HARVARD

School of Engineering
and Applied Sciences

Interprocedural Analysis

CS252r Spring 2011

Procedures

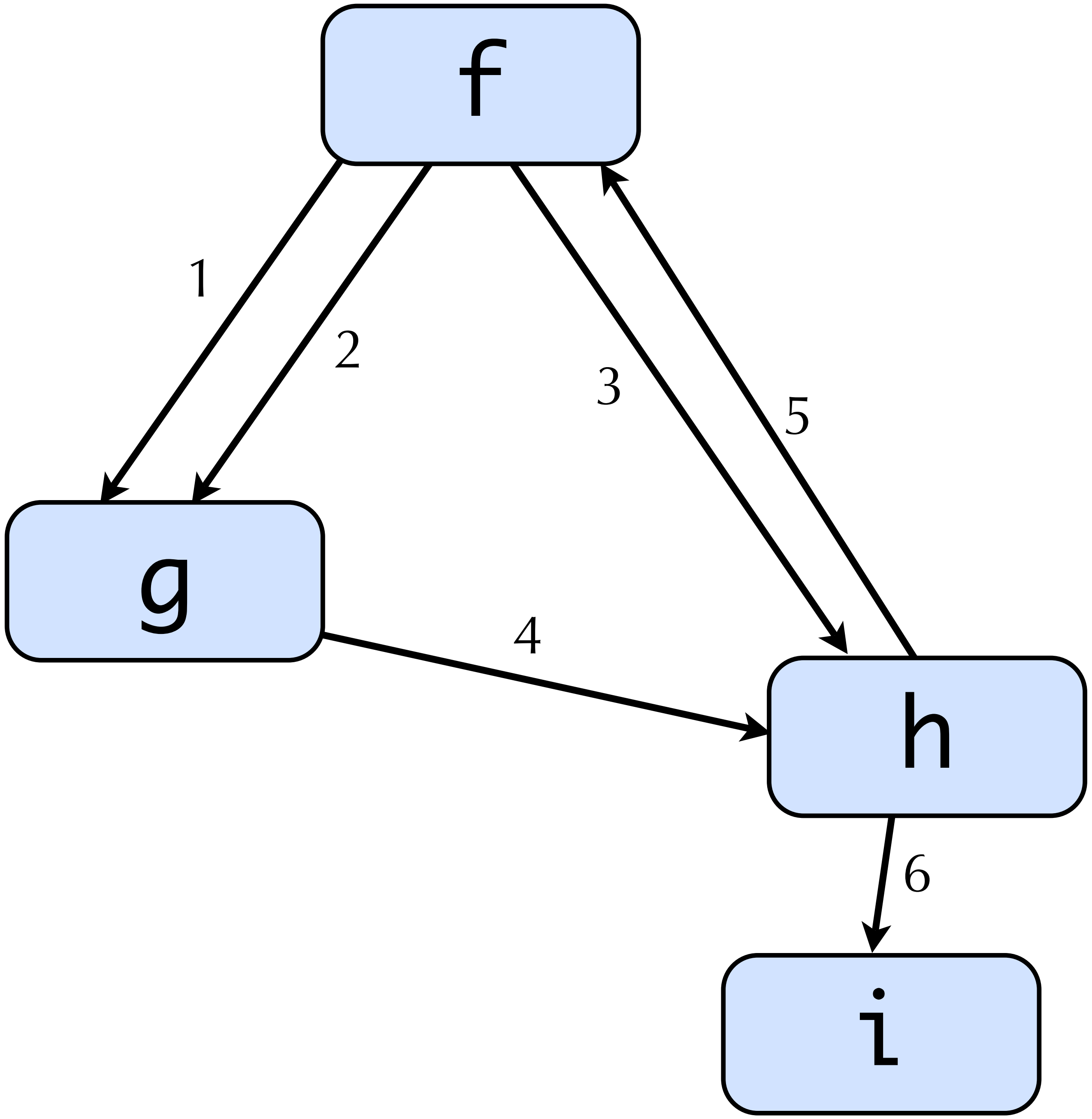
- So far looked at **intraprocedural** analysis: analyzing a single procedure
- **Interprocedural analysis** uses calling relationships among procedures
  - Enables more precise analysis information

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

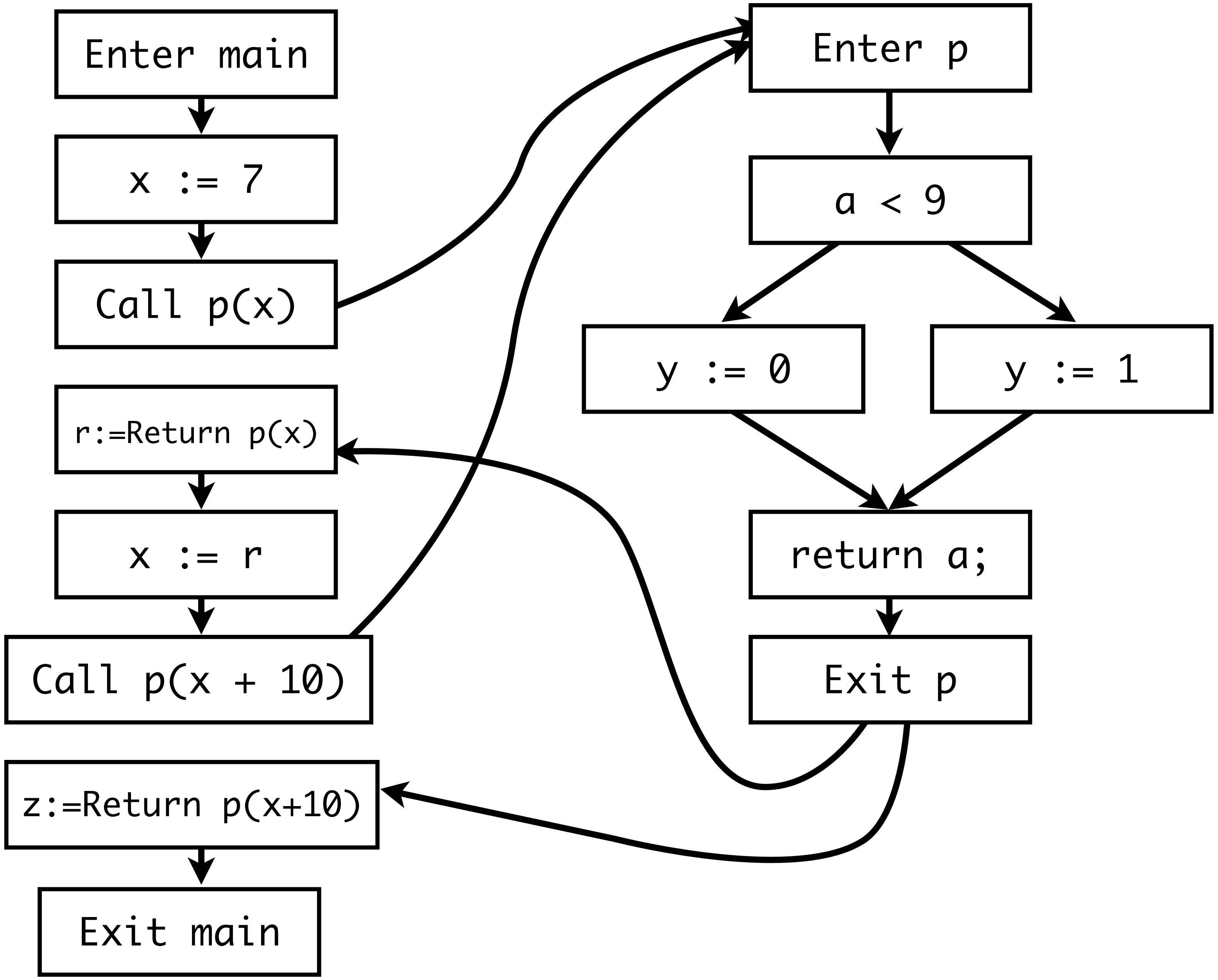
```
f() {
   1:  g();
   2:  g();
   3:  h();
}

g() {
   4: h();
}

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

i() { … }
```



Interprocedural dataflow analysis

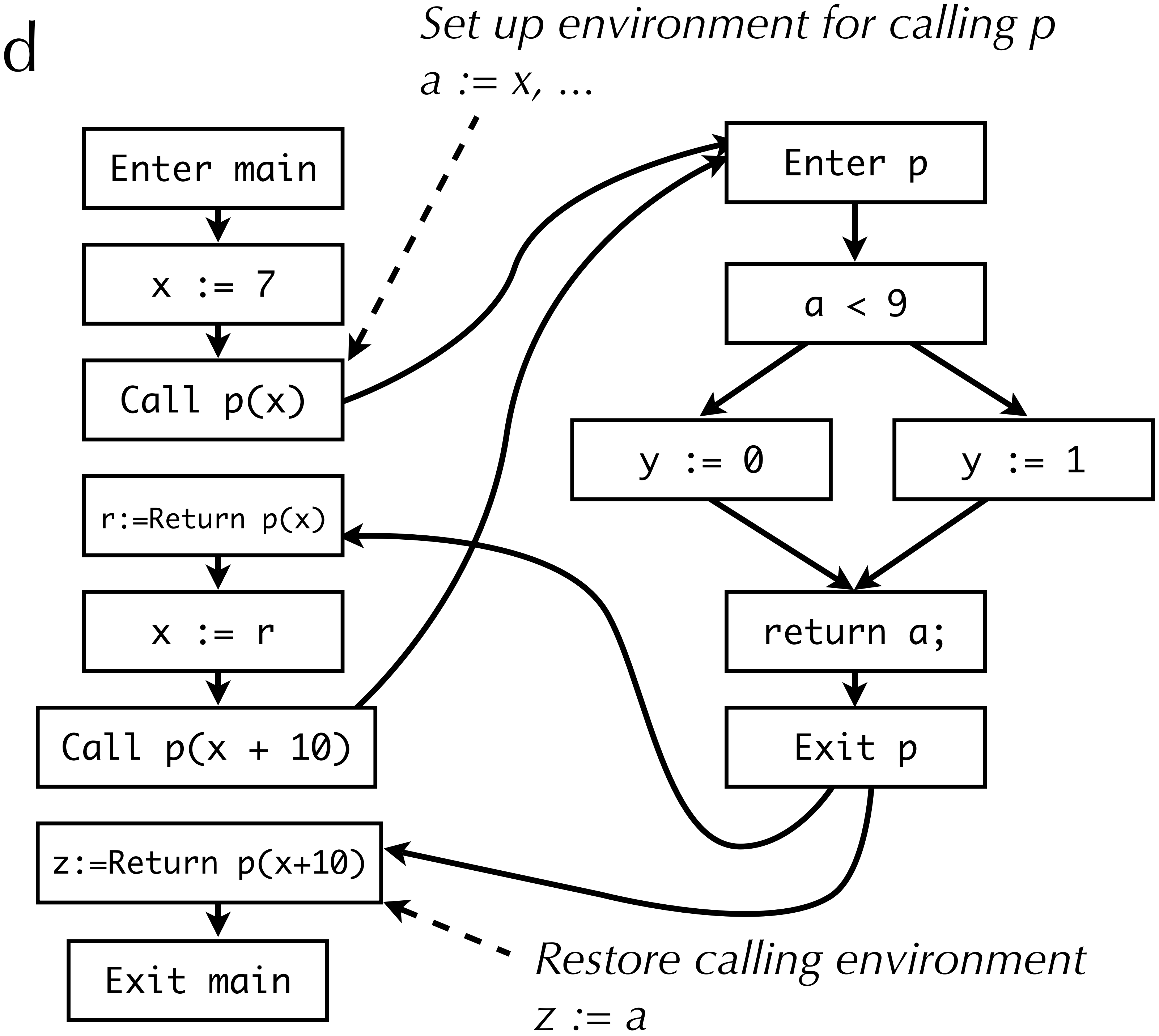
- How do we deal with procedure calls?
- Obvious idea: make one big CFG

```
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;
}
```



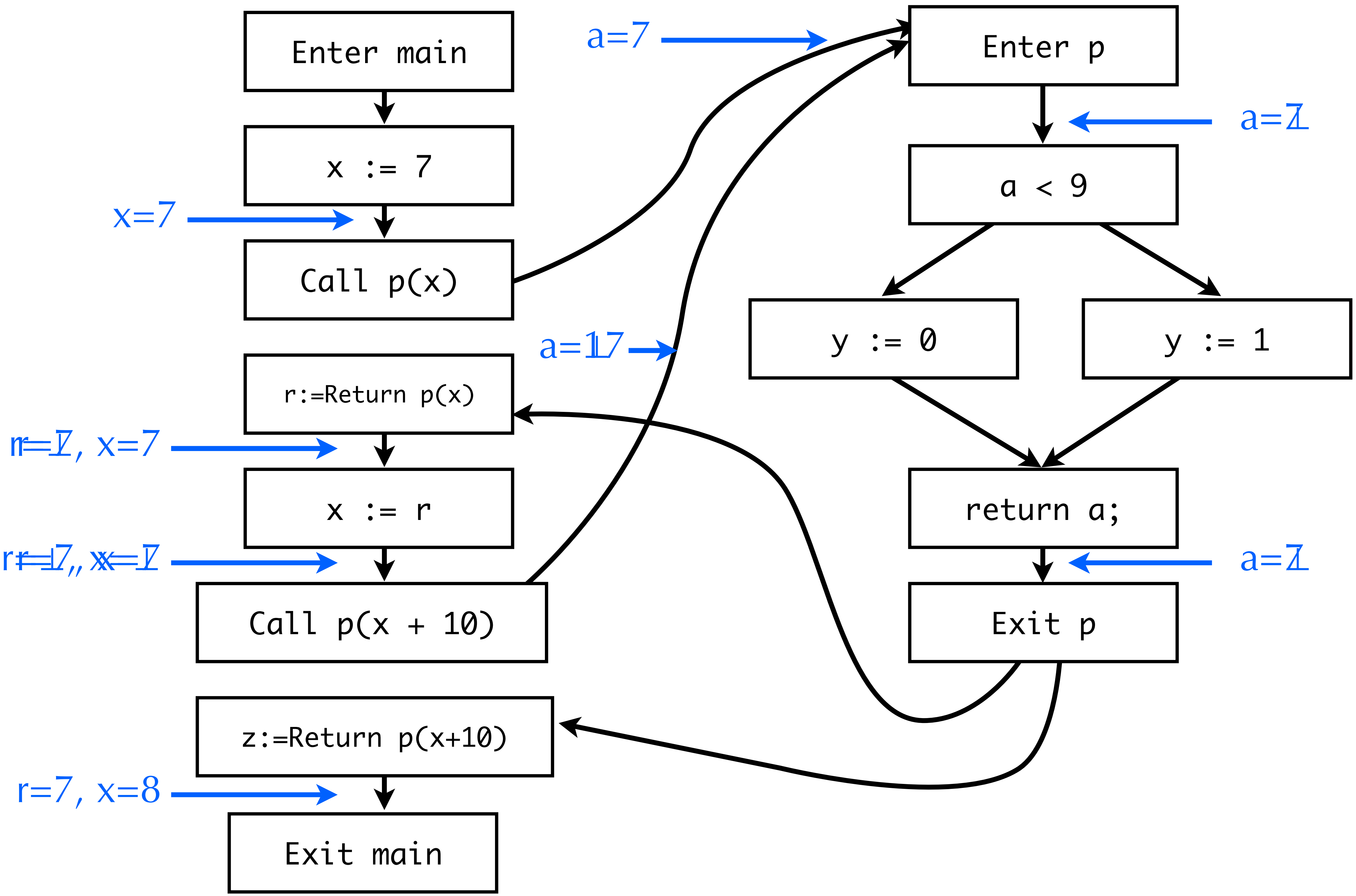
Interprocedural CFG

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



© 2010 Stephen Chong, Harvard University

Example

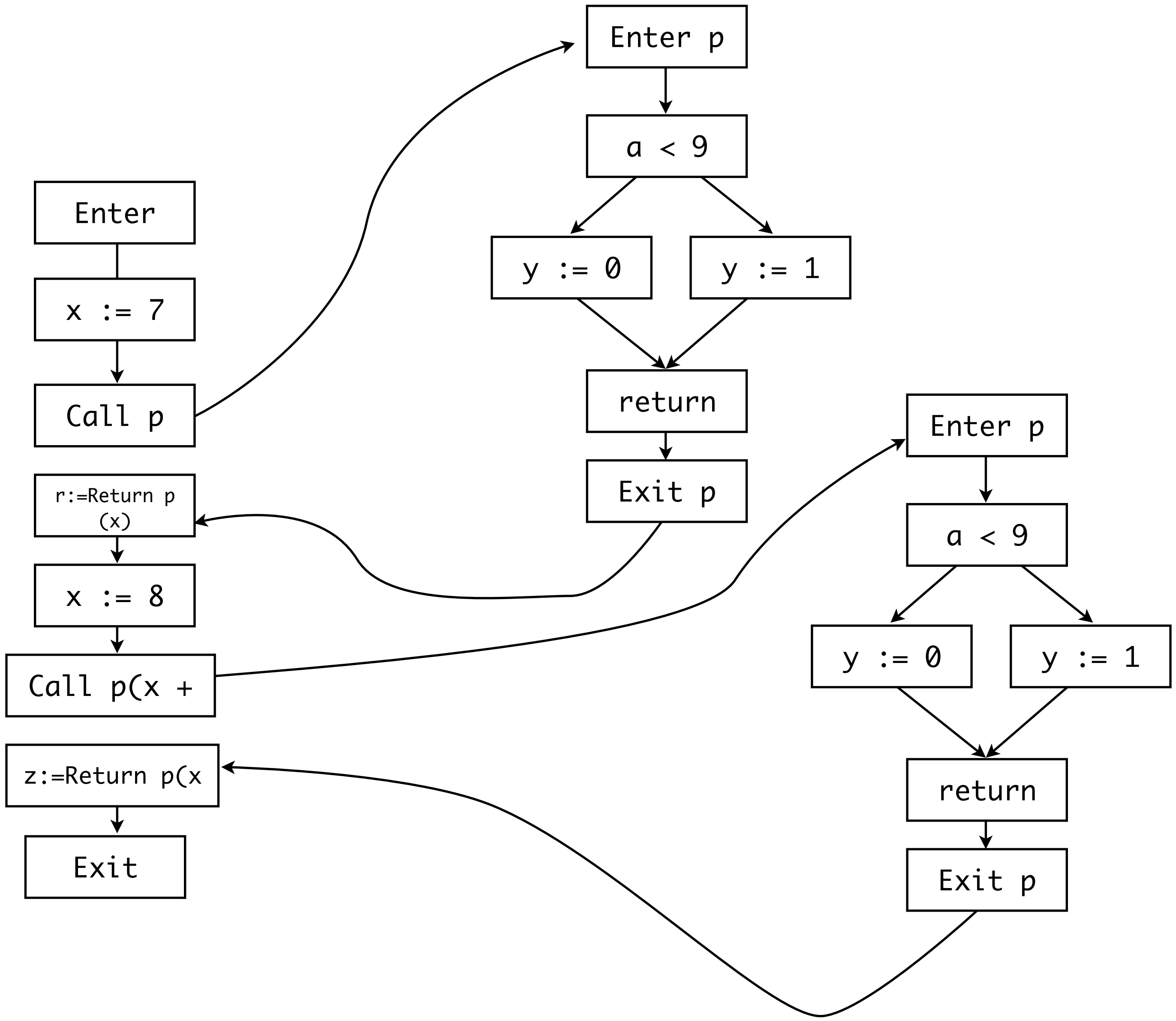

Enter main
x := 7
x=7
Call p(x)
a=7
a=17
r:=Return p(x)
r=⊥, x=7
r=7, x=7
x := r
r=⊥, x=⊥
r=7, x=7
Call p(x + 10)
z:=Return p(x+10)
r=7, x=8
Exit main
Enter p
a=7
a=⊥
a < 9
y := 0
y := 1
return a;
a=7
a=⊥
Exit p

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

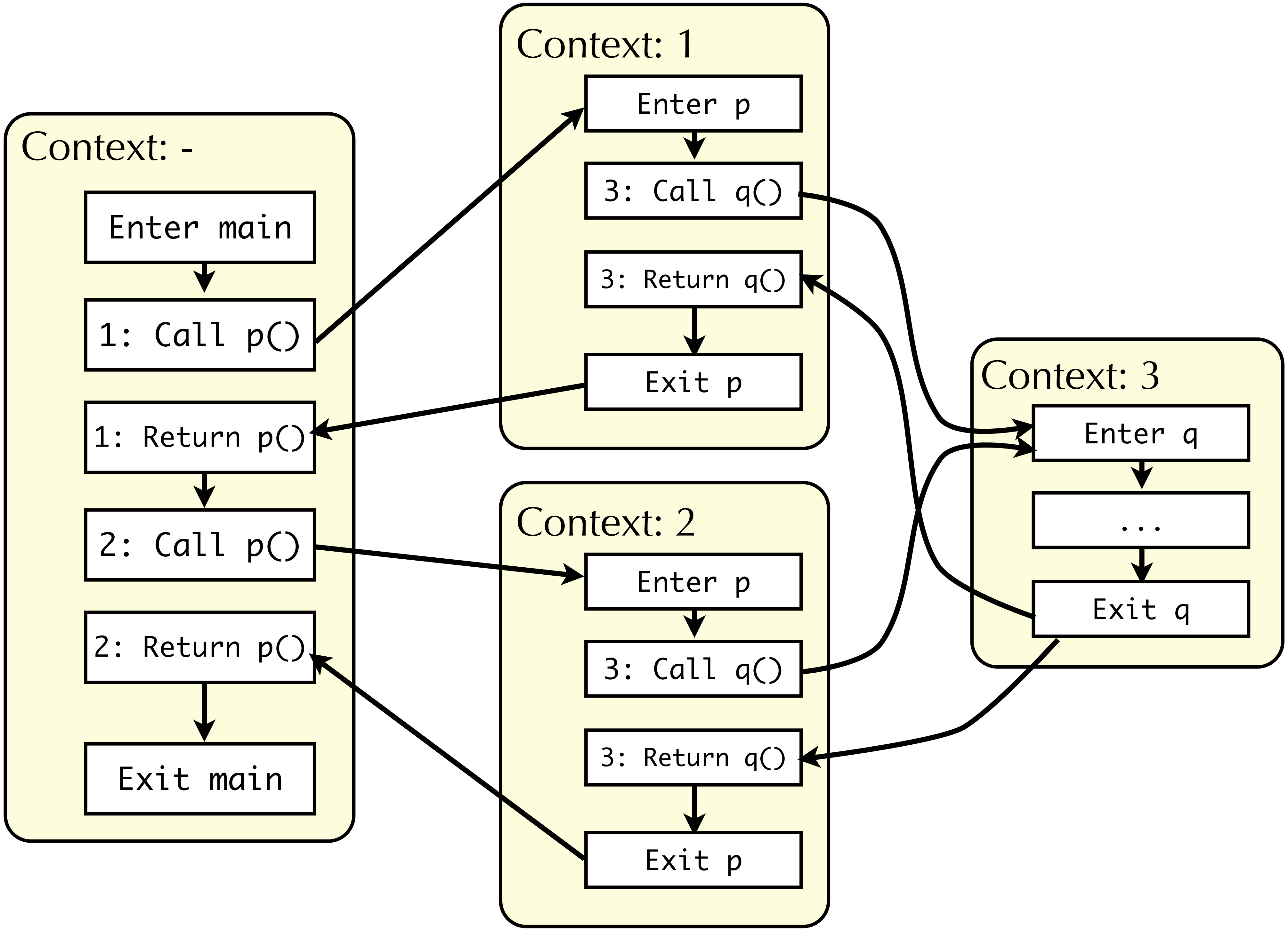


Context sensitivity

- 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

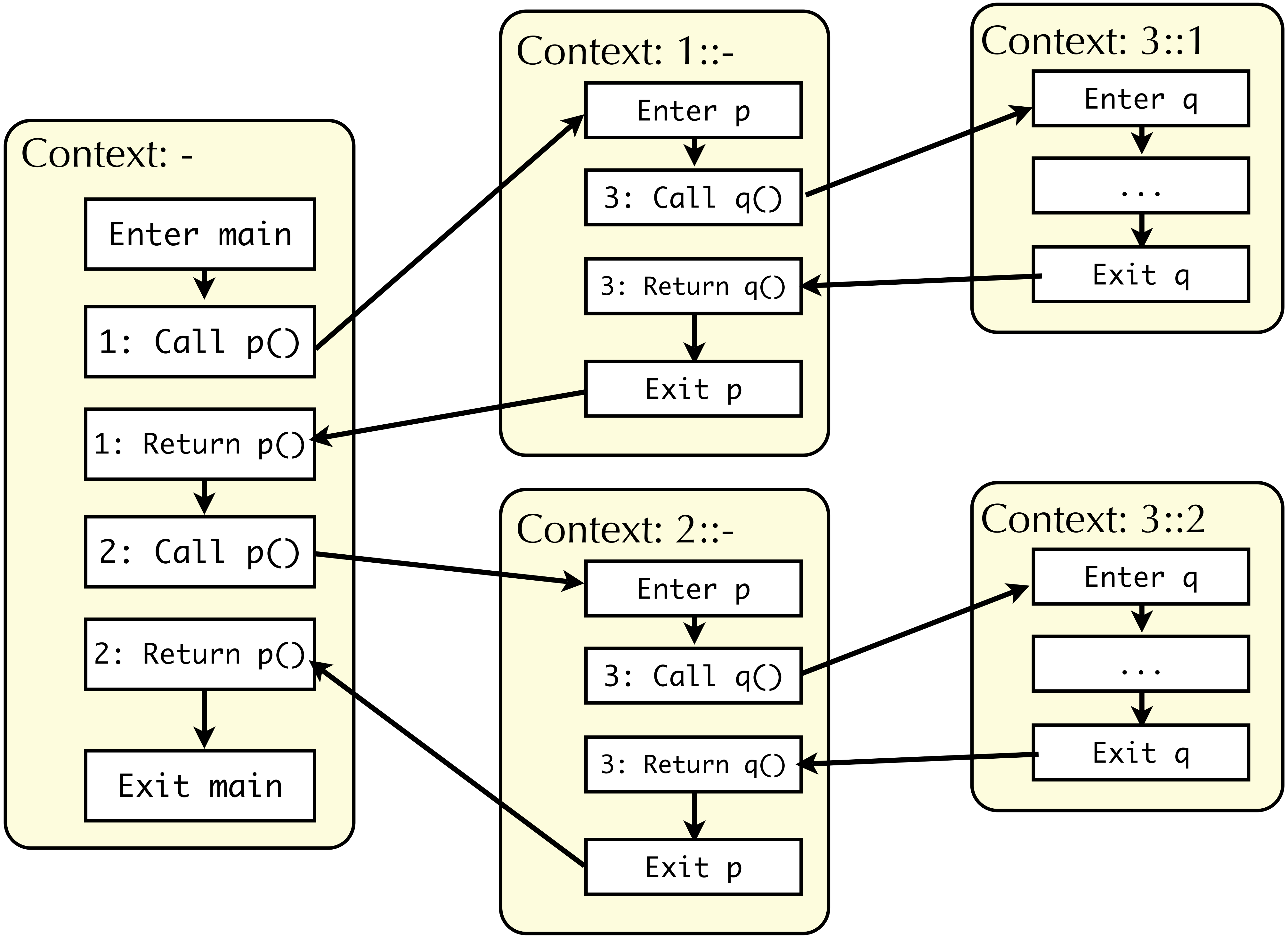
Context sensitivity example

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

p() {
  3: q();
}
q() {
  …
}
```



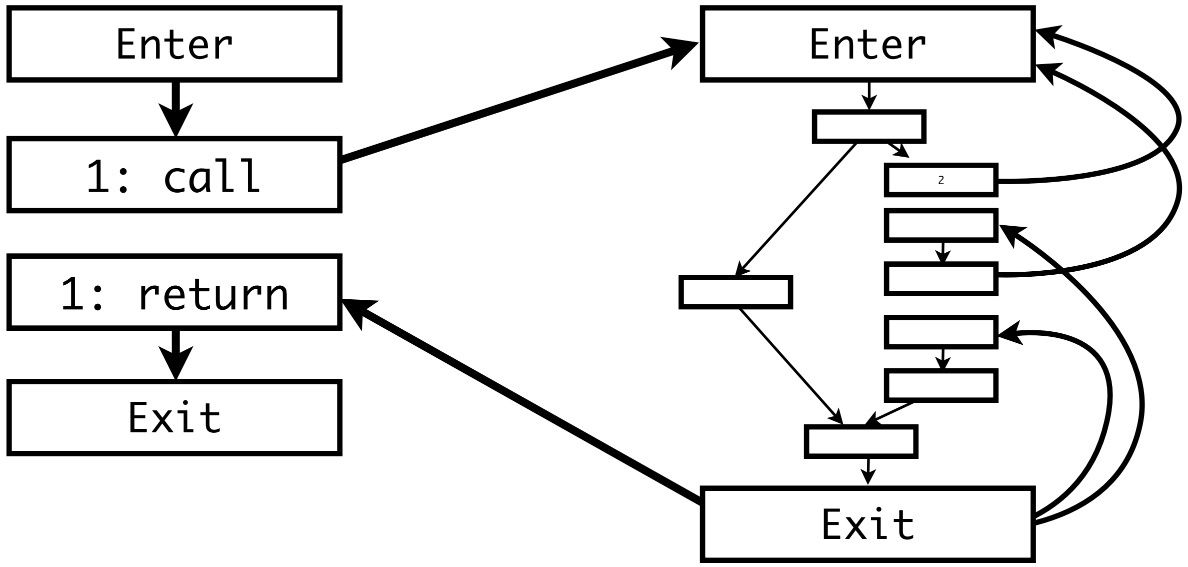
Context sensitivity example

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

p() {
  3: q();
}
q() {
  …
}
```



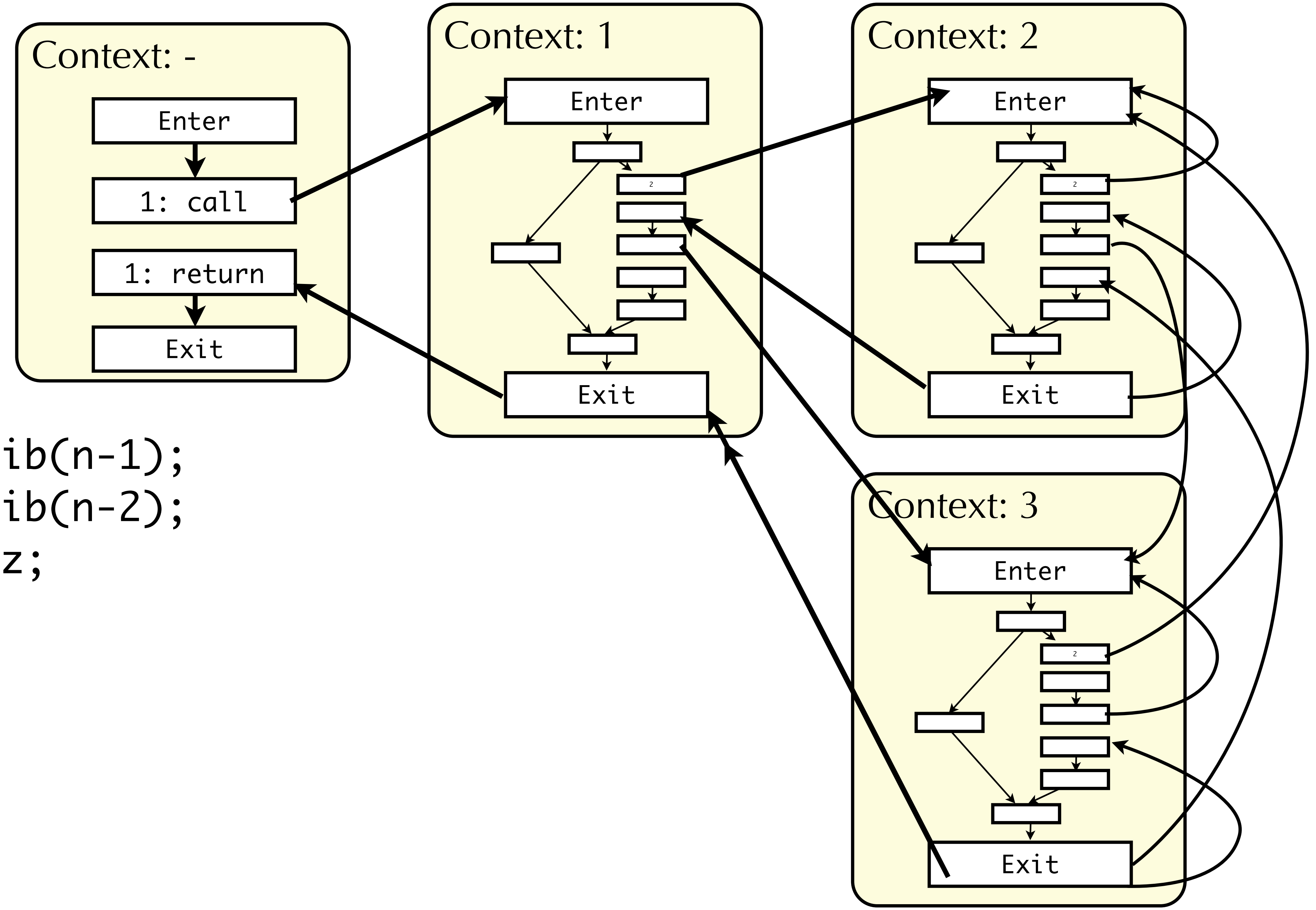
Fibonacci: context insensitive

```
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;
}
```



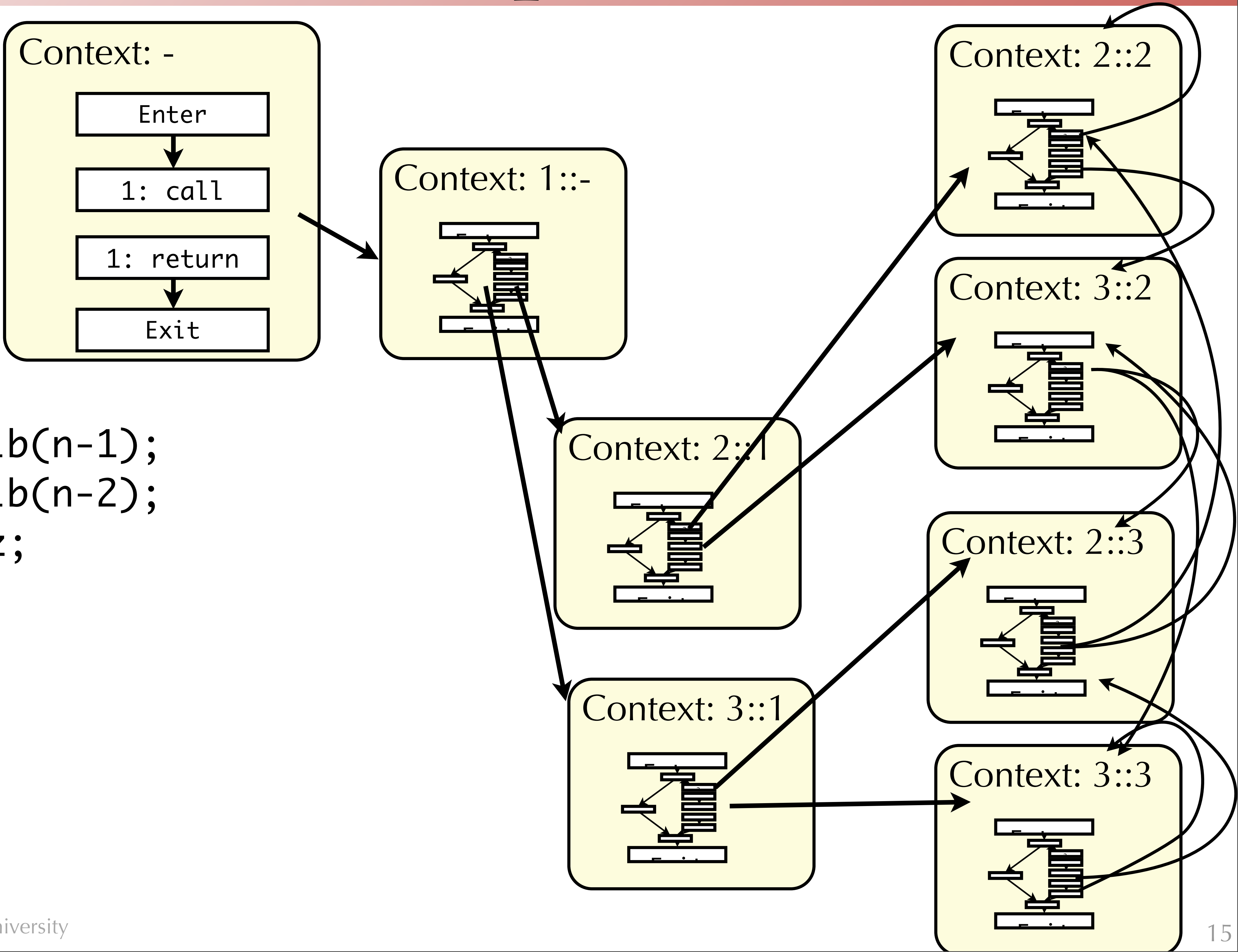
Fibonacci: context sensitive, stack depth 1

```
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;
}
```



Flow-sensitivity

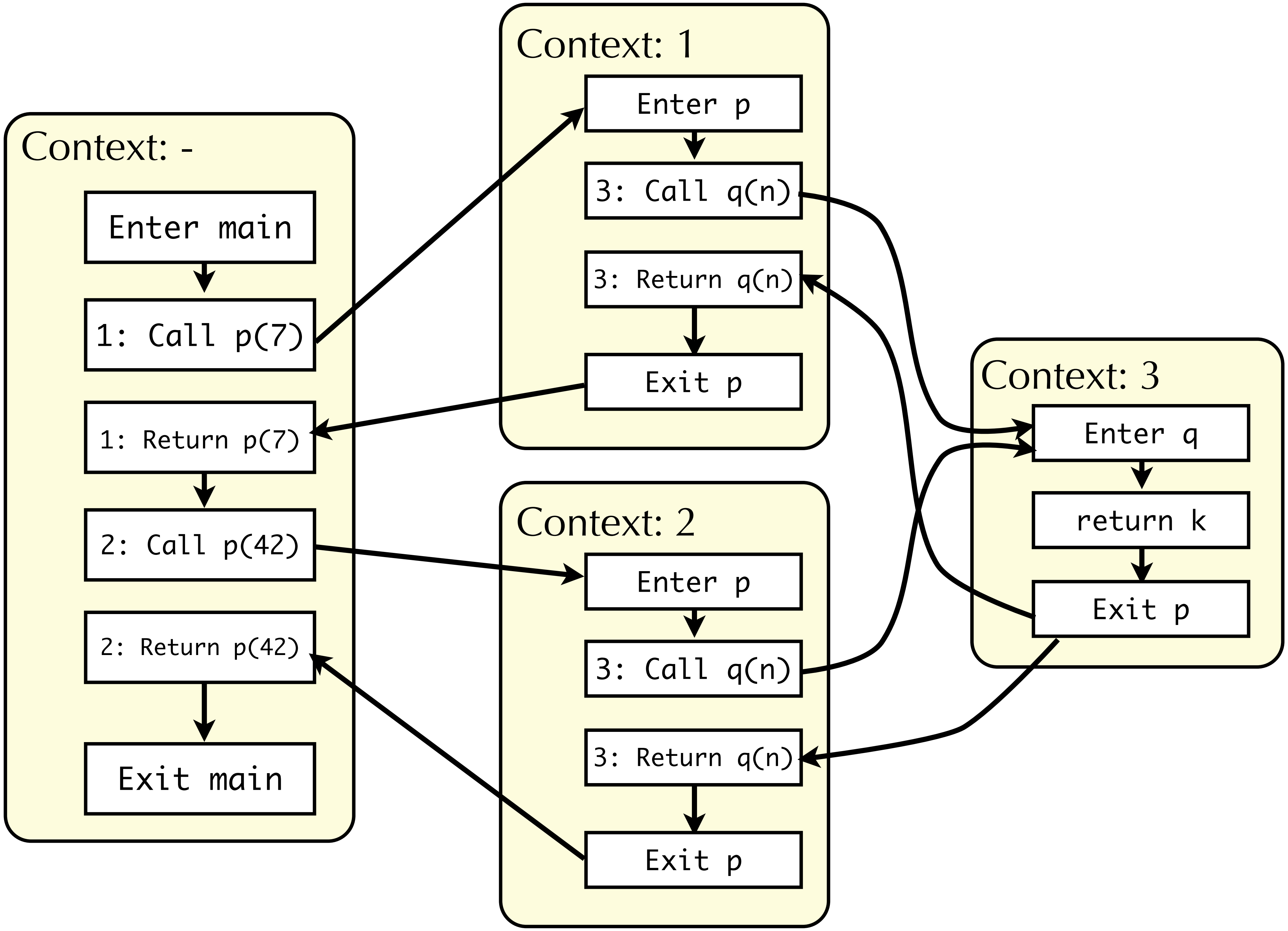
- Recall: in a **flow insensitive** analysis, order of statements is not important
  - e.g., analysis of $c_1;c_2$ will be the same as $c_2;c_1$
- Flow insensitive analyses typically cheaper than flow sensitive analyses
- Can have both flow-sensitive interprocedural analyses and flow-insensitive interprocedural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



Context: 1
Enter p
3: Call q(n)
3: Return q(n)
Exit p
Context: -
Enter main
1: Call p(7)
1: Return p(7)
2: Call p(42)
2: Return p(42)
Exit main
Context: 3
Enter q
return k
Exit p
Context: 2
Enter p
3: Call q(n)
3: Return q(n)
Exit p