exponential increase in size of CFG
    - p() { q(); q(); }   q() { r(); r() } r() { … }
  - What about recursive procedures?
    - p(int n) { … p(n-1); … }
    - More generally, cycles in the call graph

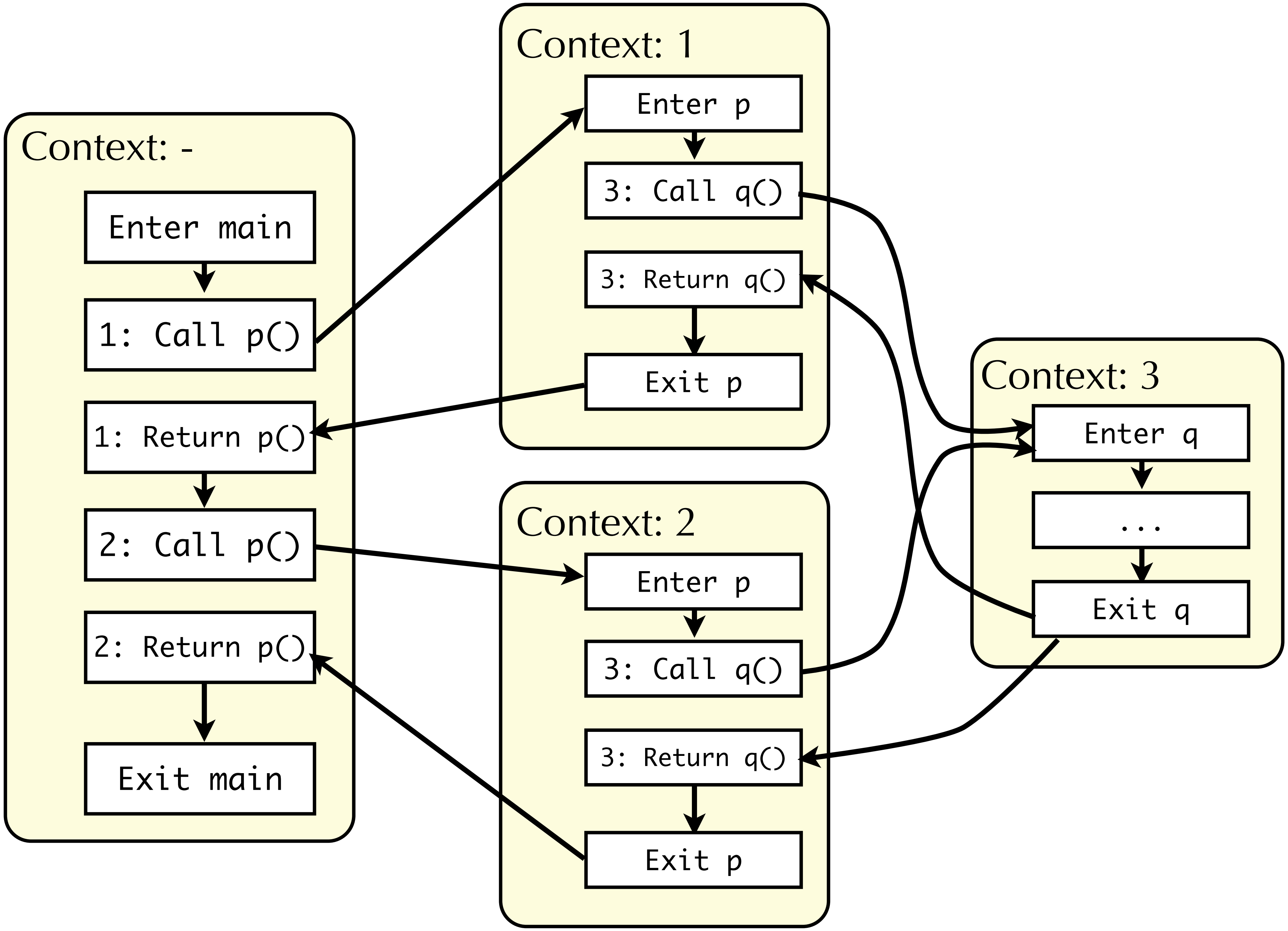


Context sensitivity

- 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

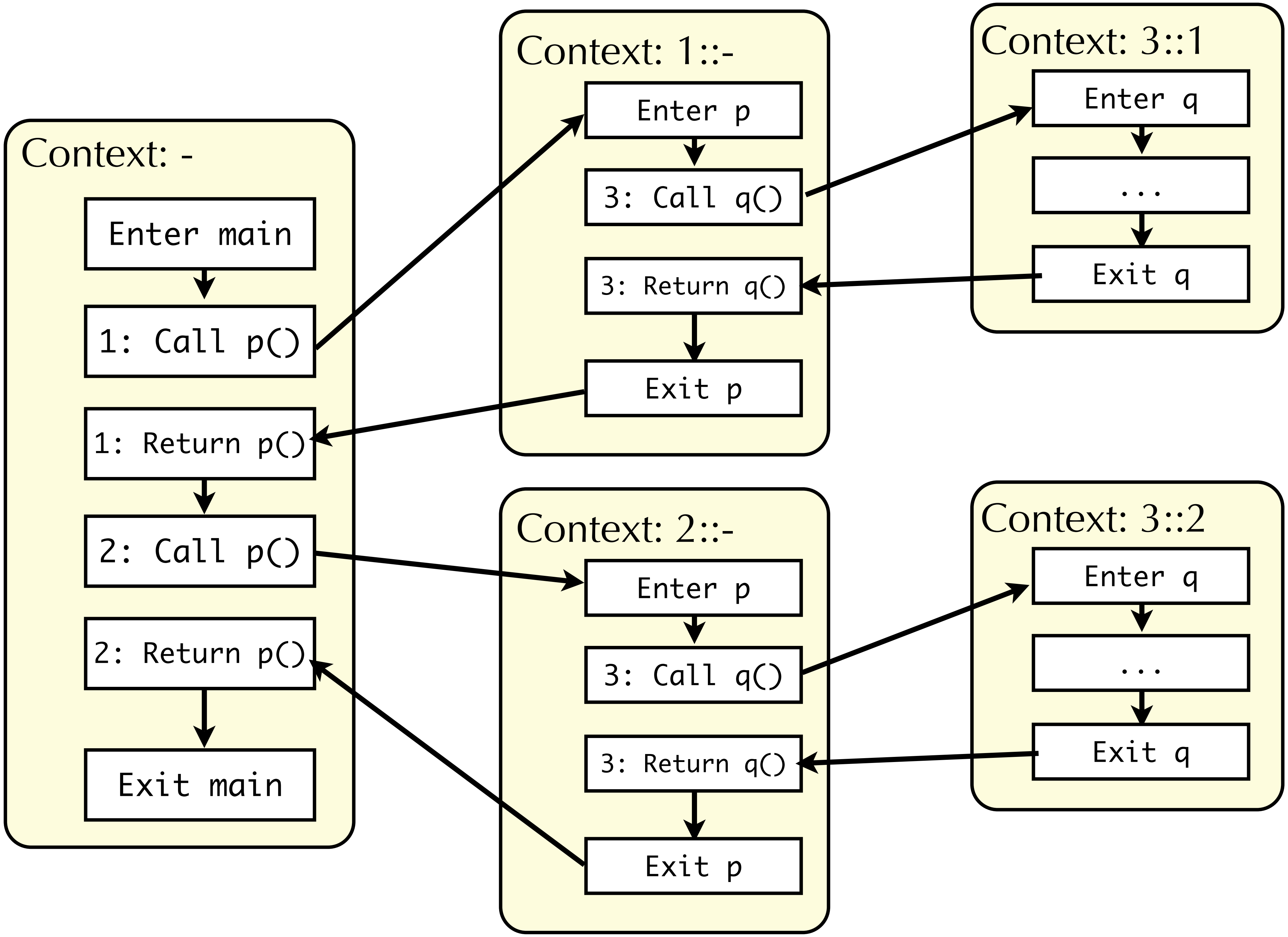
Context sensitivity example

```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```



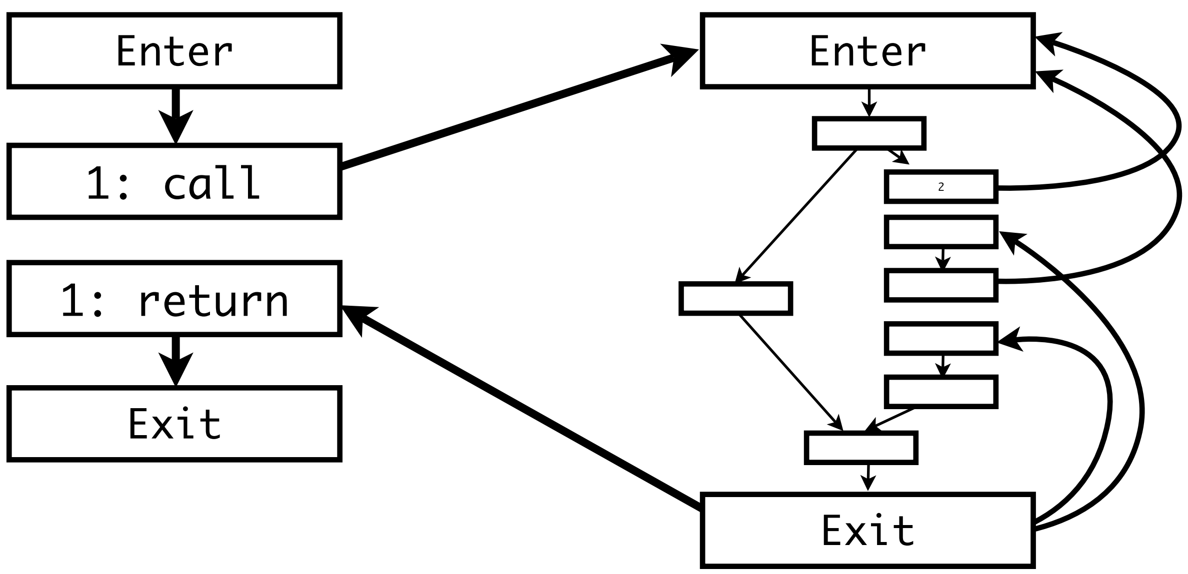
Context sensitivity example

```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```



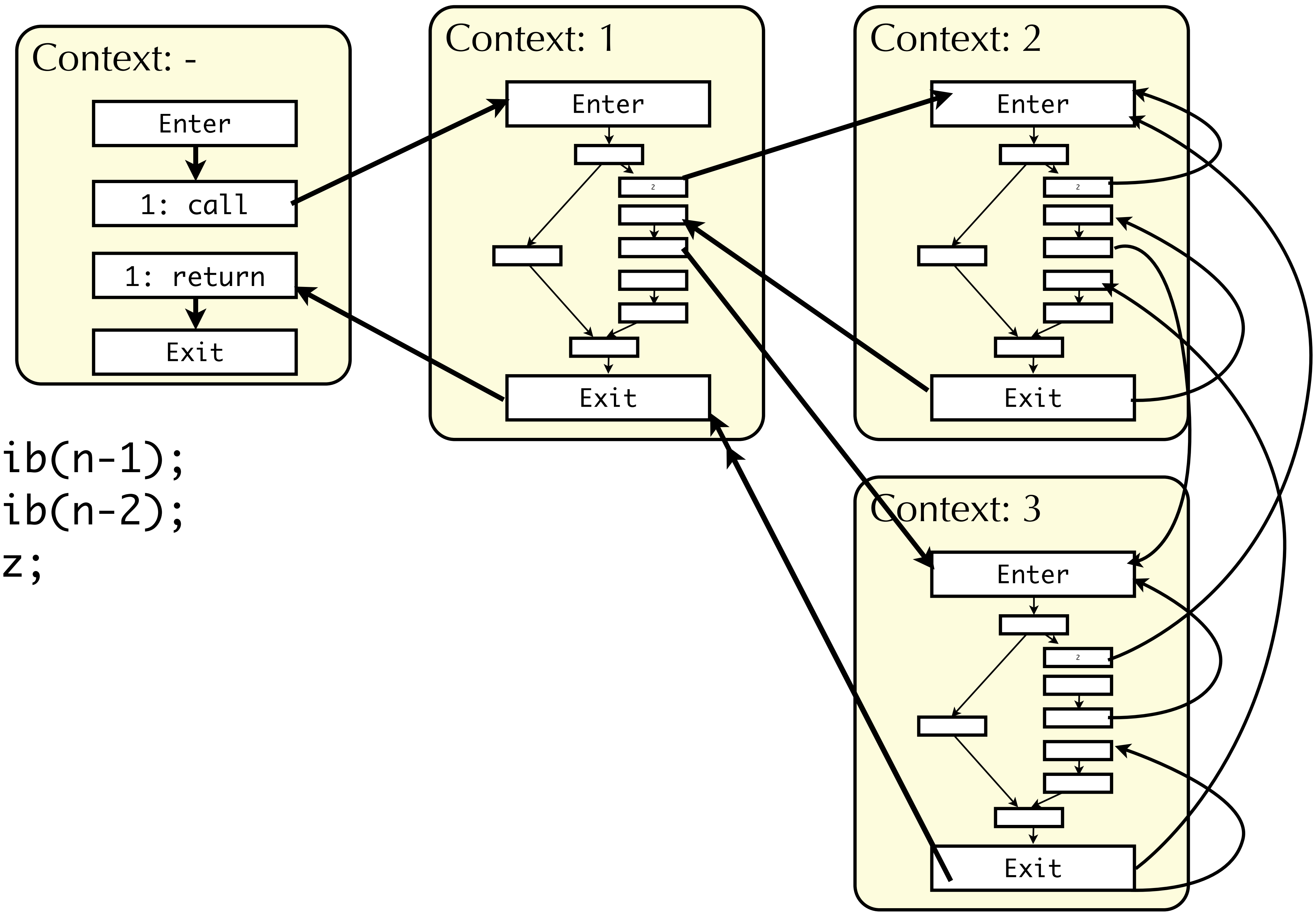
Fibonacci: context insensitive

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
    2: y := fib(n-1);
    3: z := fib(n-2);
       x:= y+z;
  return x;
}
```



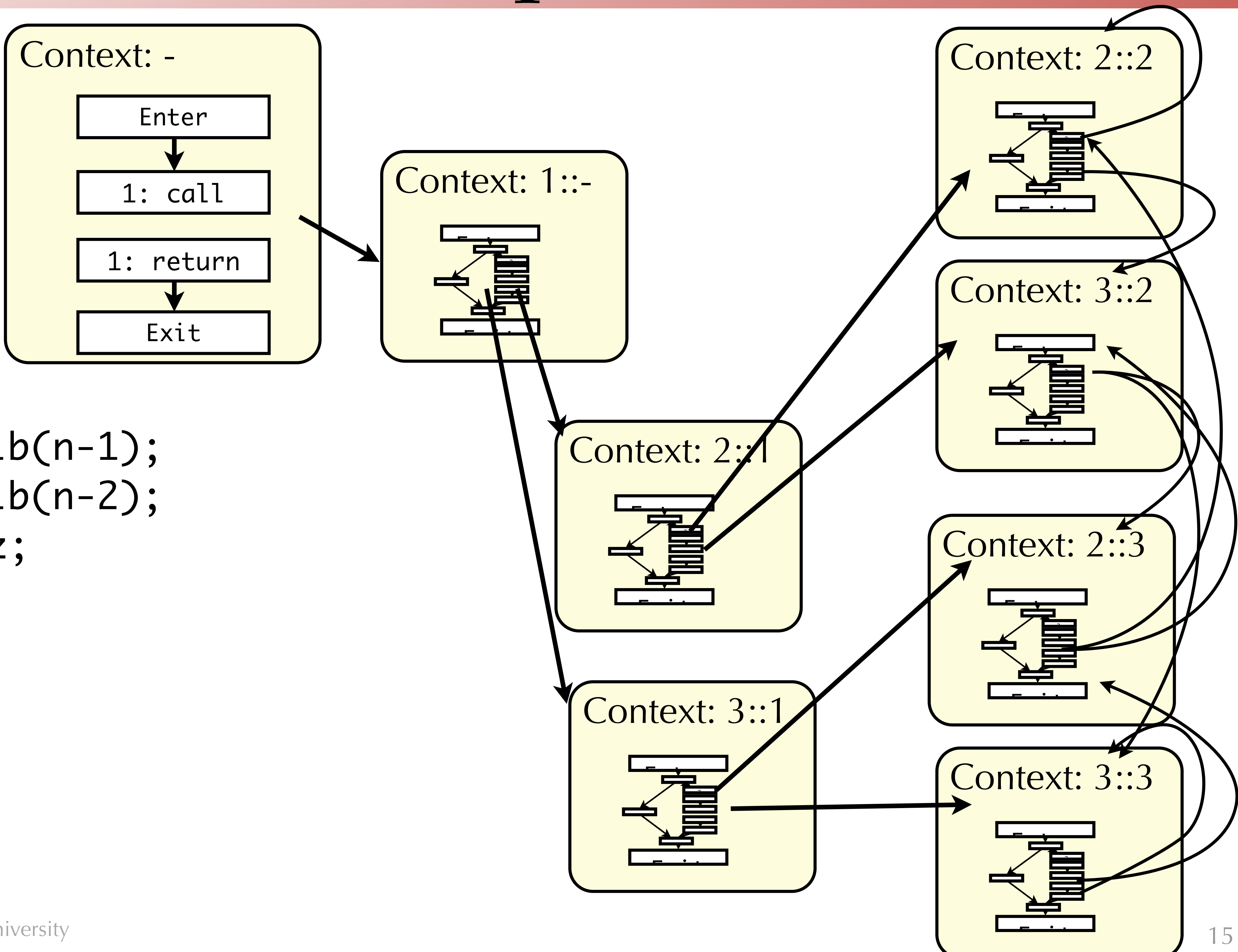
Fibonacci: context sensitive, stack depth 1

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
     2: y := fib(n-1);
     3: z := fib(n-2);
        x:= y+z;
  return x;
}
```



Fibonacci: context sensitive, stack depth 2

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
    2: y := fib(n-1);
    3: z := fib(n-2);
       x:= y+z;
  return x;
}
```



Other contexts

- Context sensitivity distinguishes between different calls of the same procedure
  - Choice of contexts determines which calls are differentiated
- Other choices of context are possible
  - Caller stack
    - Less precise than call-site stack
    - E.g., context "2::2" and "2::3" would both be "fib::fib"
  - Object sensitivity: which object is the target of the method call?
    - For OO languages.
    - Maintains precision for some common OO patterns
    - Requires pointer analysis to determine which objects are possible targets
    - Can use a stack (i.e., target of methods on call stack)

Other contexts

- More choices
  - Assumption sets
    - What state (i.e., dataflow facts) hold at the call site?
    - Used in ESP paper
  - Combinations of contexts, e.g., Assumption set and object

Procedure summaries

- In practice, often don't construct single CFG and perform dataflow
- Instead, store **procedure summaries** and use those
- When `call p` is encountered in context *C*, with input *D*, check if procedure summary for p in context *C* exists.
  - If not, process p in context *C* with input *D*
  - If yes, with input D' and output E'
    - if $D' \sqsubseteq D$, then use E'
    - if $D' \not\sqsubseteq D$, then process p in context *C* with input $D' \sqcap D$
  - If output of p in context *C* changes then may need to reprocess anything that called it
  - Need to take care with recursive calls

Flow-sensitivity

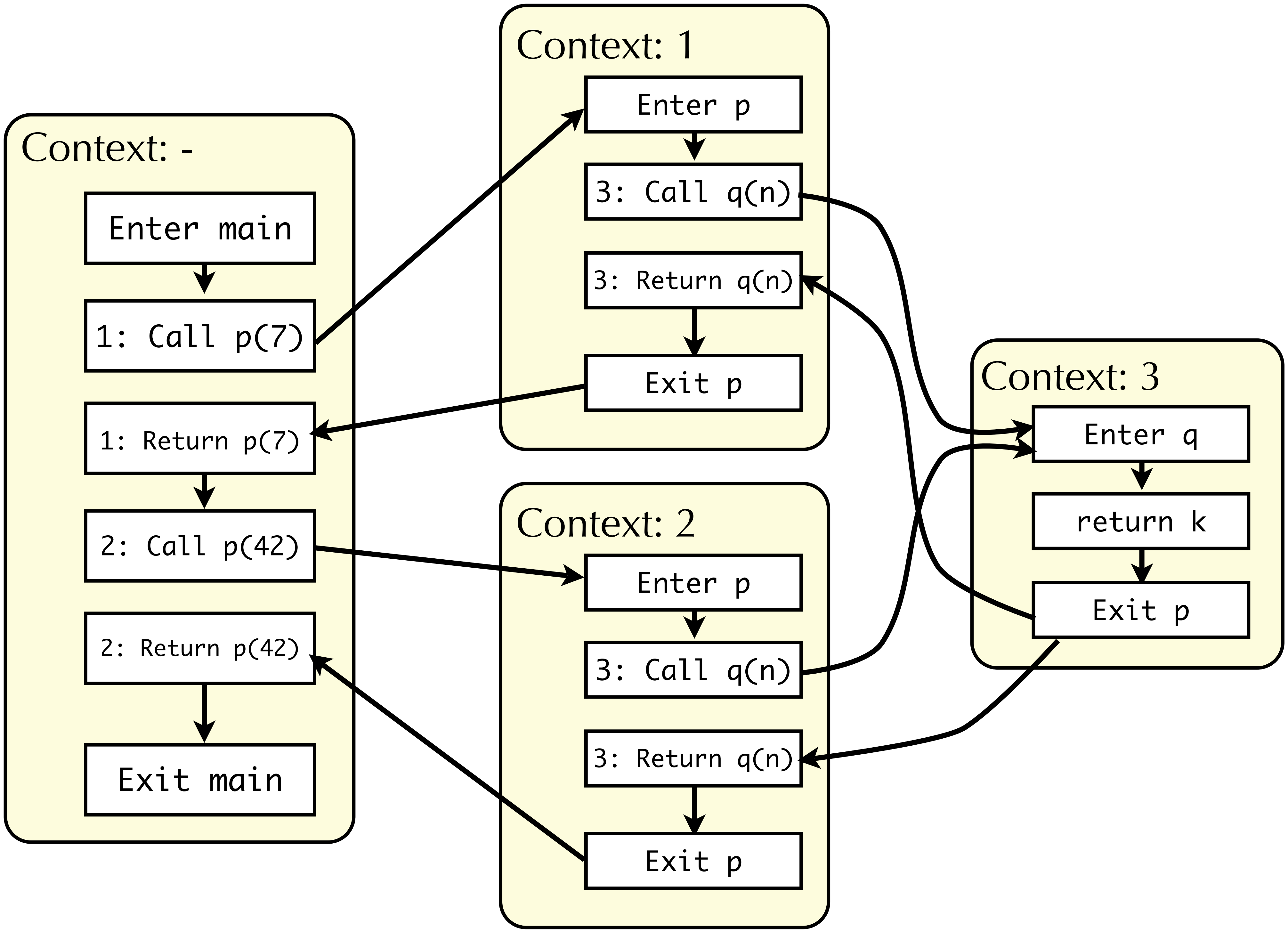
- Recall: in a **flow insensitive** analysis, order of statements is not important
  - e.g., analysis of $c_1;c_2$ will be the same as $c_2;c_1$
- Flow insensitive analyses typically cheaper than flow sensitive analyses
- Can have both flow-sensitive interprocedural analyses and flow-insensitive interprocedural analyses
  - Flow-insensitivity can reduce the cost of interprocedural analyses

Infeasible paths

- Context sensitivity increases precision by analyzing the same procedure in possibly many contexts
- But still have problem of **infeasible paths**
  - Paths in control flow graph that do not correspond to actual executions

Infeasible paths example

```
main() {
  1: p(7);
  2: x:=p(42);
}

p(int n) {
  3: q(n);
}
q(int k) {
  return k;
}
```



Realizable paths

- Idea: restrict attention to **realizable paths**: paths that have proper nesting of procedure calls and exits
- For each call site i, let's label the call edge "$(_i$" and the return edge "$)_i$"
- Define a grammar that represents balanced paren strings

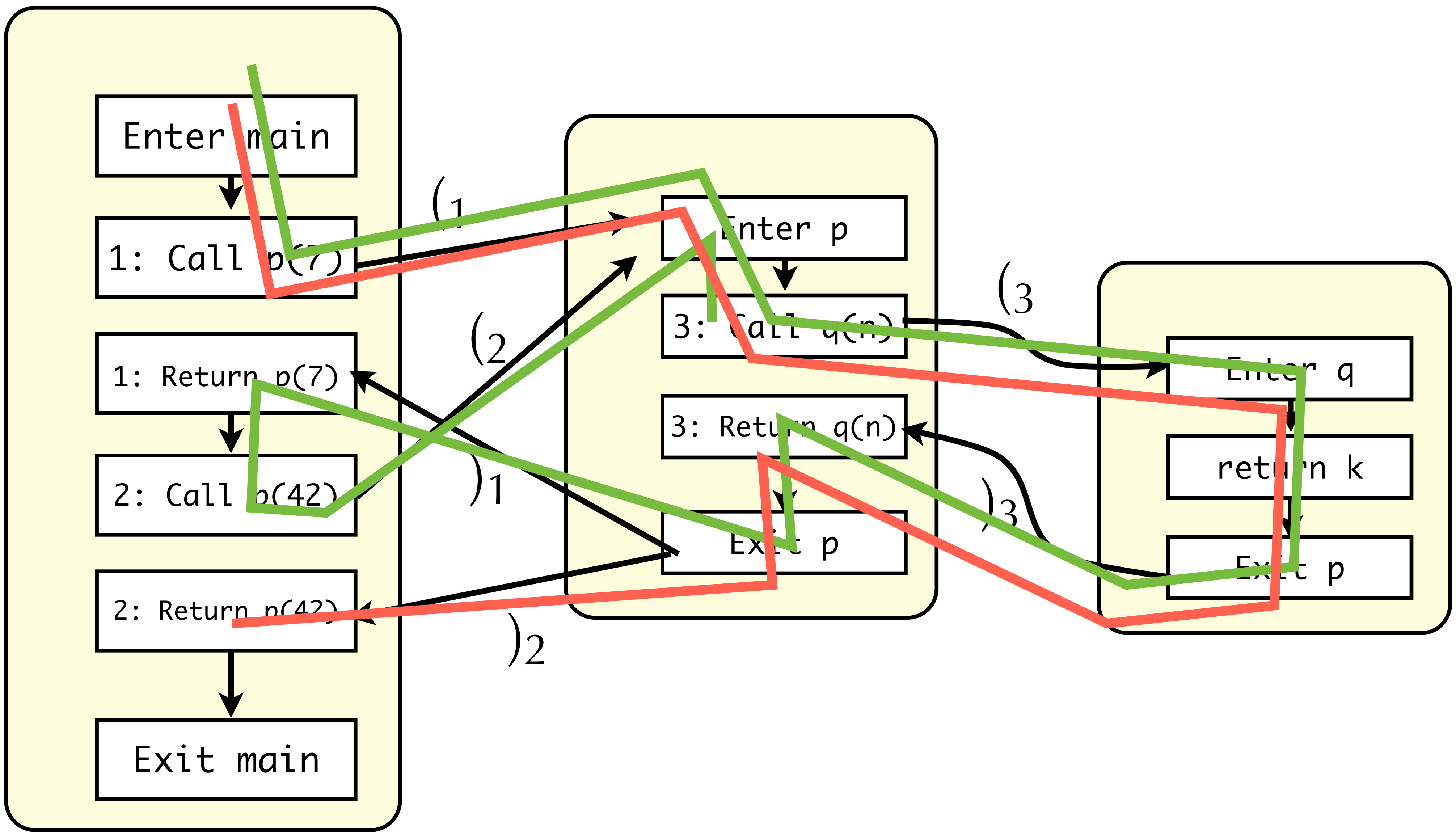
```
matched ::= ∈                          empty string
          | e                          anything not containing parens
          | matched matched
          | (i matched )i
```

  - Corresponds to matching procedure returns with procedure calls
- Define grammar of partially balanced parens (calls that have not yet returned)

```
realizable ::= ∈
             | (i realizable
             | matched realizable
```

Example

```
main() {
  1: p(7);
  2: x:=p(42);
}

p(int n) {
  3: q(n);
}
q(int k) {
  return k;
}
```



Meet over Realizable Paths

- Previously we wanted to calculate the dataflow facts that hold at a node in the CFG by taking the **meet over all paths** (MOP)
- But this may include infeasible paths
- **Meet over all realizable paths** (MRP) is more precise
  - For a given node *n*, we want the meet of all realizable paths from the start of the CFG to *n*
  - May have paths that don't correspond to any execution, but every execution will correspond to a realizable path
  - realizable paths are a subset of all paths
  - $\Rightarrow$ MRP sound but more precise: MRP $\sqsubseteq$ MOP

Program analysis as CFL reachability

- Can phrase many program analyses as **context-free language reachability** problems in directed graphs
  - "Program Analysis via Graph Reachability" by Thomas Reps, 1998
    - Summarizes a sequence of papers developing this idea

CFL Reachability

- Let *L* be a context-free language over alphabet Σ
- Let *G* be graph with edges labeled from Σ
- Each path in *G* defines word over Σ
- A path in *G* is an ***L*-path** if its word is in *L*
- CFL reachability problems:
  - **All-pairs L-path problem**: all pairs of nodes n_1, n_2 such that there is an L-path from n_1 to n_2
  - **Single-source L-path problem**: all nodes n_2 such that there is an L-path from given node n_1 to n_2
  - **Single-target L-path problem**: all nodes n_1 such that there is an L-path from n_1 to given node n_2
  - **Single-source single-target L-path problem**: is there an L-path from given node n_1 to given node n_2

Why bother?

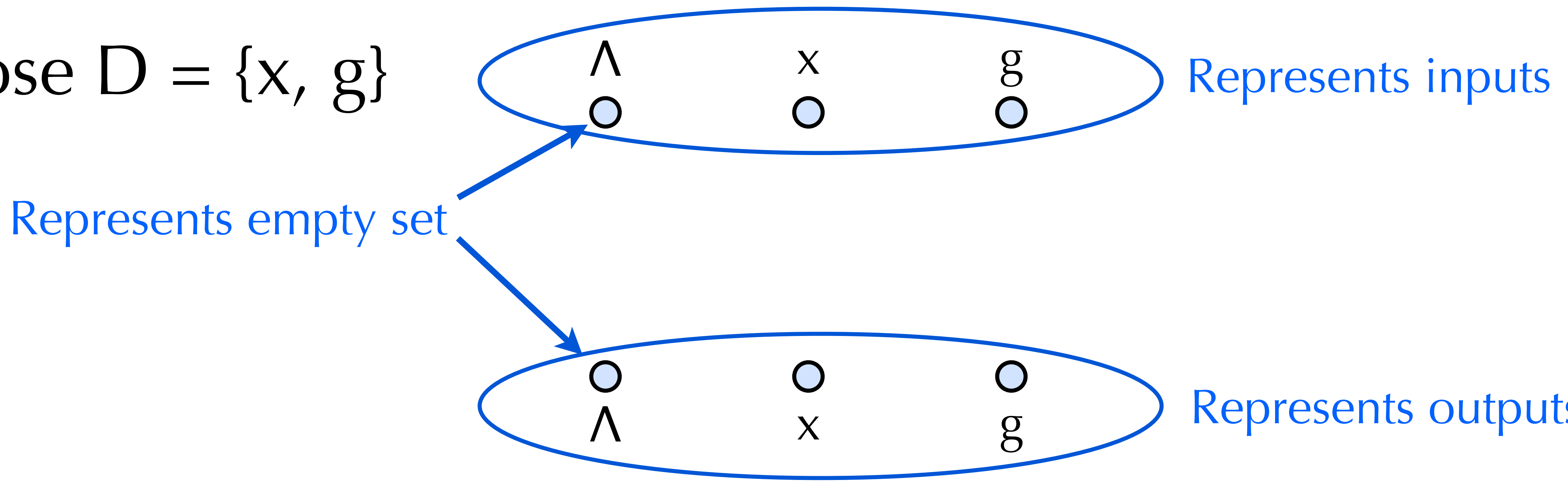
- All CFL-reachability problems can be solved in time cubic in nodes of the graph
- Automatically get a faster, approximate solution: graph reachability
- **On demand** analysis algorithm for free
- Gives insight into program analysis complexity issues

Encoding 1: IFDS problems

- **Interprocedural finite distributive subset** problems (IFDS problems)
  - Interprocedural dataflow analysis with
    - Finite set of data flow facts
    - Distributive dataflow functions ($f(a \sqcap b) = f(a) \sqcap f(b)$)
- Can convert any IFDS problem as a CFL-graph reachability problem, and find the MRP solution with no loss of precision
  - May be some loss of precision phrasing problem as IFDS

Encoding distributive functions

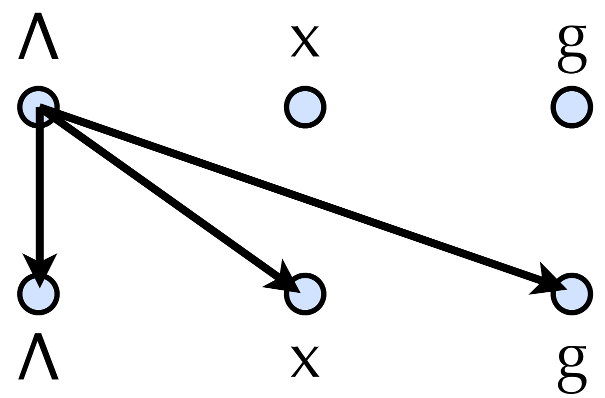
- Key insight: distributive function $f:2^D \rightarrow 2^D$ can be encoded as graph with $2D+2$ nodes
- W.L.O.G. assume $\sqcap \equiv \cup$
- E.g., suppose D = {x, g}



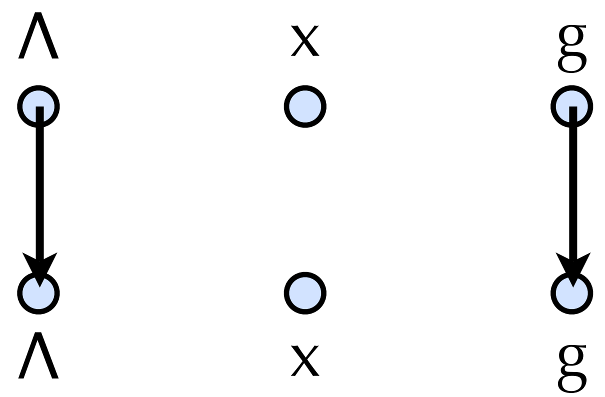
- Edge $\Lambda \rightarrow d$ means $d \in f(S)$ for all S
- Edge $d_1 \rightarrow d_2$ means $d_2 \notin f(\emptyset)$ and $d_2 \in f(S)$ if $d_1 \in S$
- Edge $\Lambda \rightarrow \Lambda$ always in graph (allows composition)

Encoding distributive functions

- λS. {x,g}

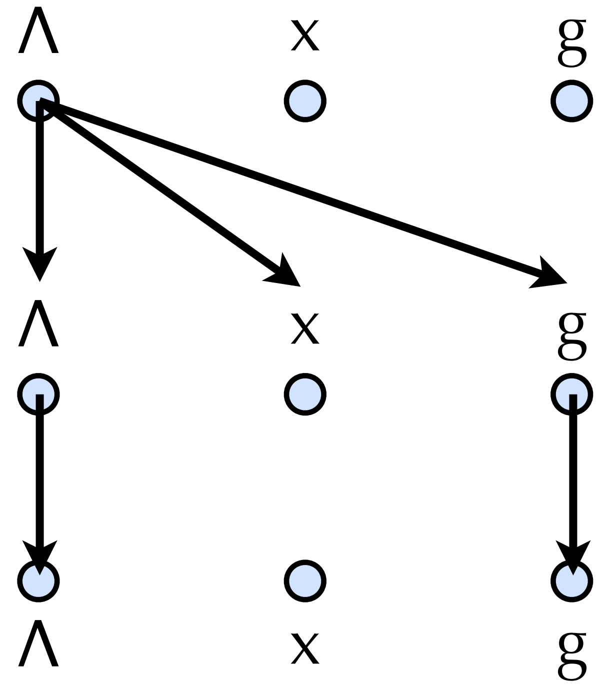


- λS. S-{x}



Encoding distributive functions

- λS. S-{x} ○ λS. {x,g}



Exploded supergraph $G^\#$

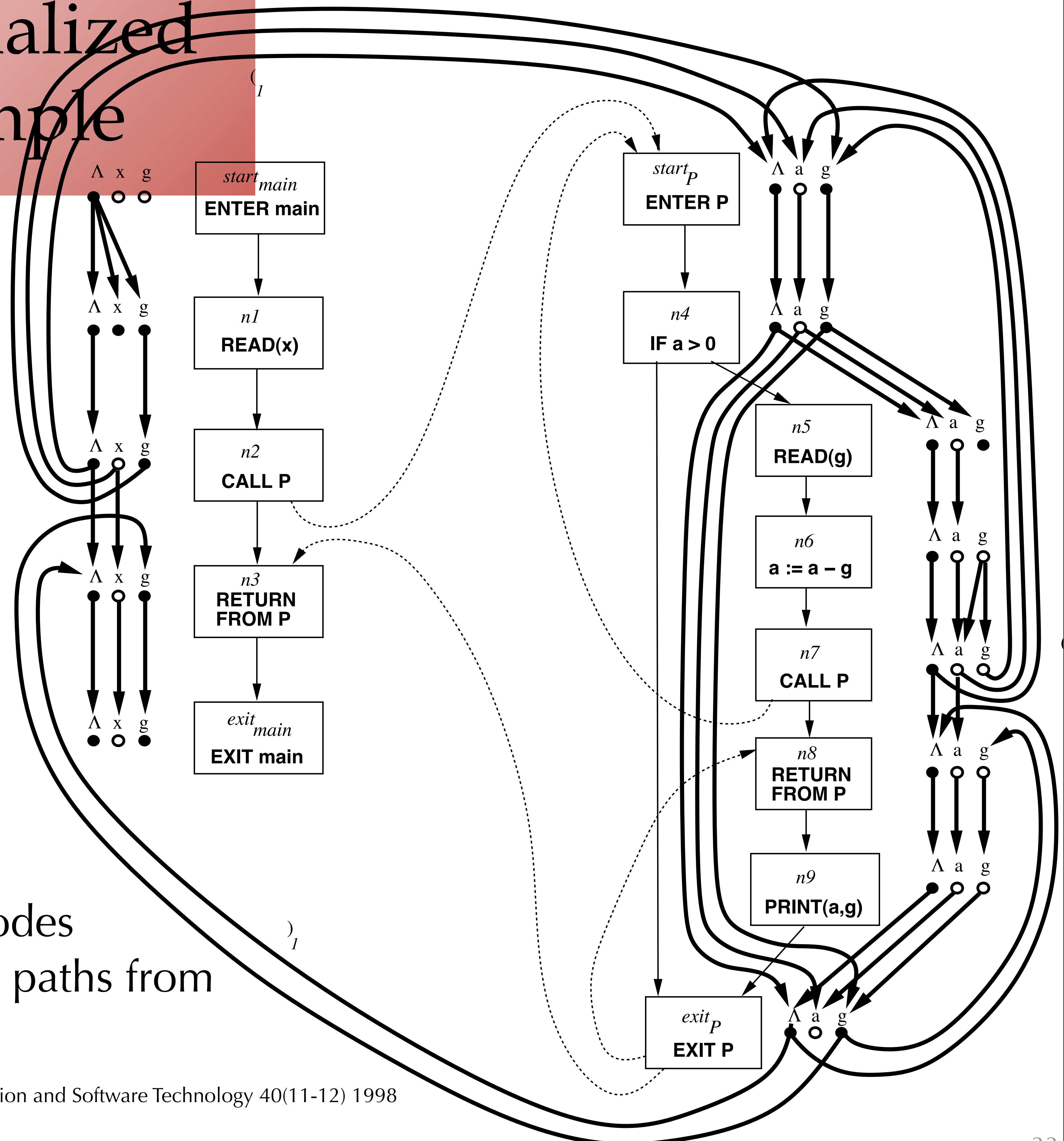
- Let G^* be supergraph (i.e., interprocedural CFP)
- For each node $n \in G^*$, there is node $\langle n, \Lambda \rangle \in G^\#$
- For each node $n \in G^*$, and $d \in D$ there is node $\langle n, d \rangle \in G^\#$
- For function f associated with edge $a \rightarrow b \in G^*$
  - Edge $\langle a, \Lambda \rangle \rightarrow \langle b, d \rangle$ for every $d \in f(\emptyset)$
  - Edge $\langle a, d_1 \rangle \rightarrow \langle b, d_2 \rangle$ for every $d_2 \in f(\{d_2\})$ - $f(\emptyset)$
  - Edge $\langle a, \Lambda \rangle \rightarrow \langle b, \Lambda \rangle$

Possibly uninitialized variable example

declare g: **int**

procedure *main*
begin
  declare x: **int**
  read(x)
  call P (x)
end

procedure P (**value** a : **int**)
begin
  if ($a > 0$) **then**
    read(g)
    $a := a - g$
    call P (a)
    print(a, g)
  fi
end



- Closed circles represent nodes reachable along realizable paths from $\langle start_{main}, \Lambda \rangle$

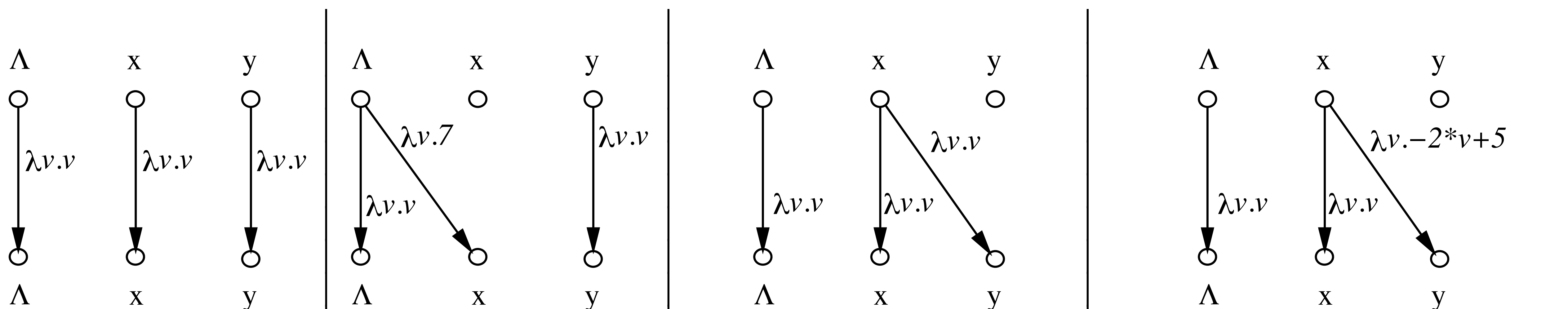
Program Analysis via Graph Reachability by Reps, Information and Software Technology 40(11-12) 1998

Encoding 2: IDE problems

- **Interprocedural Distributive Environment** problems (IDE problems)
  - Interprocedural dataflow analysis with
    - Dataflow info at program point represented as a finite **environment** (i.e., mapping from variables/locations to finite height domain of values)
    - Transfer function distributive "environment transformer"
  - E.g., copy constant propagation
    - interprets assignment statements such as x=7 and y=x
  - E.g. linear constant propagation
    - also interprets assignment statements such as y = 5*z + 9

Encoding distributive environment-transformers

- Similar trick to encoding distributive functions in IFDS
- Represent environment-transformer function as graph with each edge labeled with **micro-function**



(a) $\lambda env.env$ (b) $\lambda env.env\,[x \mapsto 7]$ (c) $\lambda env.env\,[y \mapsto env\,(x)]$ (d) $\lambda env.env\,[y \mapsto -2*env\,(x)+5]$

Solving

- Requirements for class F of micro functions
  - Must be closed under meet and composition
  - F must have finite height (under pointwise ordering)
  - f(l) can be computed in constant time
  - Representation of f is of bounded size
  - Given representation of $f_1, f_2 \in F$
    - can compute representation of $f_1 \circ f_2 \in F$ in constant time
    - can compute representation of $f_1 \sqcap f_2 \in F$ in constant time
    - can compute $f_1 = f_2$ in constant time

Solving

- First pass computes **jump functions** and **summary functions**
  - Summaries of paths within a procedure and of procedure calls, respectively
- Second pass uses these functions to computer environments at program points
- More details in "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation" by Sagiv, Reps, and Horwitz, 1996.