Program Analysis via Graph Reachability

Thomas Reps

University of Wisconsin

http://www.cs.wisc.edu/~reps/

PLDI '00 Tutorial, Vancouver, B.C., June 18, 2000

Backward Slice

```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}
```

Backward slice with respect to "printf("%d\n",i)"

Backward Slice

```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}
```

Backward slice with respect to "printf("%d\n",i)"

Slice Extraction

```
int main() {

    int i = 1;
    while (i < 11) {

        i = i + 1;
    }

    printf("%d\n",i);
}
```

Backward slice with respect to "printf("%d\n",i)"

Forward Slice

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

Forward slice with respect to "sum = 0"

Forward Slice

```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}
```

Forward slice with respect to "sum = 0"

What Are Slices Useful For?

- Understanding Programs
  - What is affected by what?
- Restructuring Programs
  - Isolation of separate "computational threads"
- Program Specialization and Reuse
  - Slices = specialized programs
  - Only reuse needed slices
- Program Differencing
  - Compare slices to identify changes
- Testing
  - What new test cases would improve coverage?
  - What regression tests must be rerun after a change?

Line-Character-Count Program

```
void line_char_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
      scan_line(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Character-Count Program

```
void char_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
      scan_line(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Line-Character-Count Program

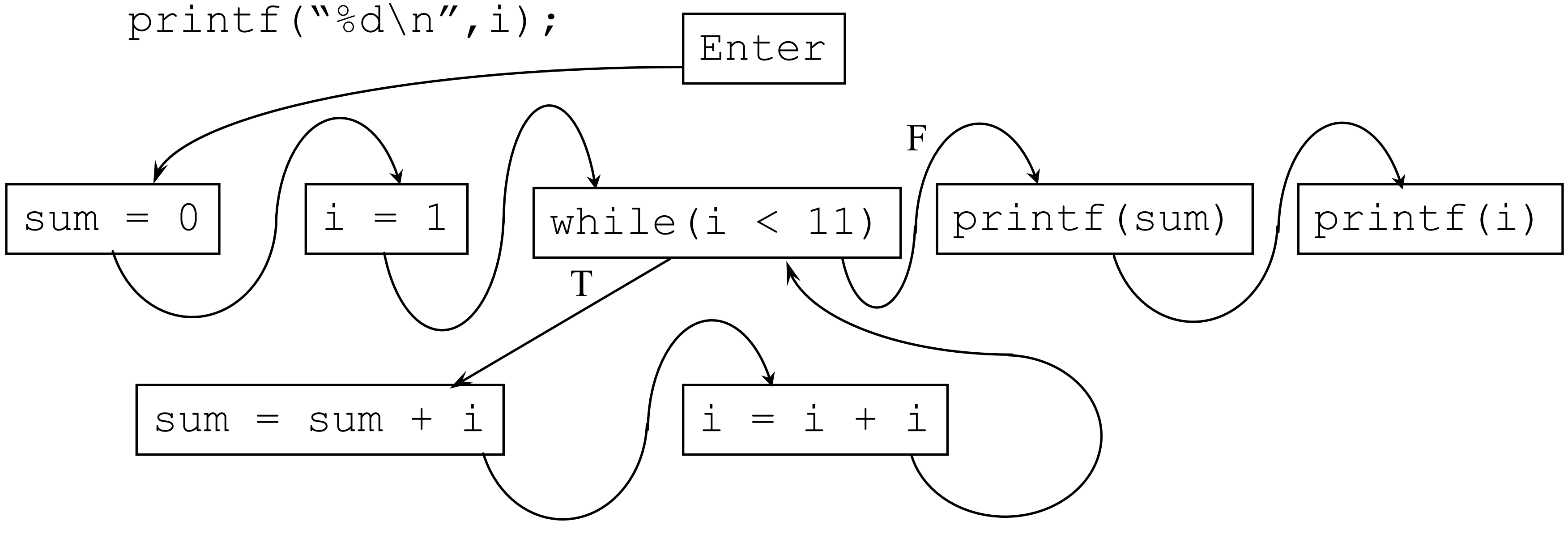
```
void line_char_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
      scan_line(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Line-Count Program

```
void line_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line2(FILE *f, BOOL *bptr, int *iptr);
      scan_line2(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line2(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Control Flow Graph

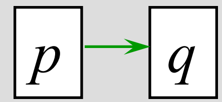
```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```



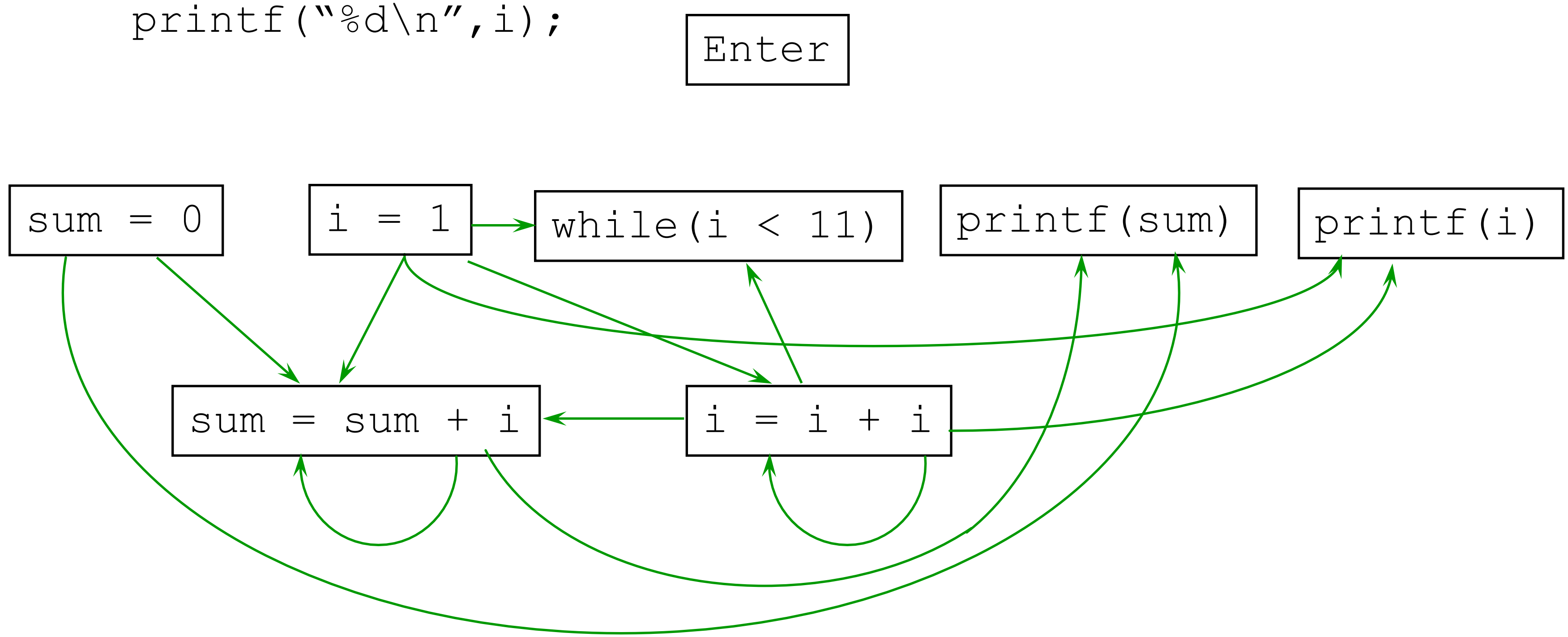
Flow Dependence Graph

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf(“%d\n”,sum);
      printf(“%d\n”,i);
}
```

Flow dependence



Value of variable assigned at *p* may be used at q.



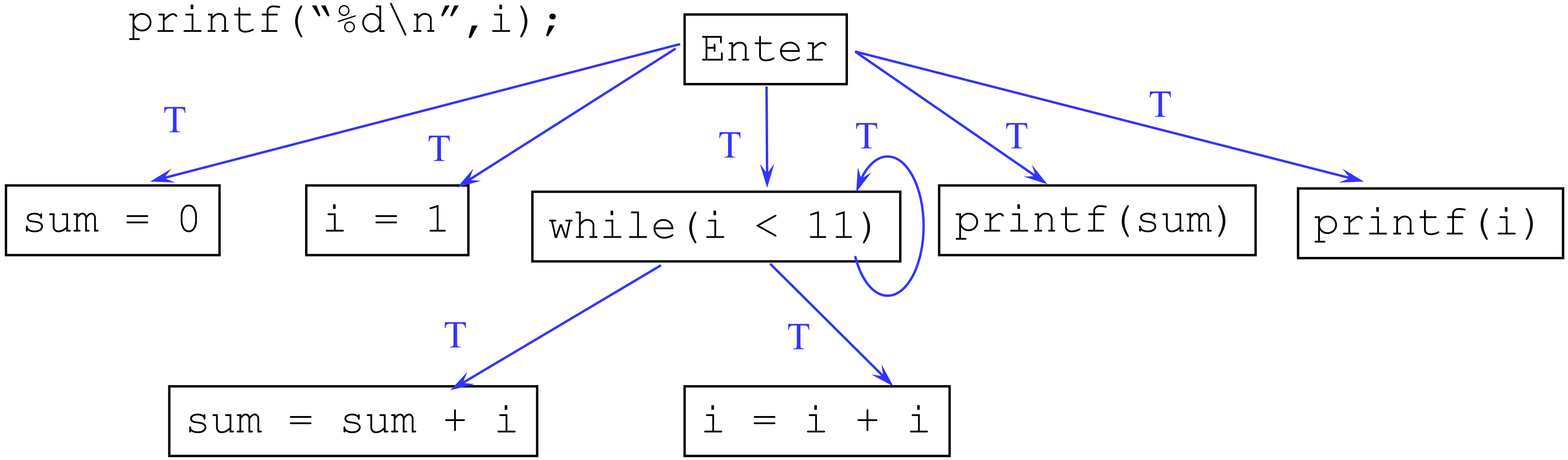
Control Dependence Graph

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf(“%d\n”,sum);
      printf(“%d\n”,i);
}
```

Control dependence

$p \xrightarrow[T]{} q$ q is reached from p if condition p is true (T), not otherwise.

$p \xrightarrow[F]{} q$ Similar for false (F).

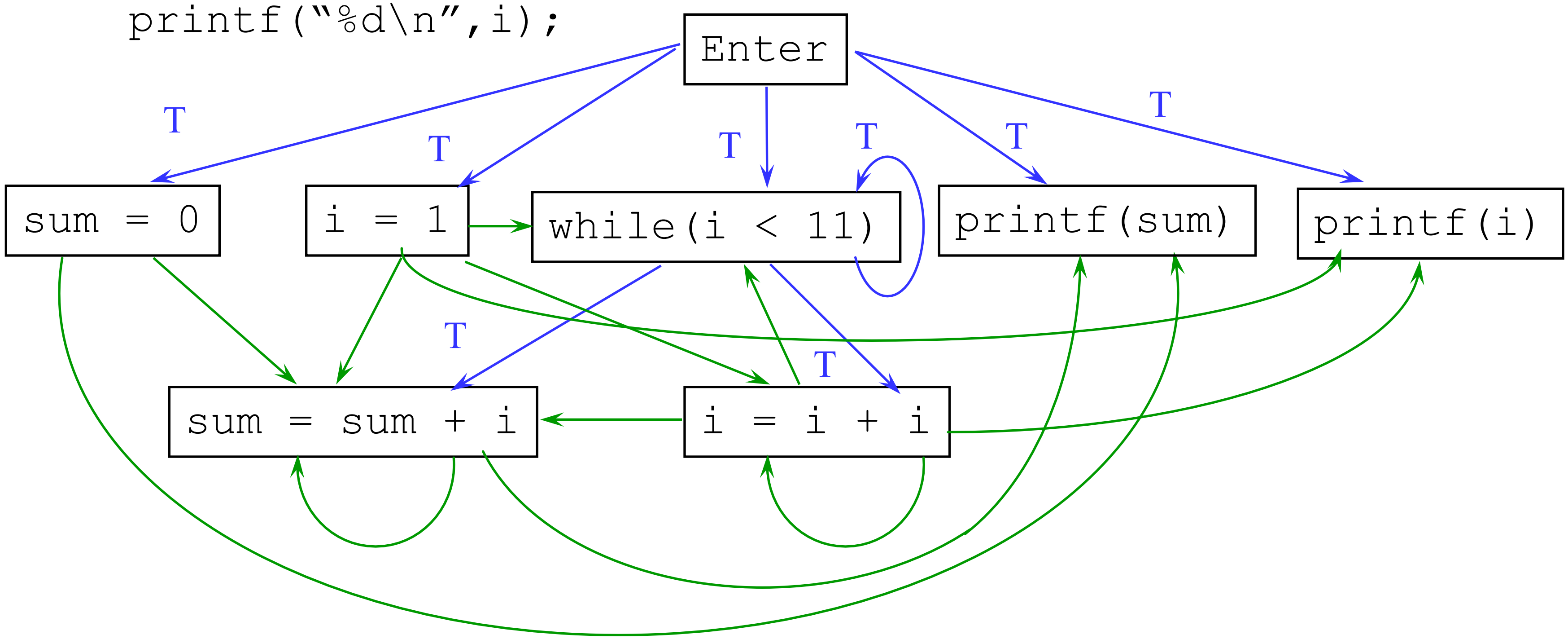


Program Dependence Graph (PDG)

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

Control dependence

Flow dependence

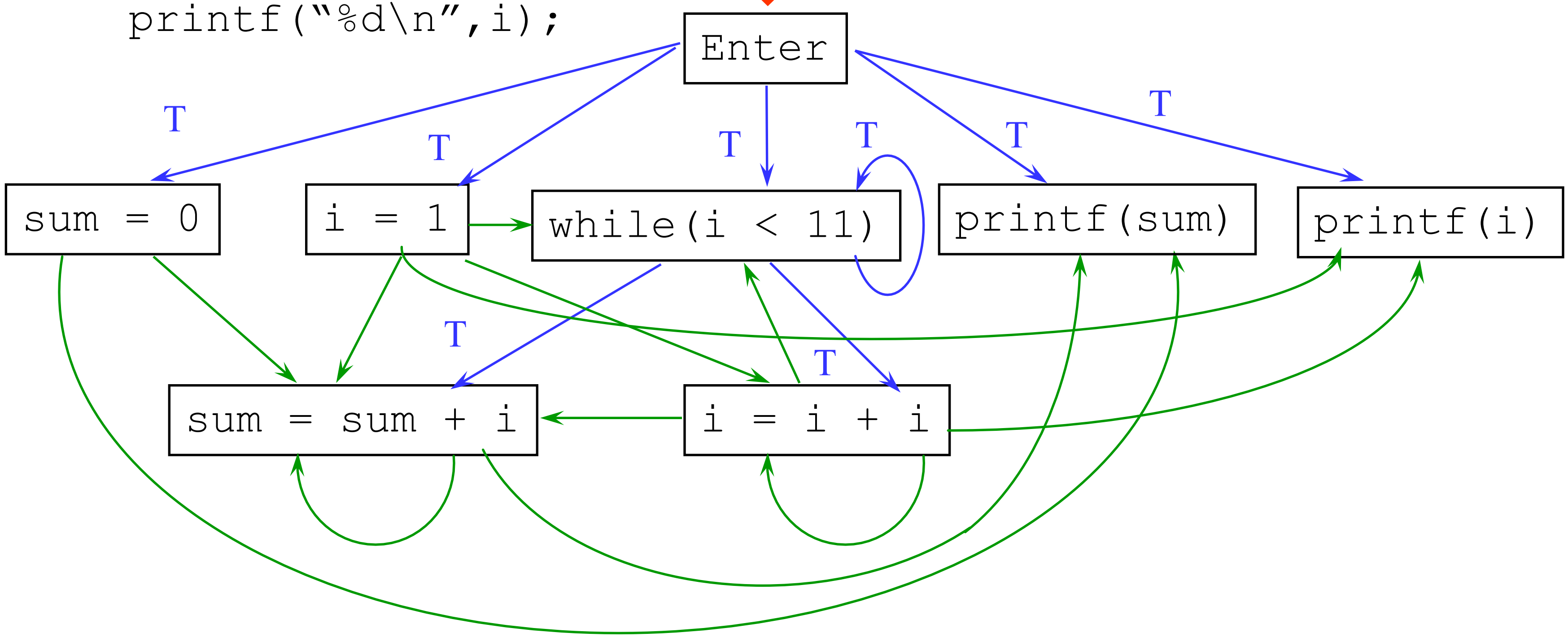


Program Dependence Graph (PDG)

```
int main() {
        int i = 1;
        int sum = 0;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```

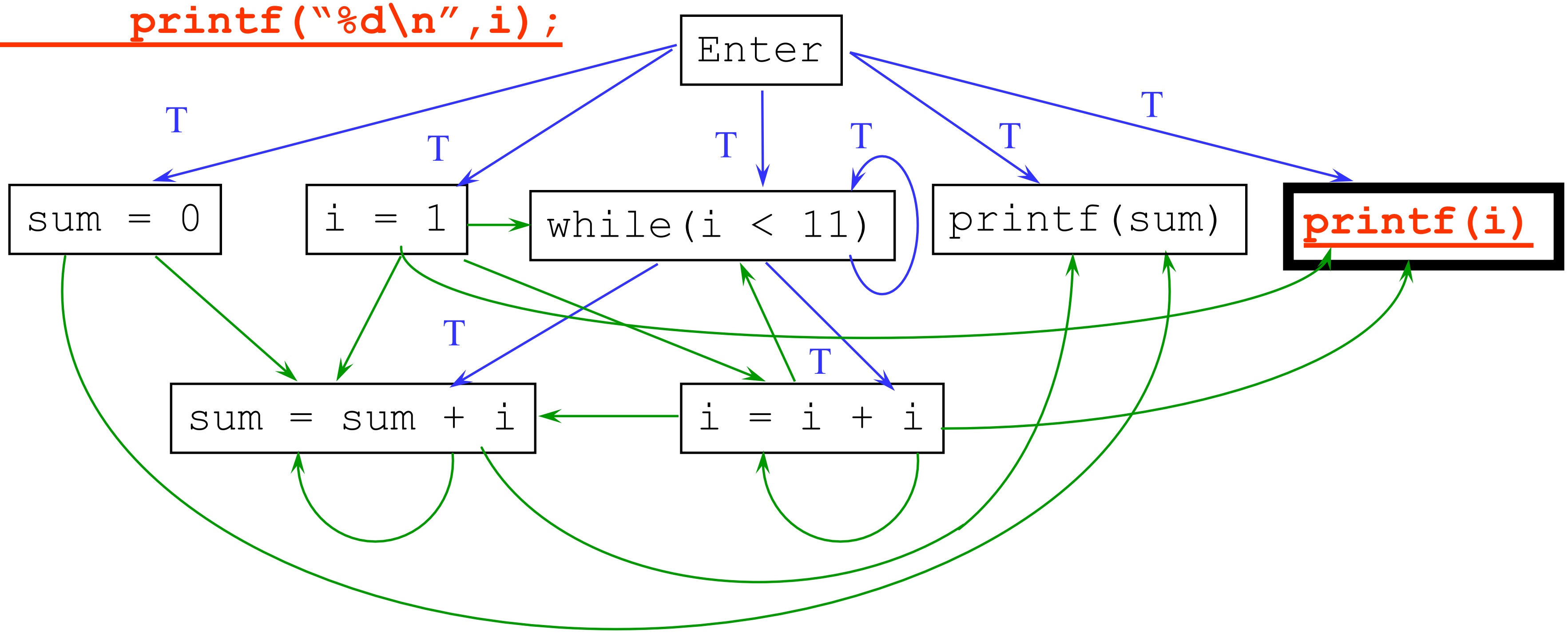
Opposite Order

Same PDG



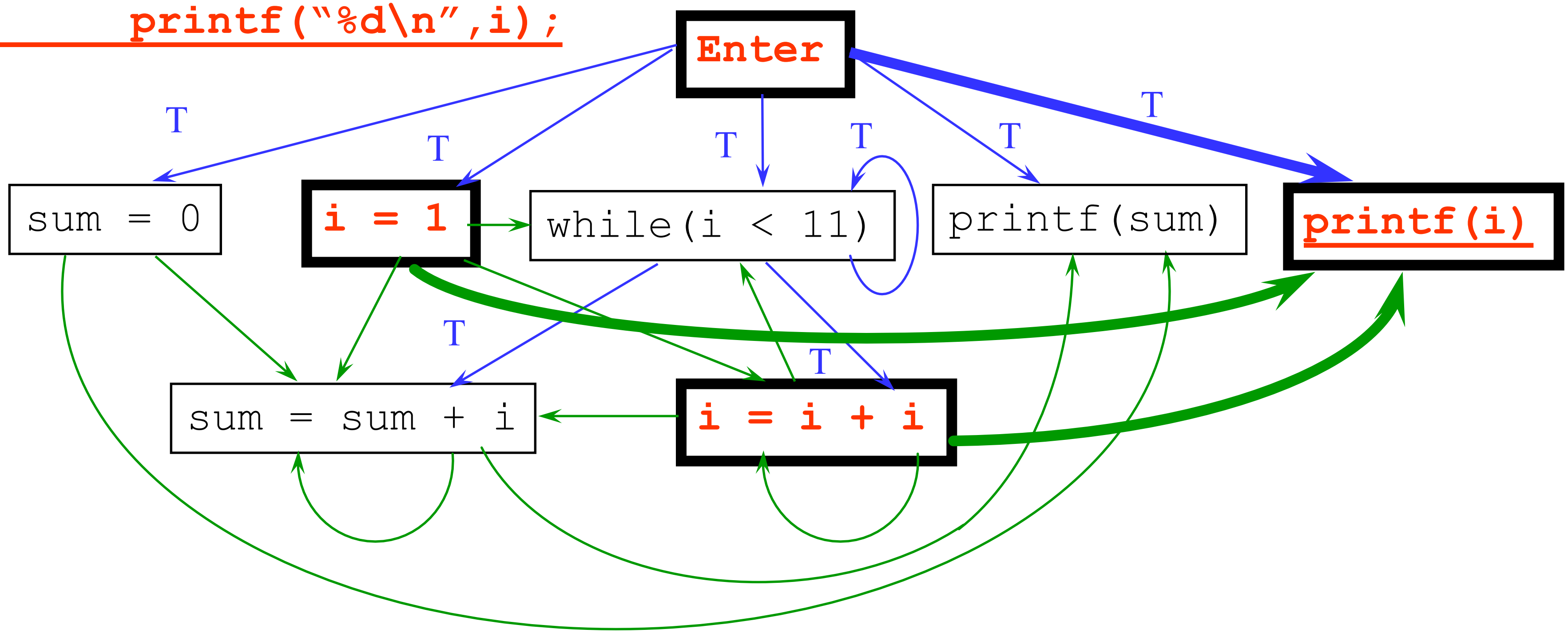
Backward Slice

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```



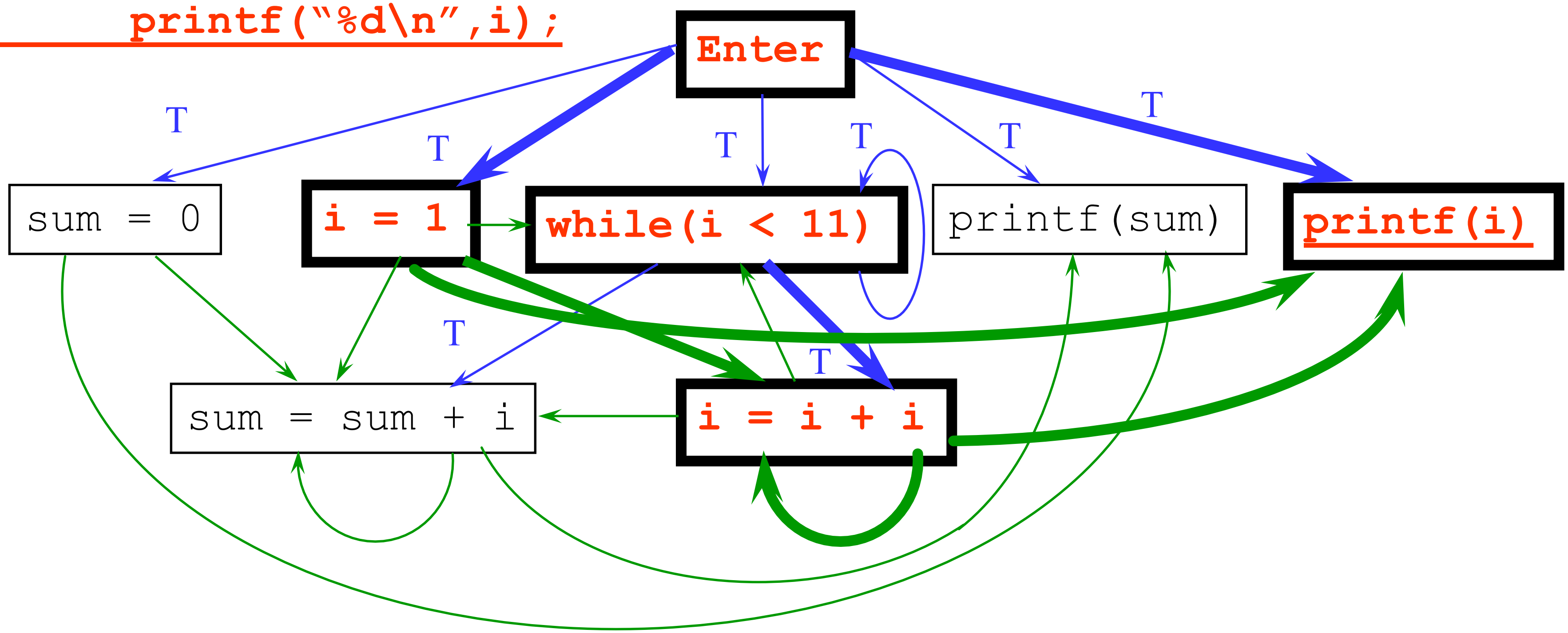
Backward Slice (2)

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```



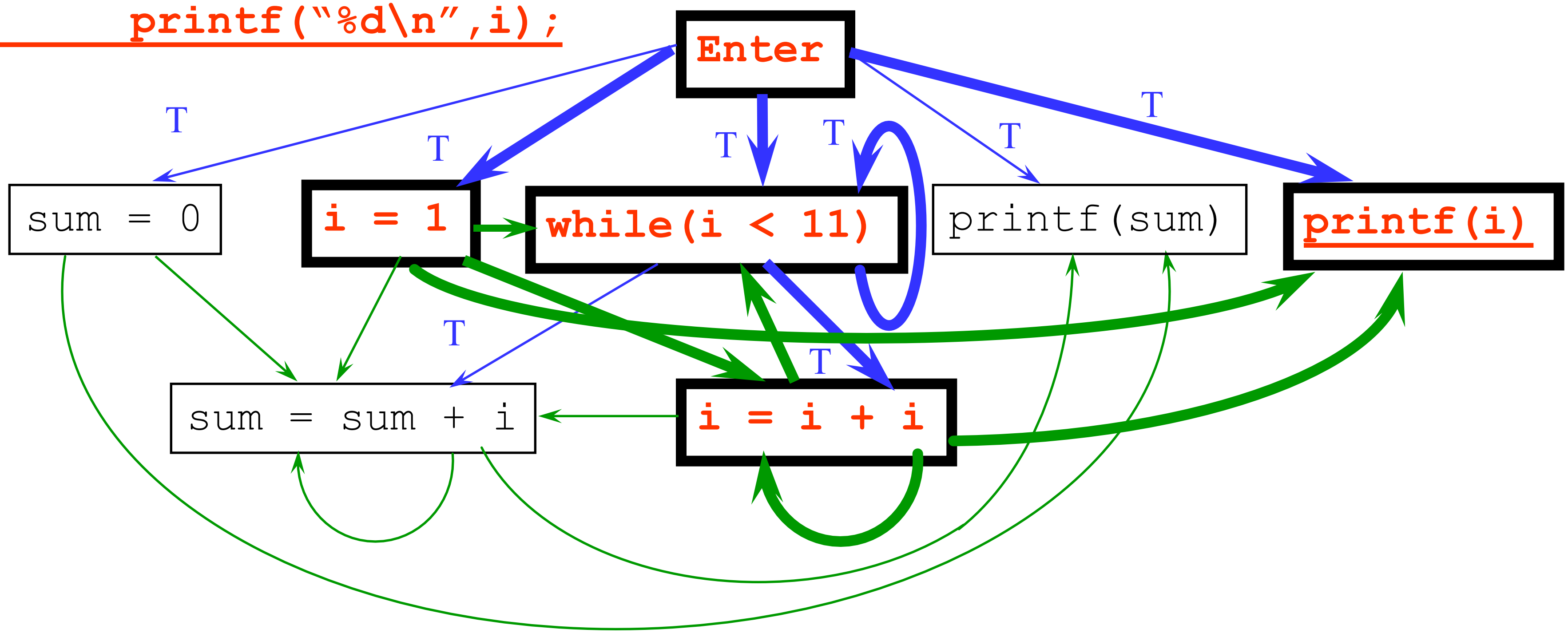
Backward Slice (3)

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```



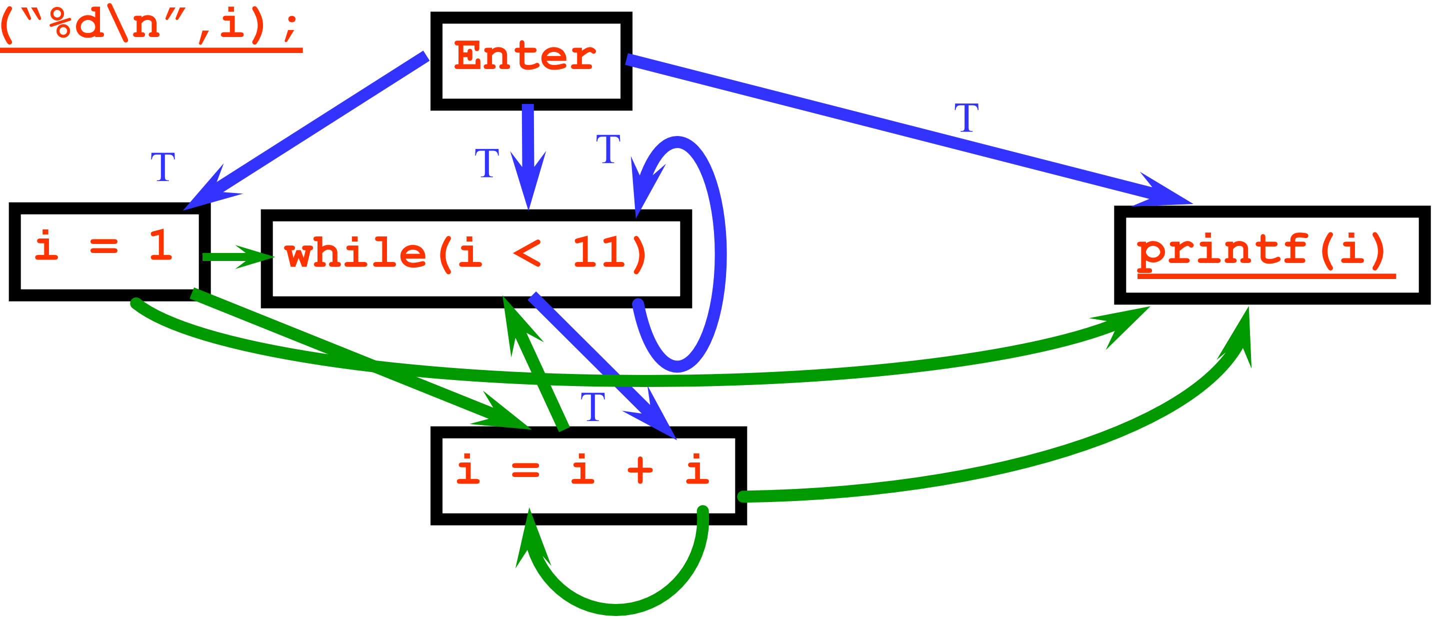
Backward Slice (4)

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```



Slice Extraction

```
int main() {

      int i = 1;
      while (i < 11) {

            i = i + 1;
      }

      printf("%d\n",i);
}
```



compress95.c

File Edit Queries Functions Go Window Help

```
    InBuff = (unsigned char *)from_buf;
    OutBuff = (unsigned char *)to_buf;
    do_decomp = action;

        if (do_decomp == 0) {
                compress();
#ifdef DEBUG
                if(verbose)             dump_tab();
#endif /* DEBUG */
        } else {
            /* Check the magic number */
            if (nomagic == 0) {
                if ((getbyte() != (magic_header[0] & 0xFF))
                 || (getbyte() != (magic_header[1] & 0xFF))) {
                    fprintf(stderr, "stdin: not in compressed format\n");
                    exit(1);
                }
                maxbits = getbyte();    /* set -b from file */
                block_compress = maxbits & BLOCK_MASK;
                maxbits &= BIT_MASK;
                maxmaxcode = 1 << maxbits;
                fsize = 100000;         /* assume stdin large for USERMEM */
                if(maxbits > BITS) {
                        fprintf(stderr,
                        "stdin: compressed with %d bits, can only handle %d bits
                        maxbits, BITS);
                        exit(1);
                }
            }
#ifndef DEBUG
            decompress();
#else
```

*Inter*procedural Slice

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```

Backward slice with respect to "printf("%d\n",i)"

*Inter*procedural Slice

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```

Backward slice with respect to "printf("%d\n",i)"

*Inter*procedural Slice

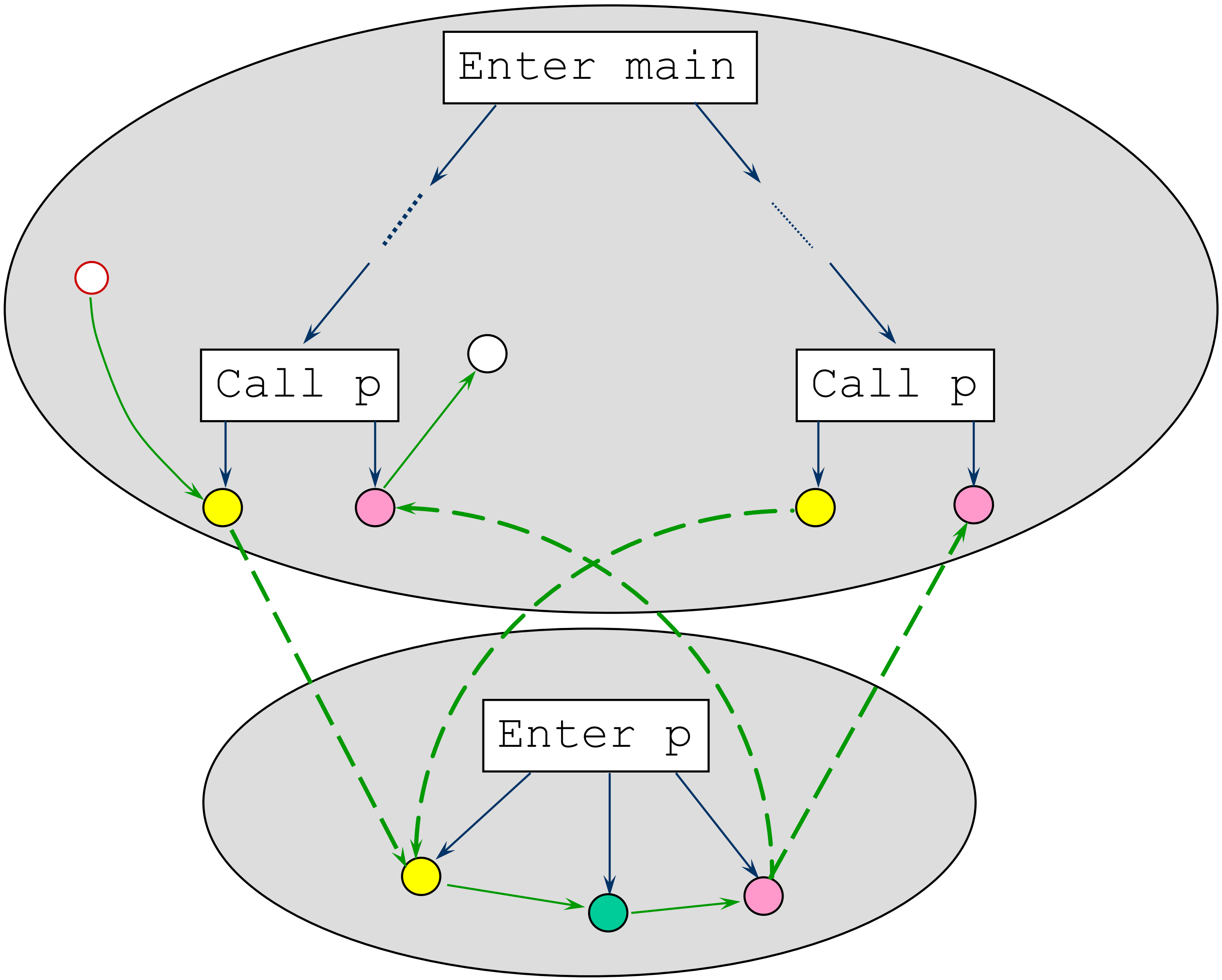
```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum,i);
        i = add(i,1);
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}

int add(int x, int y) {
    return x + y;
}
```

Superfluous components included by Weiser's slicing algorithm [TSE 84]
Left out by algorithm of Horwitz, Reps, & Binkley [PLDI 88; TOPLAS 90]

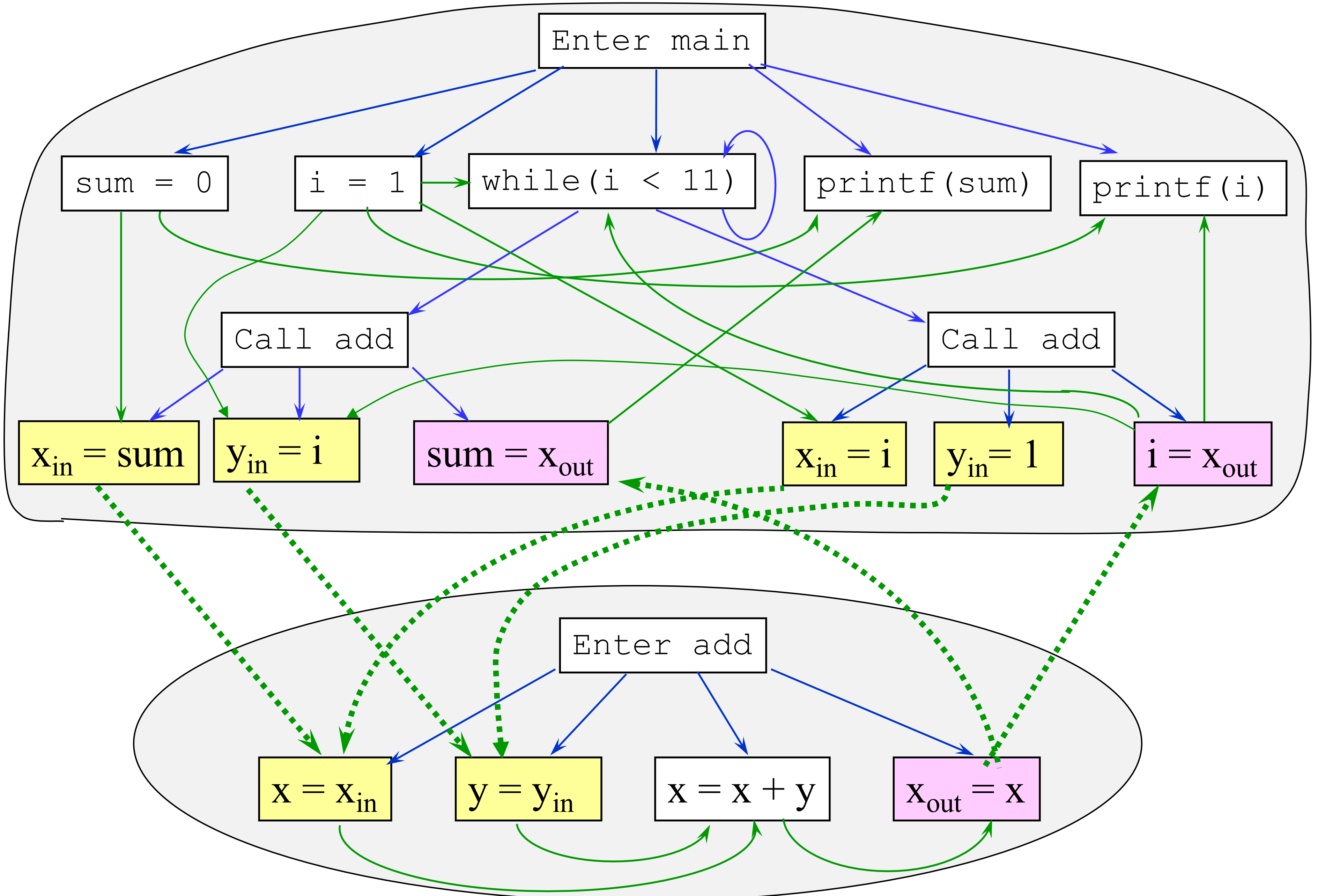
How is an SDG Created?

- Each PDG has nodes for
  - entry point
  - procedure parameters and function result
- Each call site has nodes for
  - call
  - arguments and function result
- Appropriate edges
  - entry node to parameters
  - call node to arguments
  - call node to entry node
  - arguments to parameters

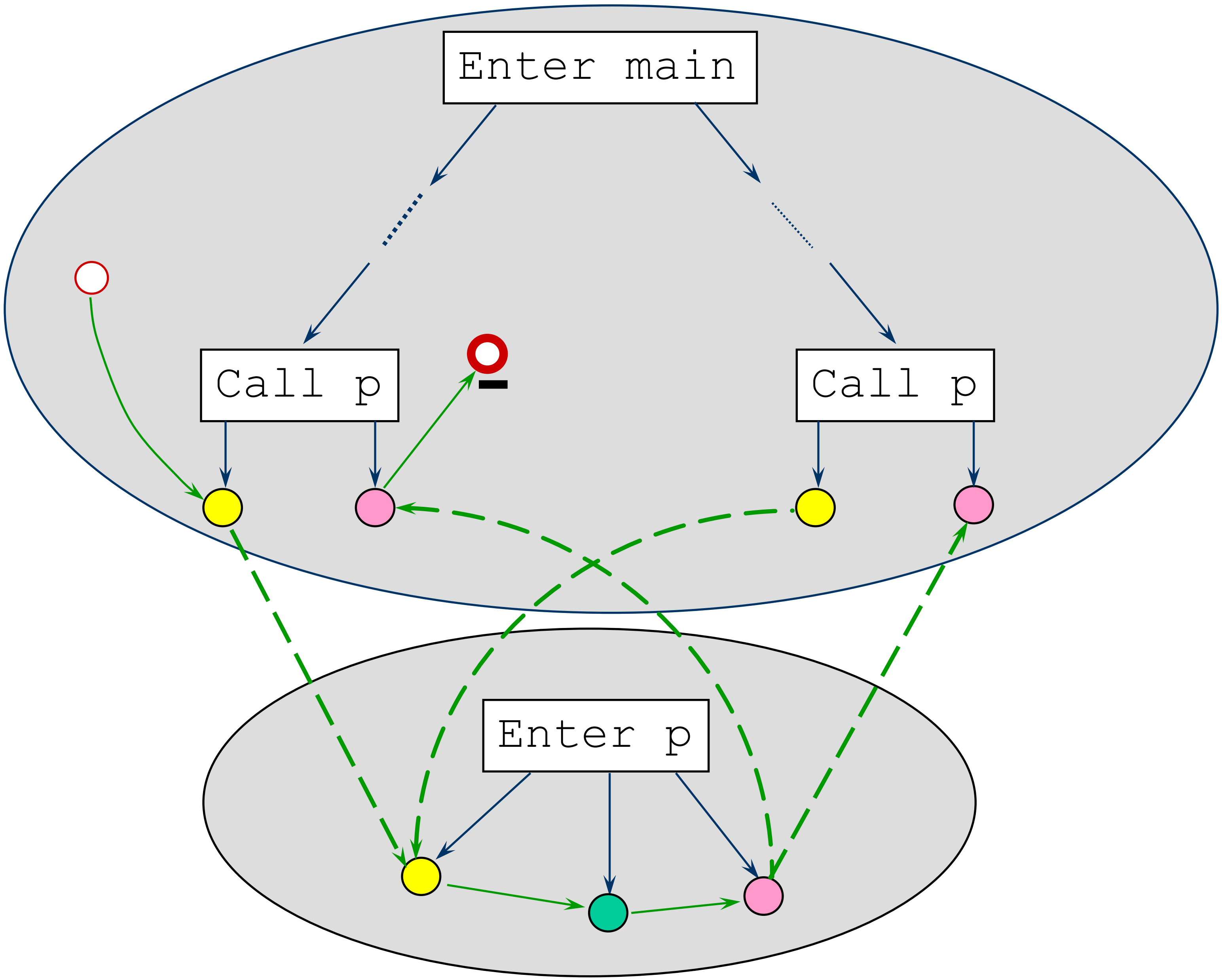
System Dependence Graph (SDG)



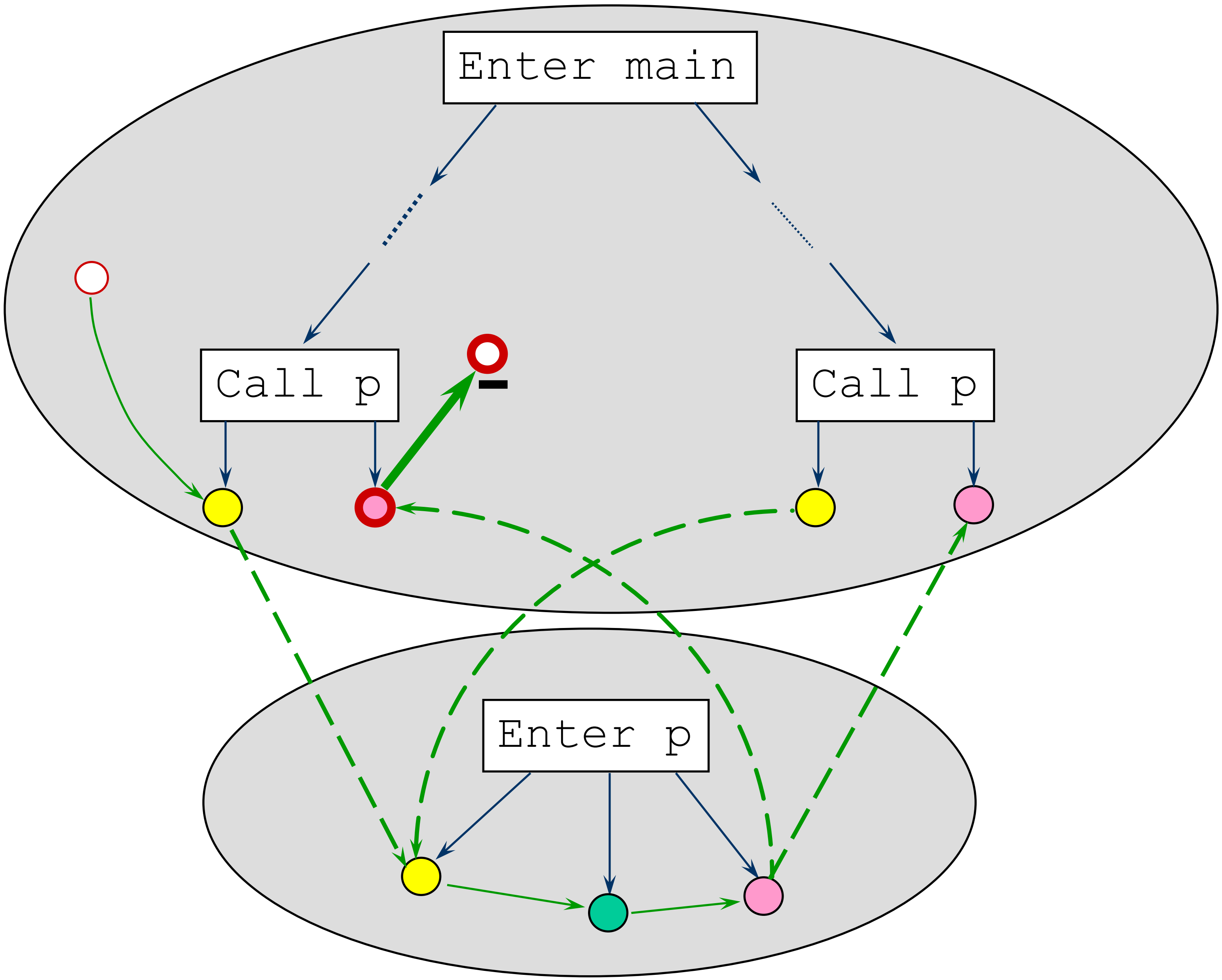
SDG for the Sum Program



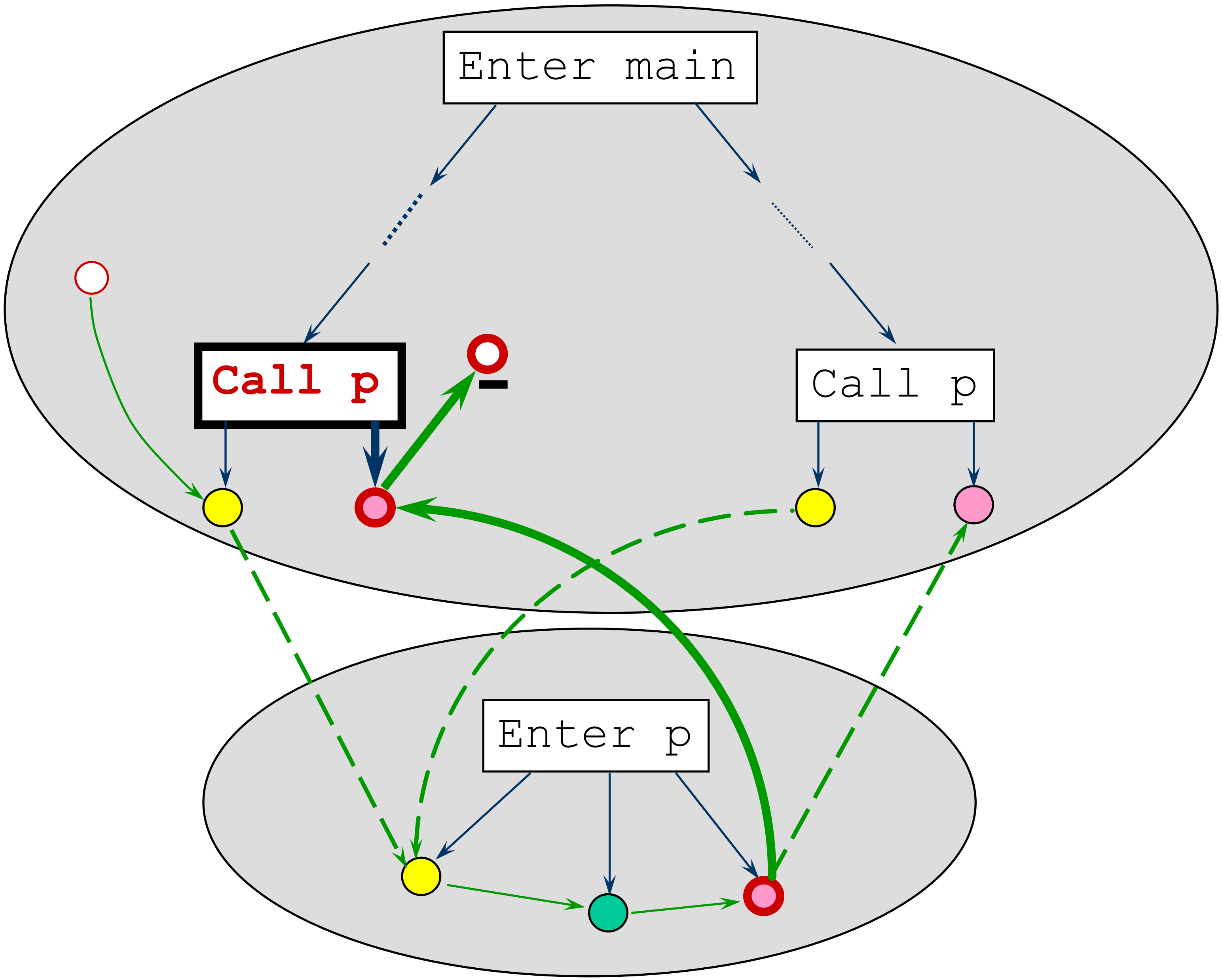
*Inter*procedural Backward Slice



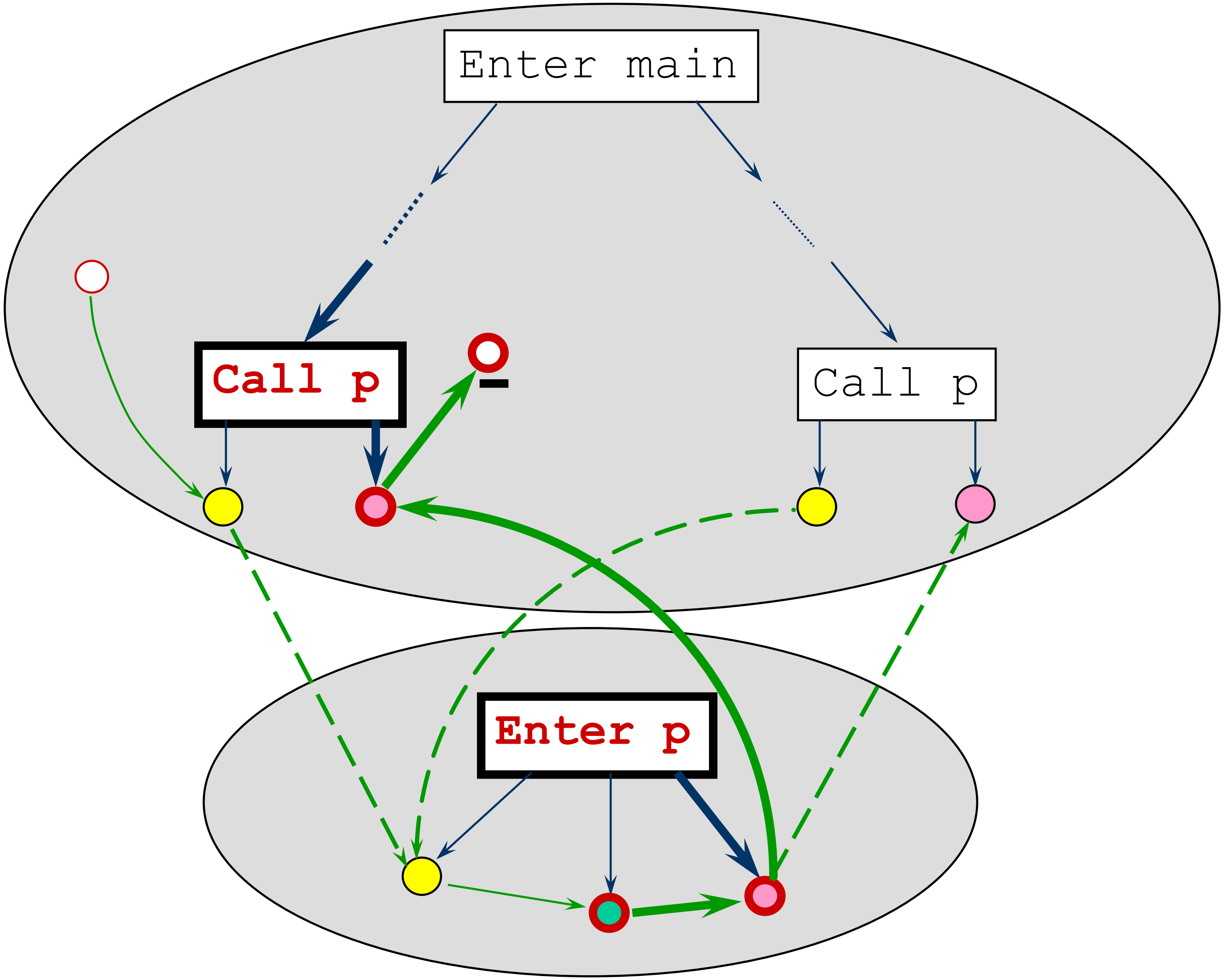
*Inter*procedural Backward Slice (2)



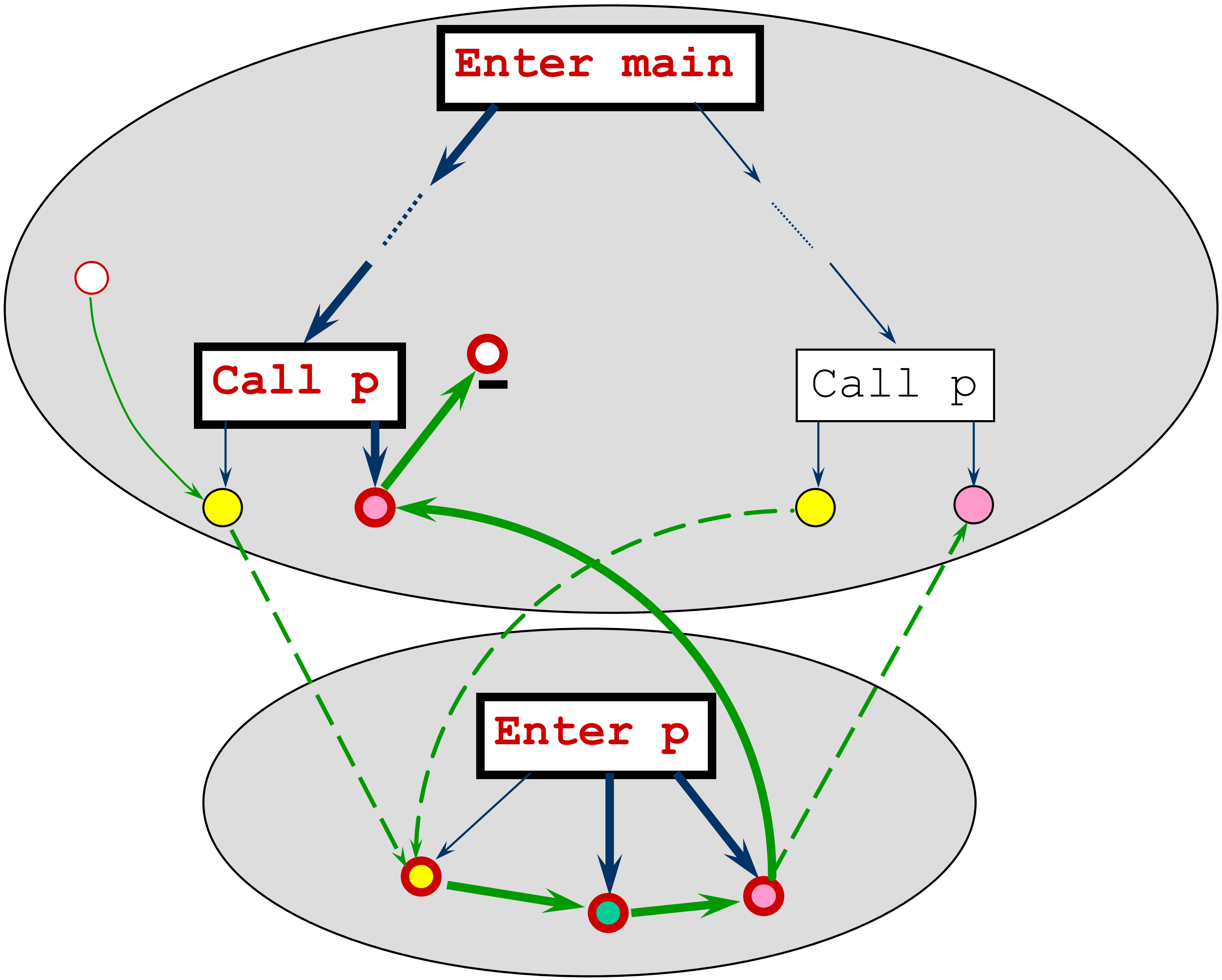
*Inter*procedural Backward Slice (3)



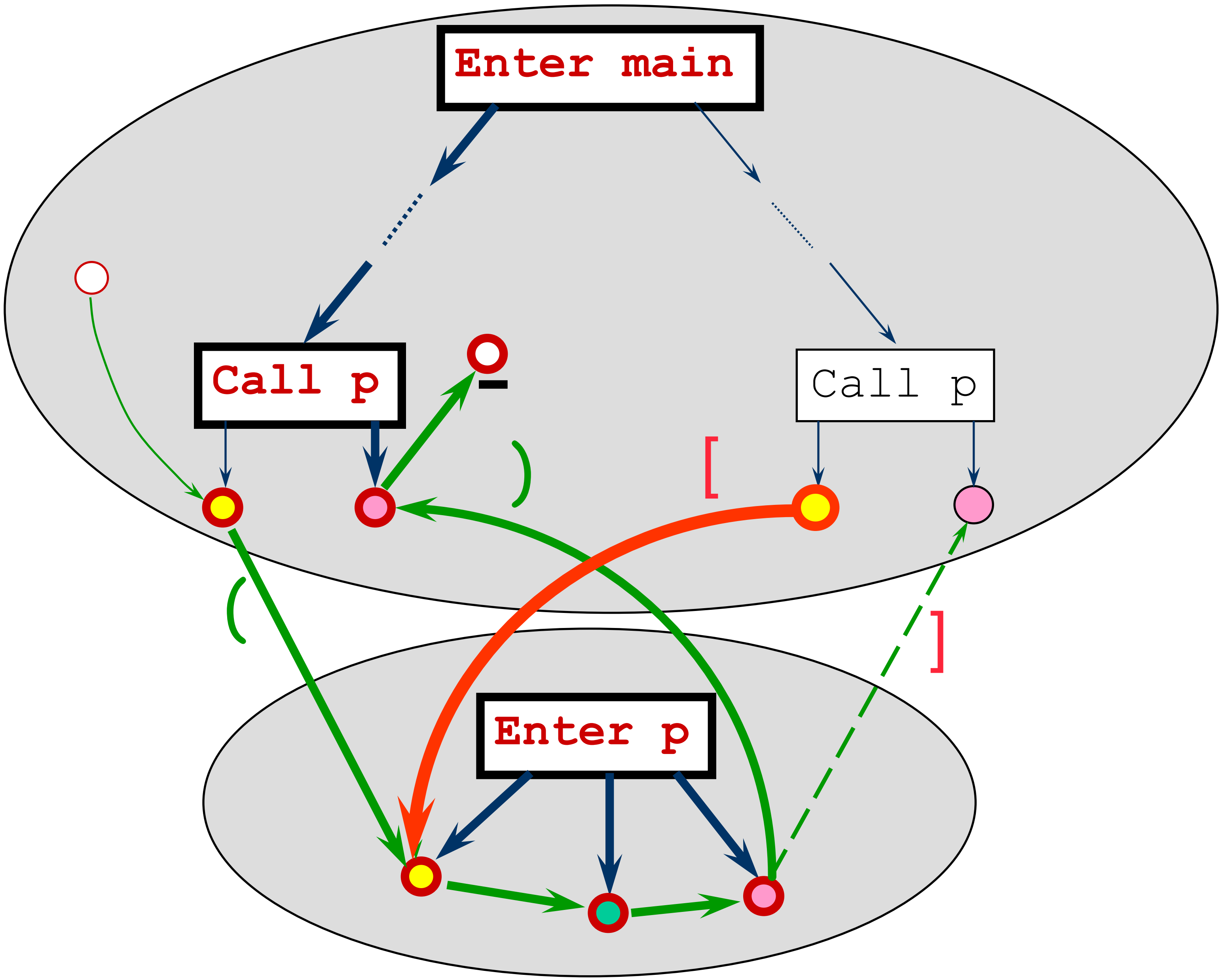
*Inter*procedural Backward Slice (4)



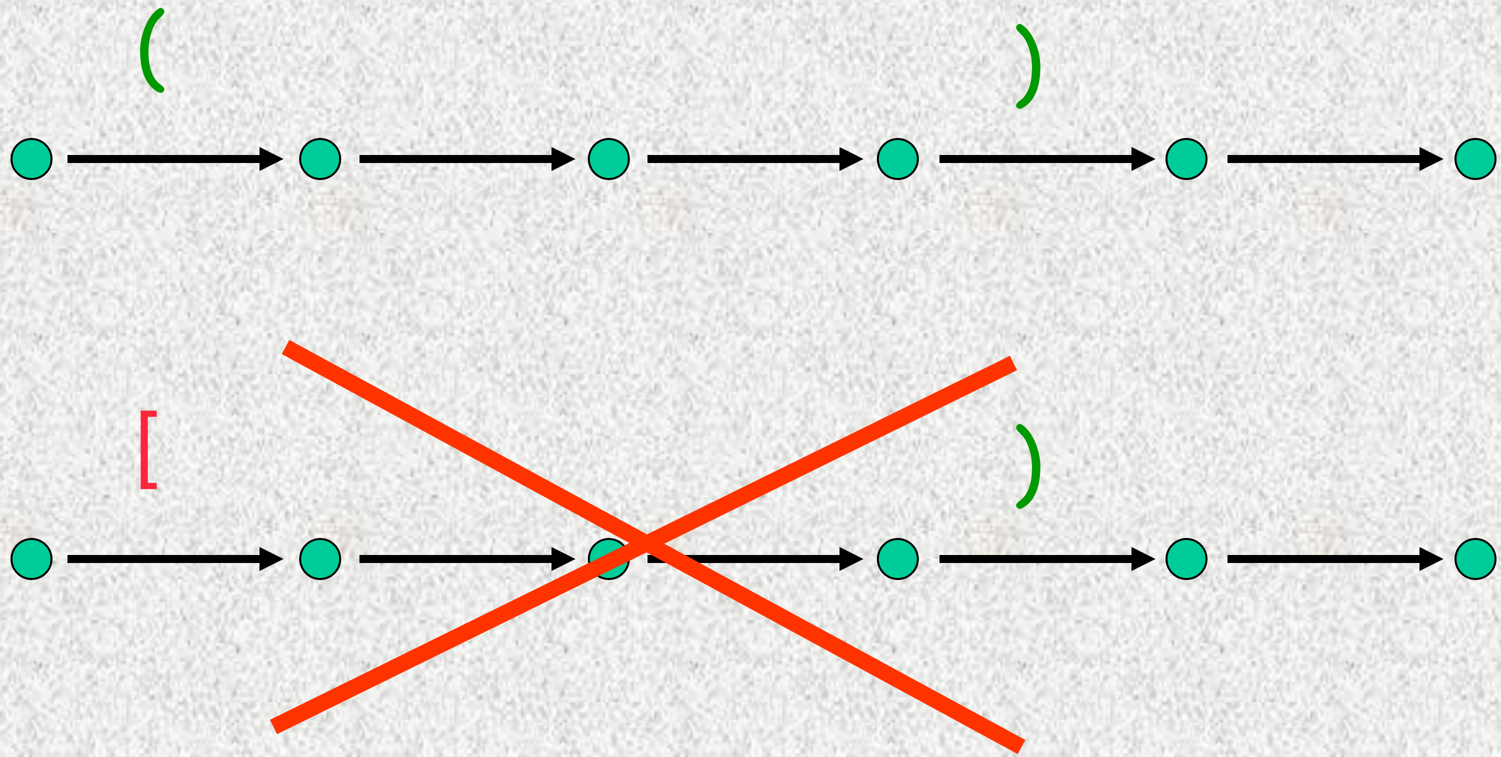
*Inter*procedural Backward Slice (5)



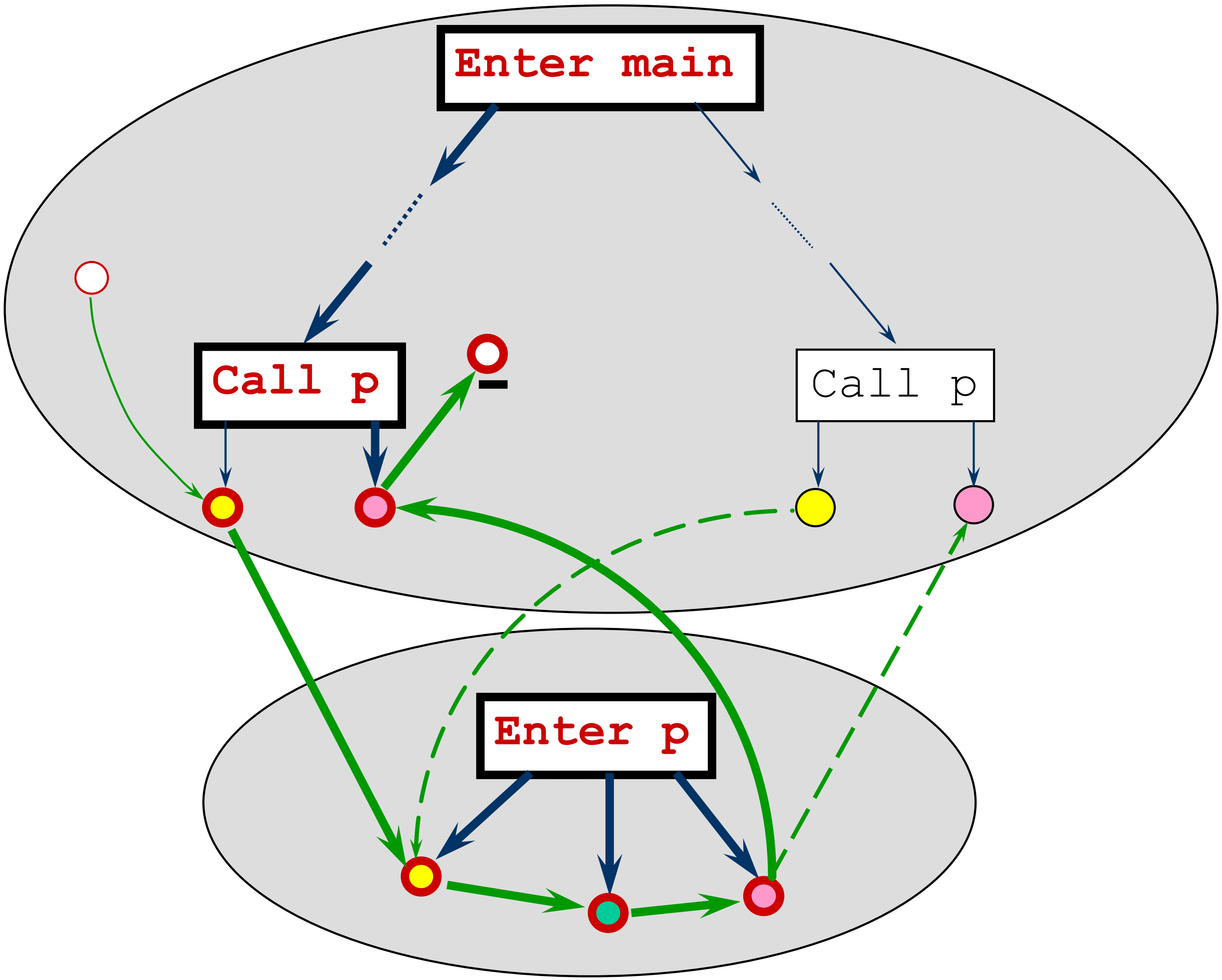
*Inter*procedural Backward Slice (6)



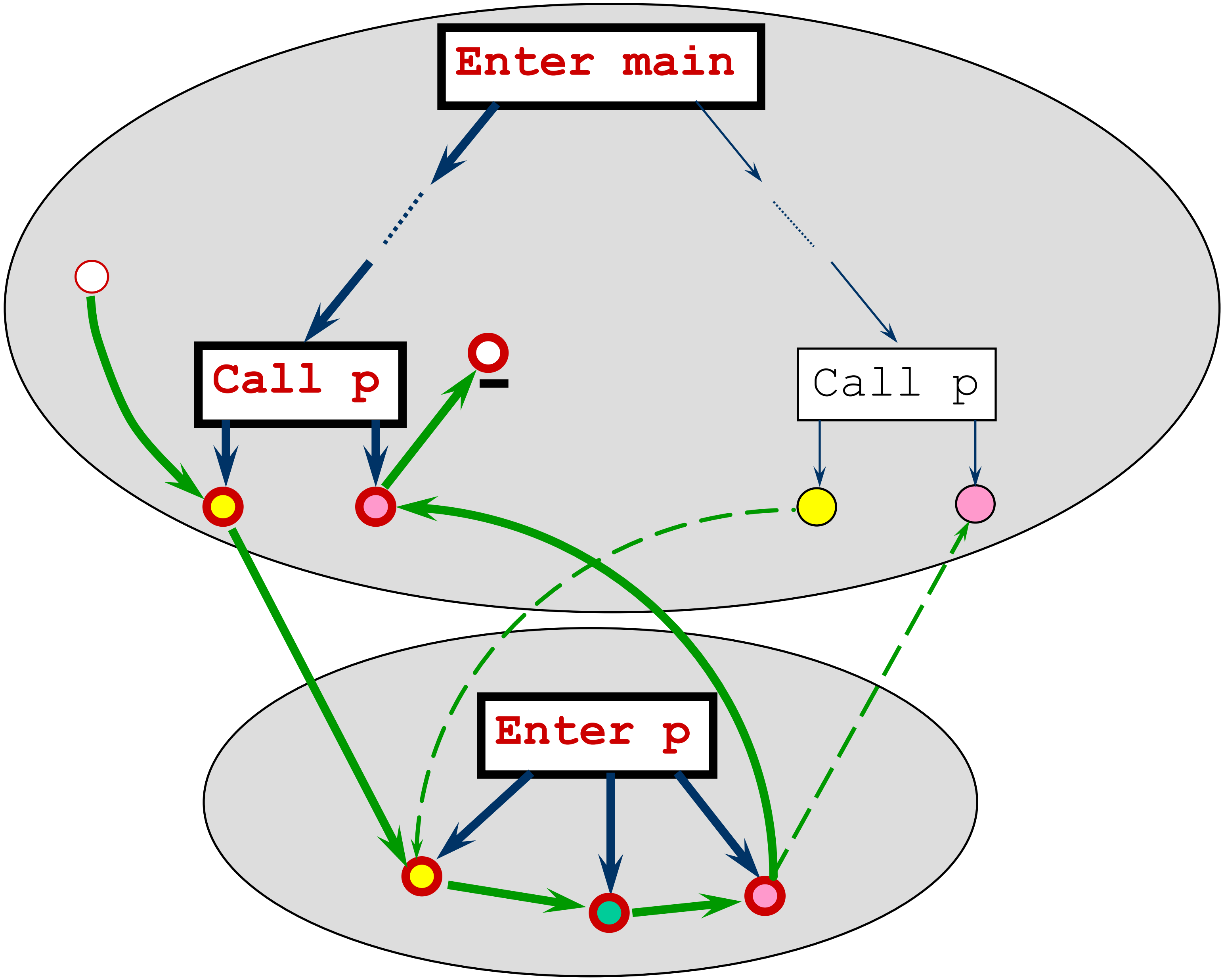
Matched-Parenthesis Path



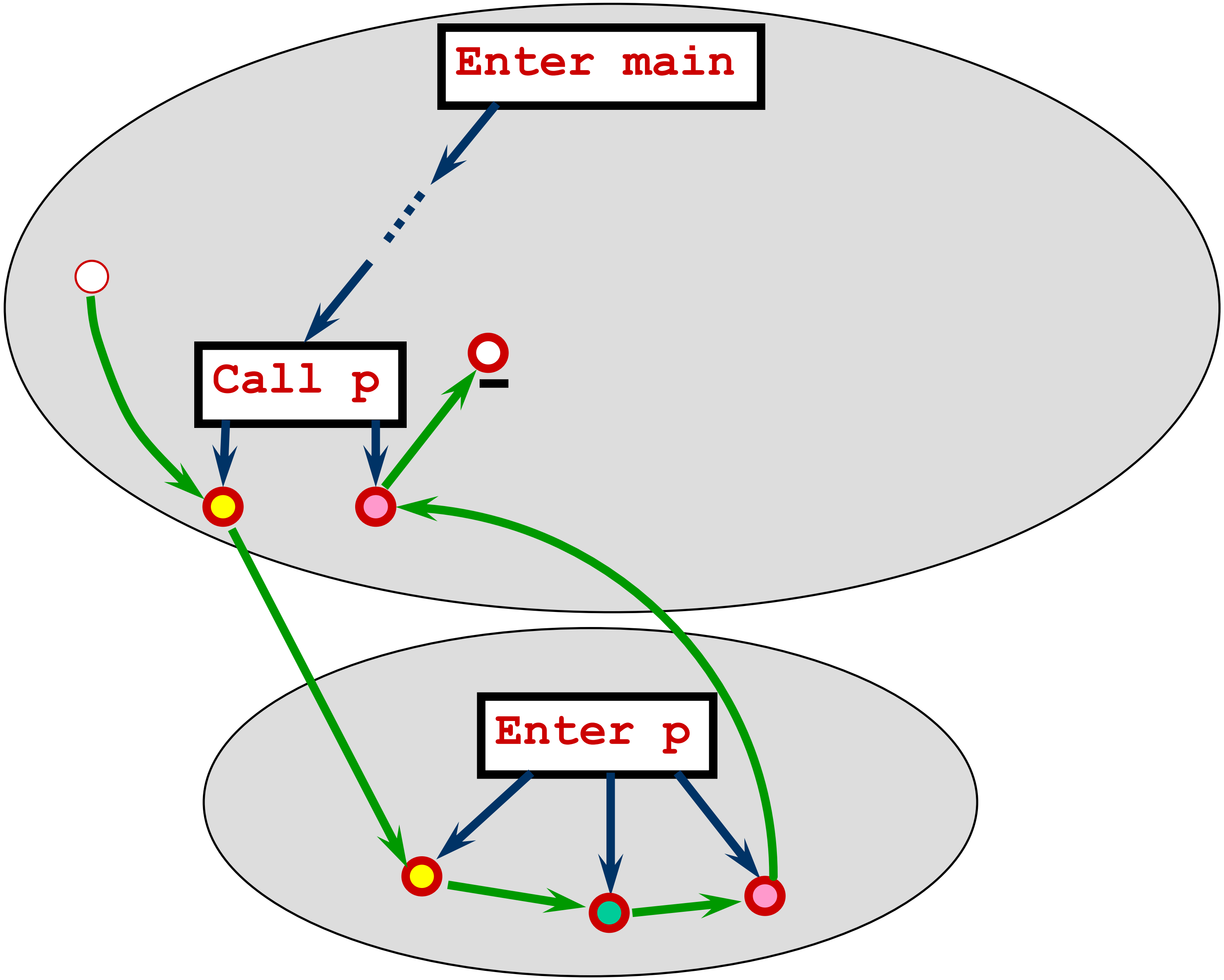
*Inter*procedural Backward Slice (6)



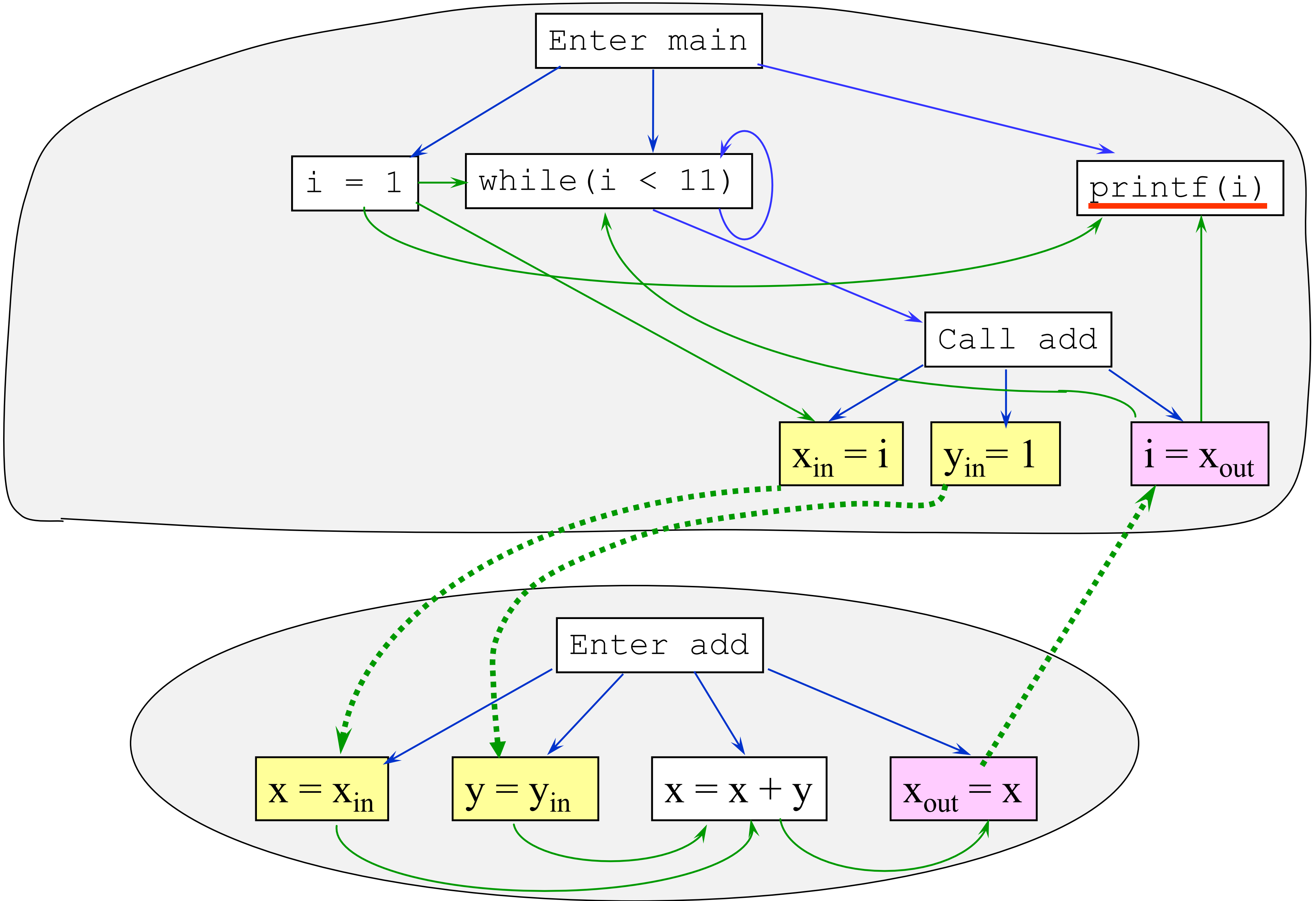
*Inter*procedural Backward Slice (7)



Slice Extraction



Slice of the Sum Program



CFL-Reachability

[Yannakakis 90]

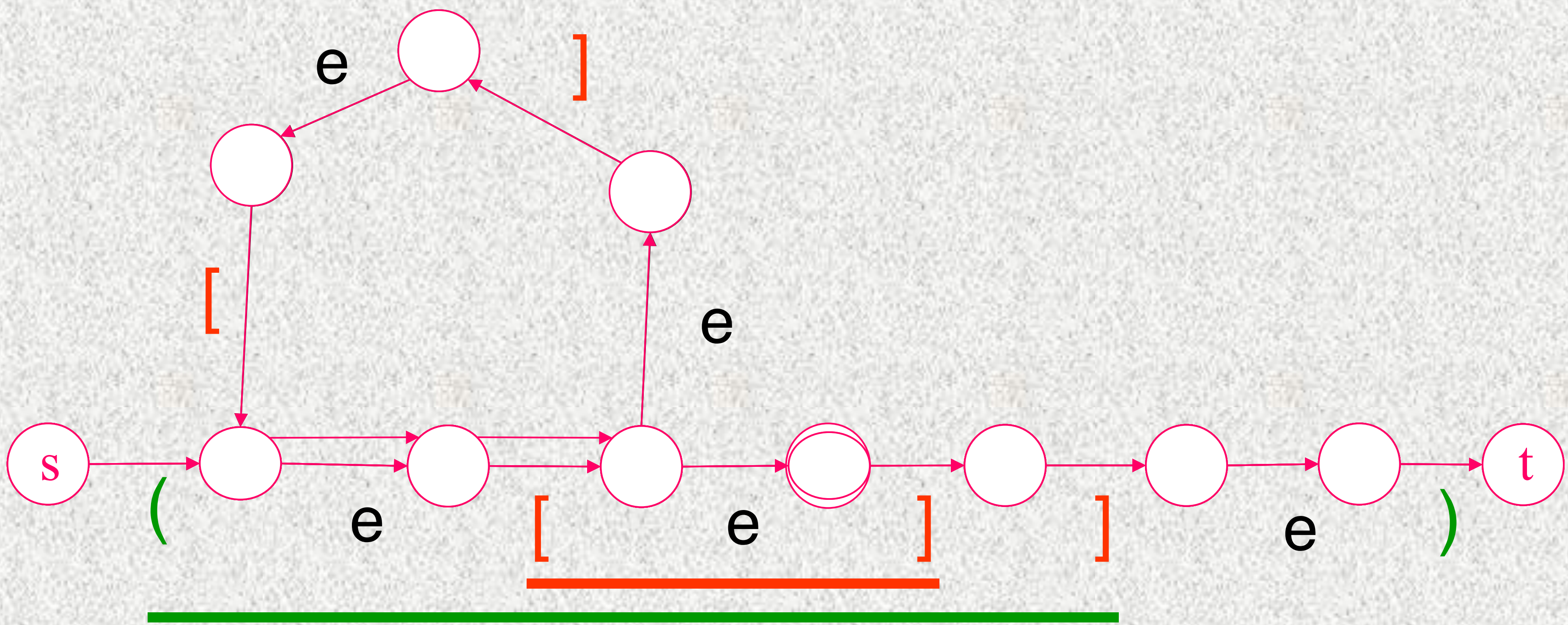
- G: Graph (N nodes, E edges)
- L: A context-free language
- L-path from s to t iff $s \xrightarrow{\alpha}{}^{*} t, \alpha \in L$
- Running time: $O(N^3)$

Interprocedural Slicing via CFL-Reachability

- Graph: System dependence graph
- *L*: *L*(*matched*)   [roughly]
- Node *m* is in the slice w.r.t. *n* iff there is an *L*(*matched*)-path from *m* to *n*

$$
\begin{aligned}
\textit{matched} \rightarrow\ & \varepsilon \\
\mid\ & \mathrm{e} \\
\mid\ & [\ \textit{matched}\ ] \\
\mid\ & (\ \textit{matched}\ ) \\
\mid\ & \textit{matched}\ \ \textit{matched}
\end{aligned}
$$

(e [e] e [e [e]] e)

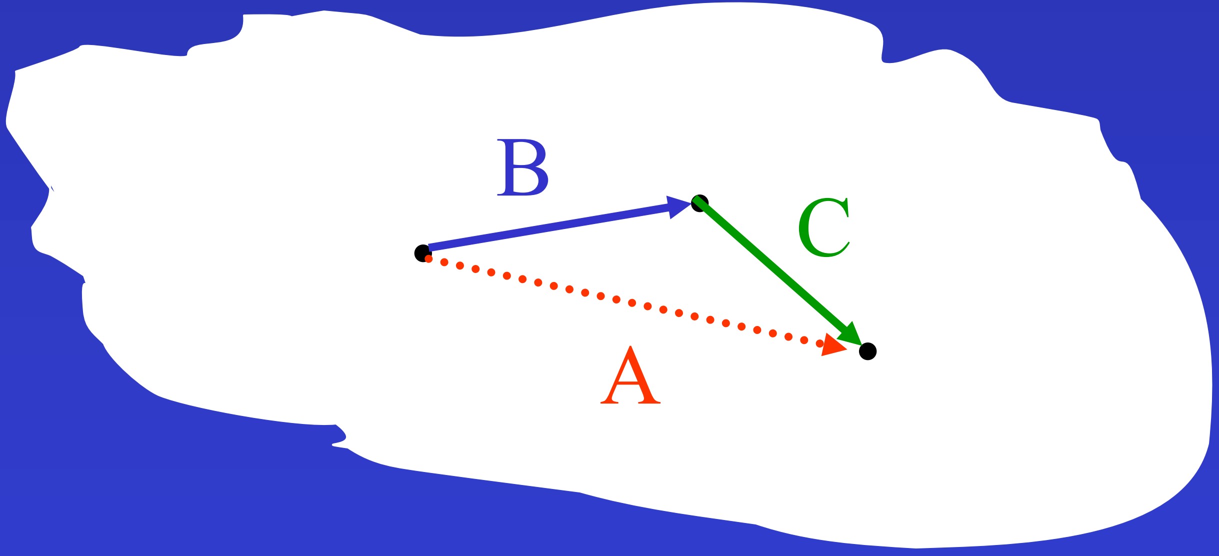


Ordinary Graph Reachability

CFL-Reachability

CFL-Reachability via Dynamic Programming

Graph



Grammar

$$\vdots$$
$$A \to B\ C$$
$$\vdots$$

Degenerate Case: CFL-Recognition

$$exp \;\rightarrow\; id \;|\; exp + exp \;|\; exp * exp \;|\; (\, exp \,)$$

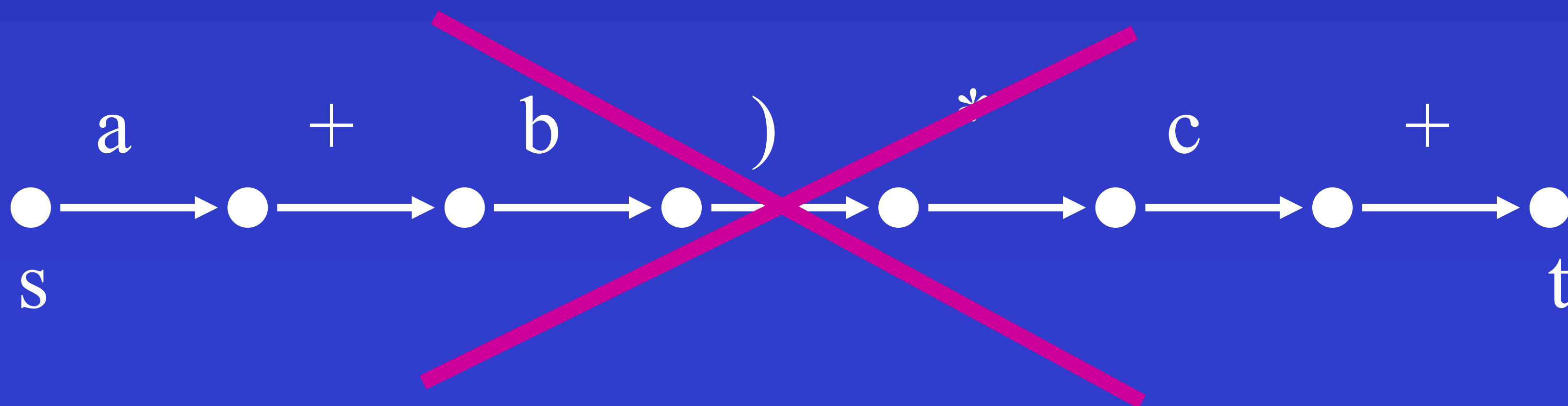
$$\text{“(a + b) * c”} \in L(exp) \,?$$

$$s \bullet \xrightarrow{(} \bullet \xrightarrow{a} \bullet \xrightarrow{+} \bullet \xrightarrow{b} \bullet \xrightarrow{)} \bullet \xrightarrow{*} \bullet \xrightarrow{c} \bullet\, t$$

Degenerate Case: CFL-Recognition

$$exp \rightarrow id \mid exp + exp \mid exp * exp \mid (exp)$$

"a + b) * c +" $\in L(exp)$?



CYK: Context-Free Recognition

$$
\begin{aligned}
M \rightarrow\ & M\ M \\
\mid\ & (\ M\ ) \\
\mid\ & [\ M\ ] \\
\mid\ & (\ ) \\
\mid\ & [\ ]
\end{aligned}
$$

$\boxed{?}$ = "([ ]) [ ]"

Is $\boxed{?} \in L(M)$?

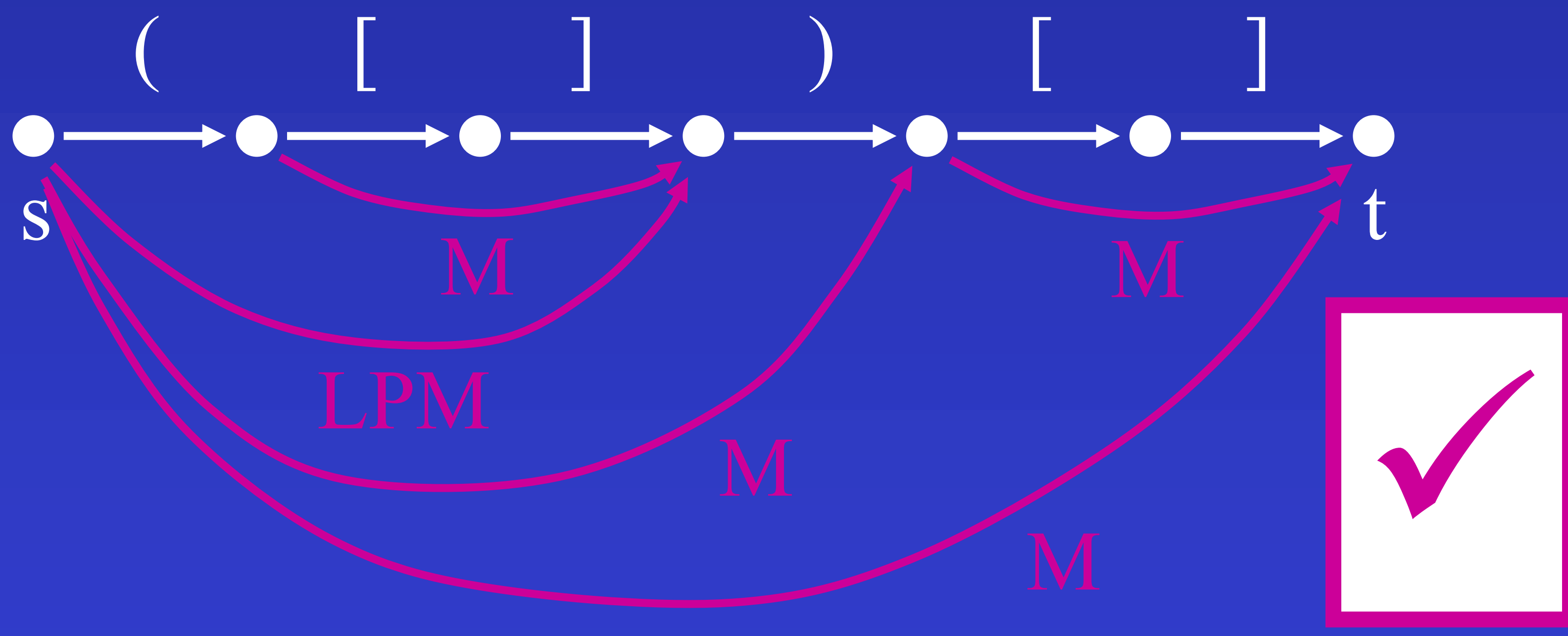
CYK: Context-Free Recognition

$$
\begin{aligned}
M &\rightarrow M\ M \\
&\mid\ (\ M\ ) \\
&\mid\ [\ M\ ] \\
&\mid\ (\ ) \\
&\mid\ [\ ]
\end{aligned}
$$

$$
\begin{aligned}
M &\rightarrow M\ M \\
&\mid\ LPM\ ) \\
&\mid\ LBM\ ] \\
&\mid\ (\ ) \\
&\mid\ [\ ] \\
LPM &\rightarrow (\ M \\
LBM &\rightarrow [\ M
\end{aligned}
$$

CYK

Is "([ ]) [ ]" $\in L(M)$?

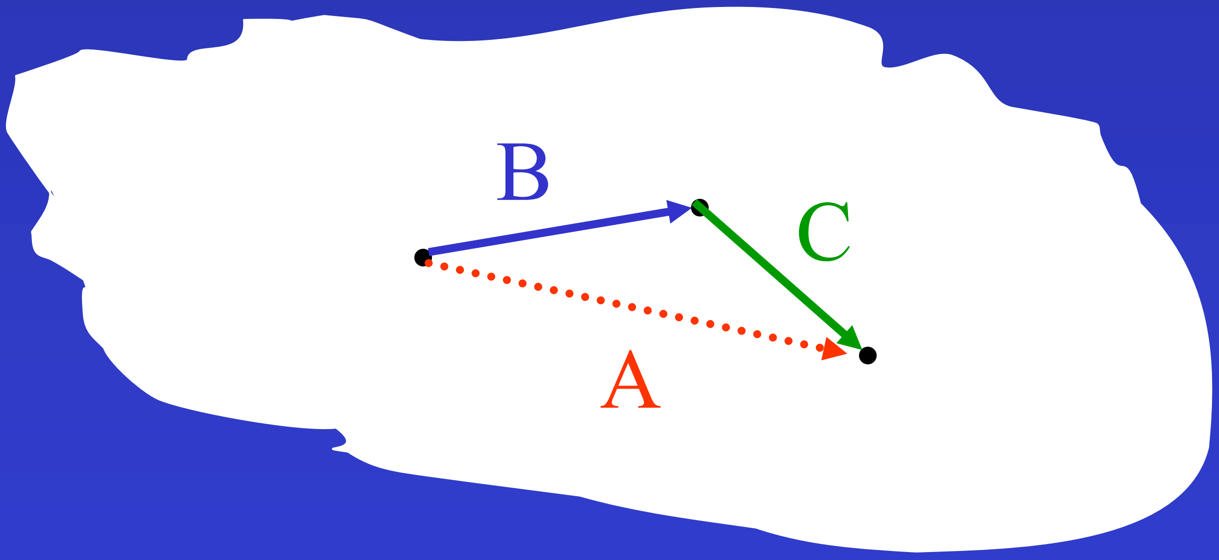


$M \rightarrow M\ M \quad | \quad LPM\ ) \quad | \quad LBM\ ] \quad | \quad (\ ) \quad | \quad [\ ]$

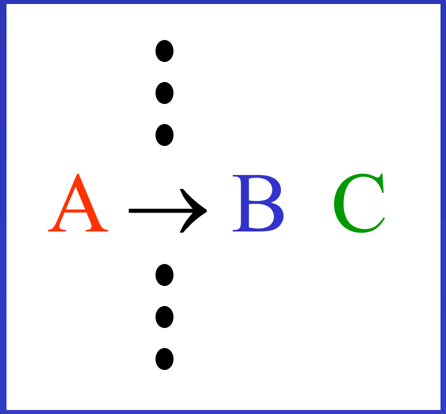
$LPM \rightarrow (\ M \qquad LBM \rightarrow [\ M$

CFL-Reachability via Dynamic Programming

Graph



Grammar

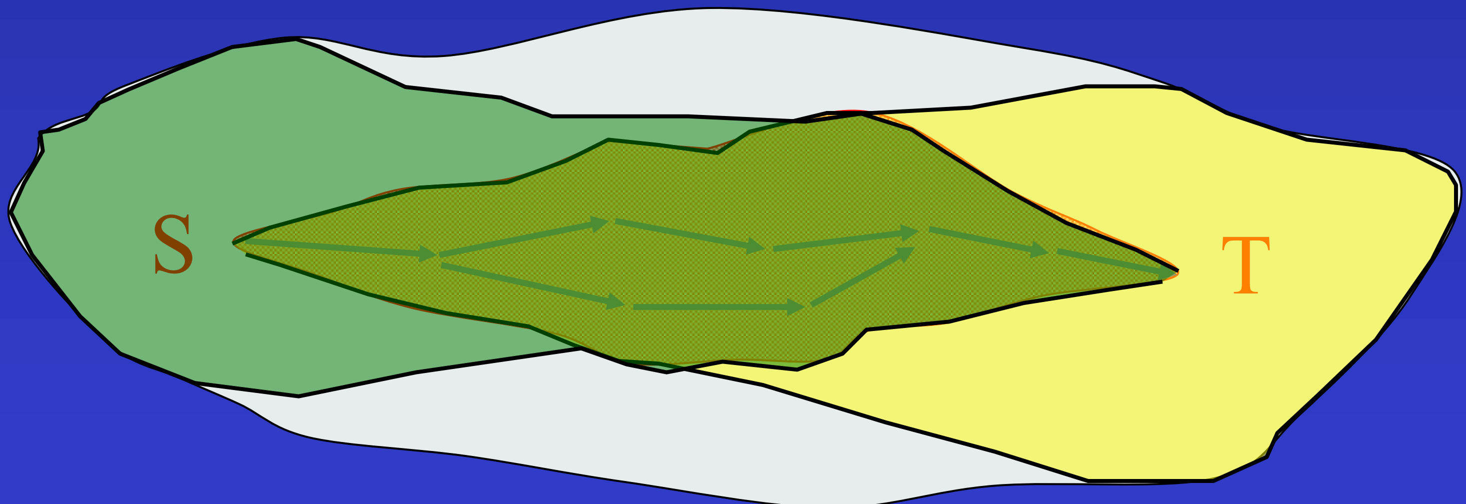


Dynamic Transitive Closure ?!

- Aiken et al.
  - Set-constraint solvers
  - Points-to analysis
- Henglein et al.
  - type inference
- But a CFL captures a **non-transitive** reachability relation [Valiant 75]

Program Chopping

Given source *S* and target *T*, what program points transmit effects from *S* to *T*?



Intersect forward slice from *S* with backward slice from *T*, right?

Non-Transitivity and Slicing

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```

Forward slice with respect to "sum = 0"

Non-Transitivity and Slicing

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```

Forward slice with respect to "sum = 0"

Non-Transitivity and Slicing

```
int main() {
     int sum = 0;
     int i = 1;
     while (i < 11) {
          sum = add(sum,i);
          i = add(i,1);
     }
     printf("%d\n",sum);
     printf("%d\n",i);
}
```

```
int add(int x, int y) {
     return x + y;
}
```

Backward slice with respect to "printf("%d\n",i)"

Non-Transitivity and Slicing

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```

Backward slice with respect to "printf("%d\n",i)"

Non-Transitivity and Slicing

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```

Forward slice with respect to "sum = 0"

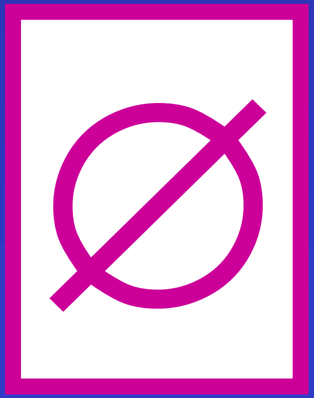
∩

Backward slice with respect to "printf("%d\n",i)"

Non-Transitivity and Slicing

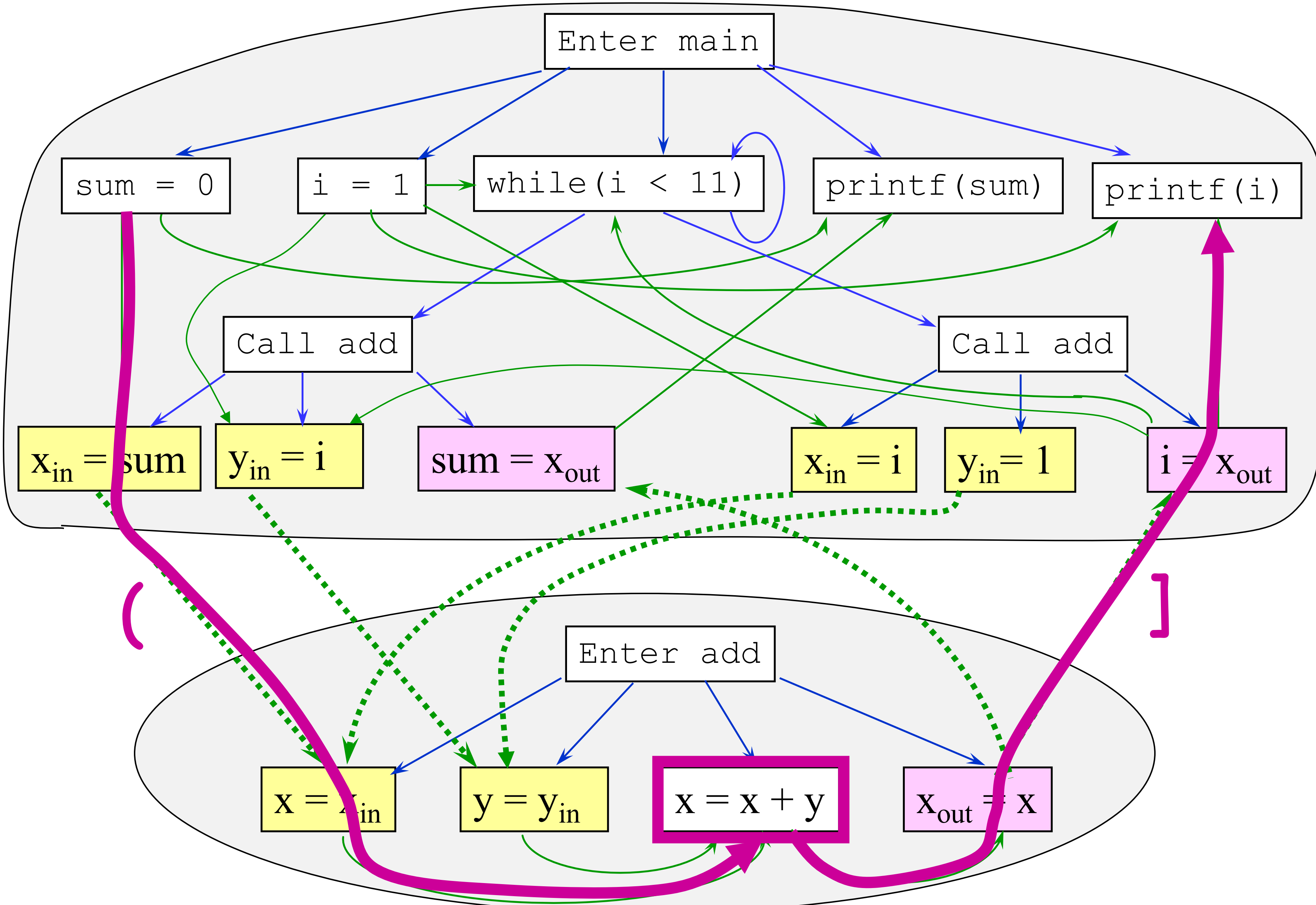
```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```



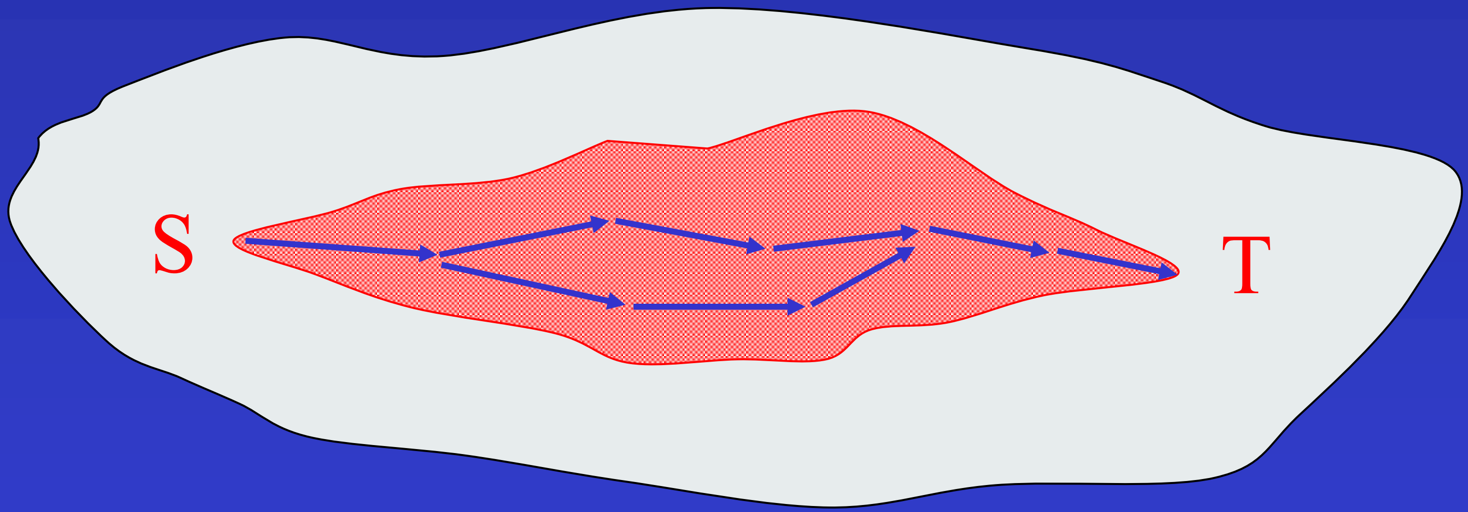
Chop with respect to "sum = 0" and "printf("%d\n",i)"

Non-Transitivity and Slicing



Program Chopping

Given source *S* and target *T*, what program points transmit effects from *S* to *T*?



"Precise interprocedural chopping"
[Reps & Rosay FSE 95]

CF-Recognition vs. CFL-Reachability

- CF-Recognition
  - Chain graphs
  - General grammar: sub-cubic time [Valiant75]
  - **LL(1), LR(1):** linear time
- CFL-Reachability
  - General graphs: $O(N^3)$
  - **LL(1):** $O(N^3)$
  - **LR(1):** $O(N^3)$
  - Certain kinds of graphs: $O(N+E)$
  - Regular languages: $O(N+E)$

Gen/kill IDFA

GMOD IDFA

Regular-Language Reachability

[Yannakakis 90]

- G: Graph (N nodes, E edges)
- L: A regular language
- L-path from s to t iff $s \xrightarrow{\alpha}^{*} t, \alpha \in L$
- Running time: $\boxed{O(N+E)}$ $\boxed{\text{vs. } O(N^3)}$
- Ordinary reachability (= transitive closure)
  - Label each edge with e
  - L is e^*

Themes

- Harnessing CFL-reachability
- Relationship to other analysis paradigms
- Exhaustive alg. $\Rightarrow$ Demand alg.
- Understanding complexity
  - Linear . . . cubic . . . undecidable
- Beyond CFL-reachability

Relationship to Other Analysis Paradigms

- Dataflow analysis
  - –reachability versus equation solving
- Deduction
- Set constraints

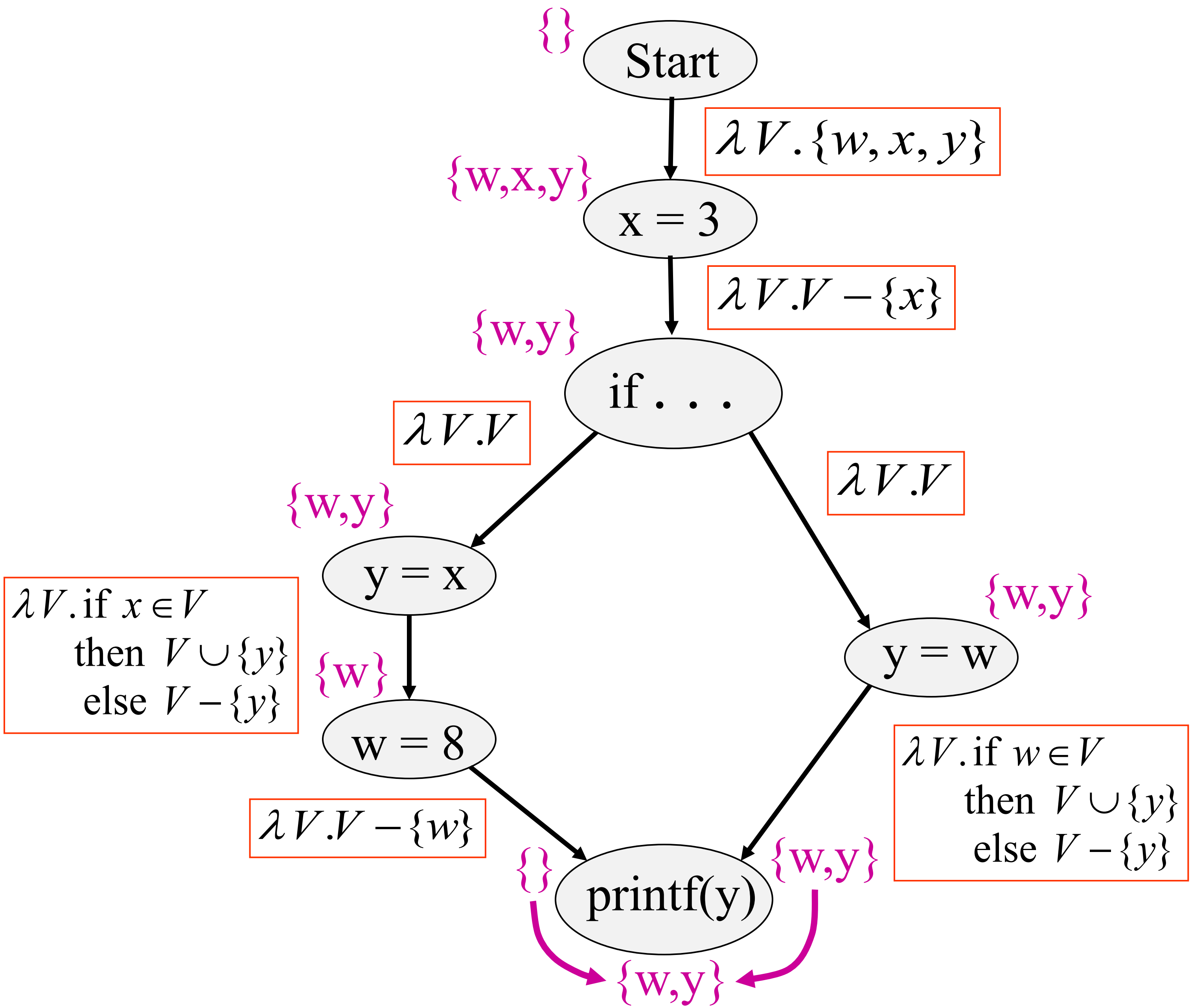
Dataflow Analysis

- Goal: For each point in the program, determine a superset of the “facts” that could possibly hold during execution
- Examples
  - Constant propagation
  - Reaching definitions
  - Live variables
  - Possibly uninitialized variables

Useful For . . .

- Optimizing compilers
- Parallelizing compilers
- Tools that detect possible logical errors
- Tools that show the effects of a proposed modification

Possibly Uninitialized Variables

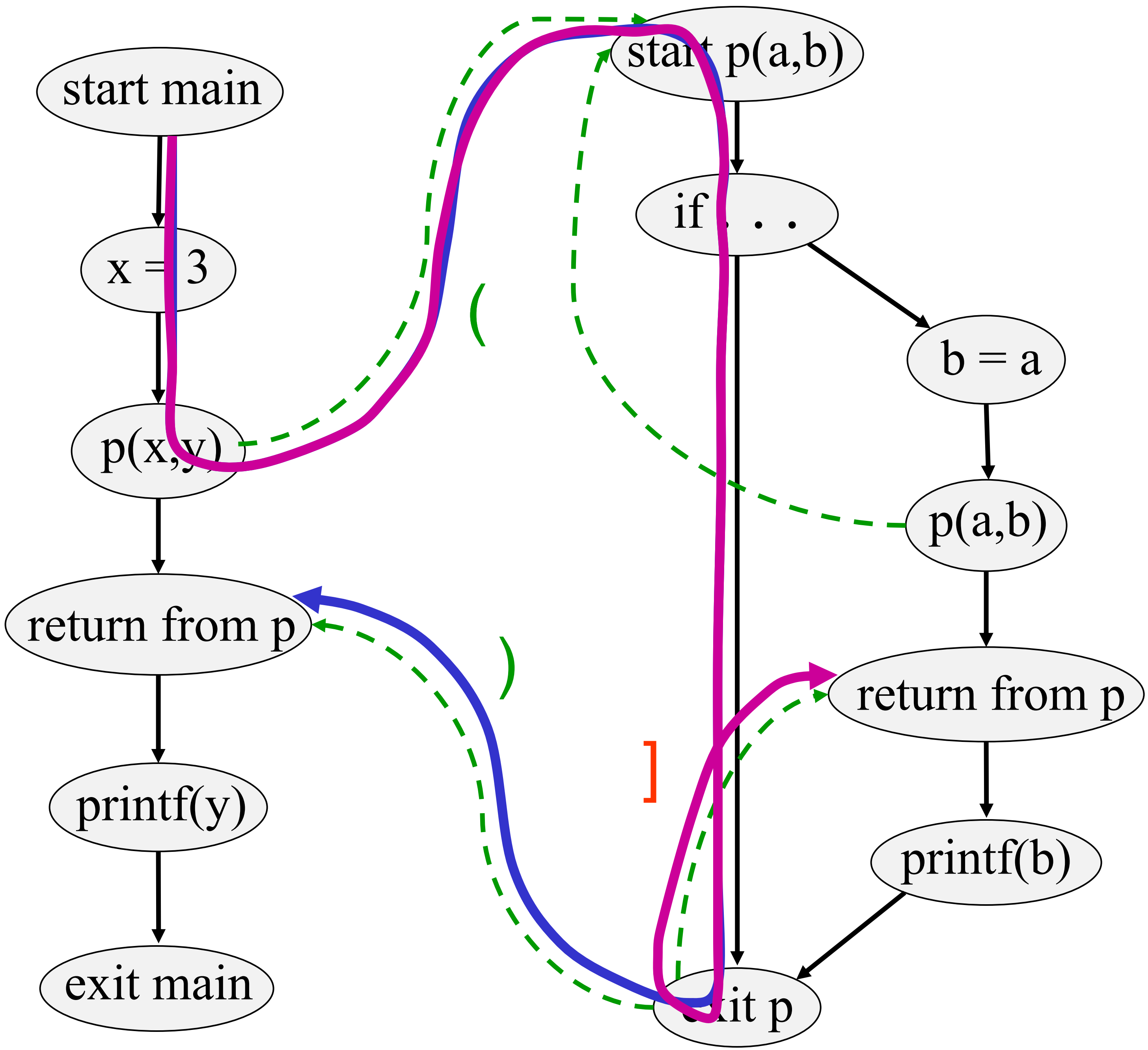


Precise *Intra*procedural Analysis

$$\overset{C}{\underset{start}{\bullet}} \xrightarrow{f_1} \bullet \xrightarrow{f_2} \bullet \cdots \bullet \xrightarrow{f_{k-1}} \bullet \xrightarrow{f_k} \underset{n}{\bullet}$$

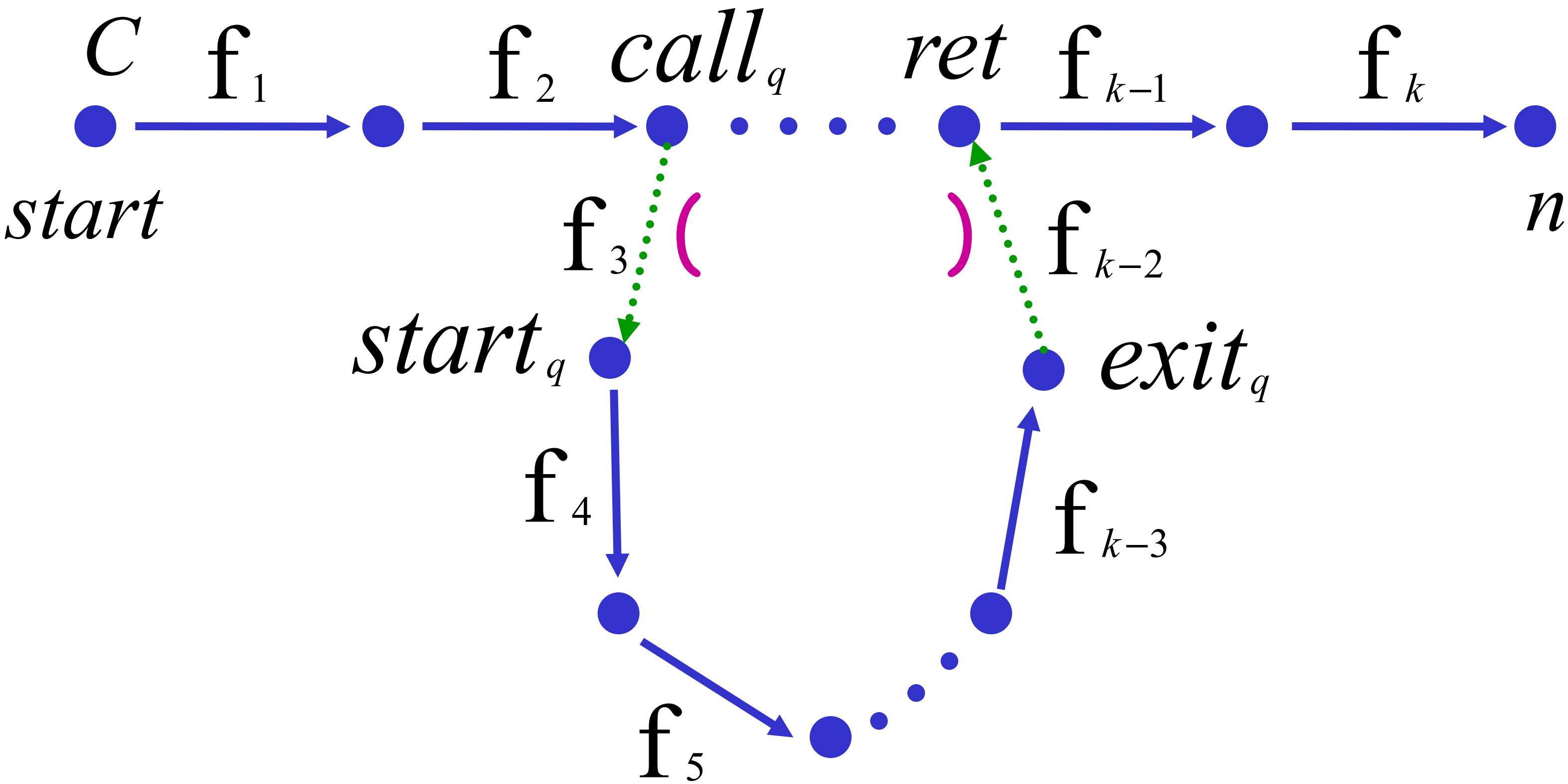
$$\mathrm{pf}_p = f_k \circ f_{k-1} \cdots \circ f_2 \circ f_1$$

$$\mathrm{MOP}[n] = \bigcup_{p \in \underline{\mathrm{PathsTo}}[n]} \mathrm{pf}_p(C)$$



start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
(
)
]

Precise *Inter*procedural Analysis



$$\mathrm{MOMP}[n] = \bigcup_{p \in \underline{\mathrm{MatchedPathsTo}}[n]} \mathrm{pf}_p(C)$$

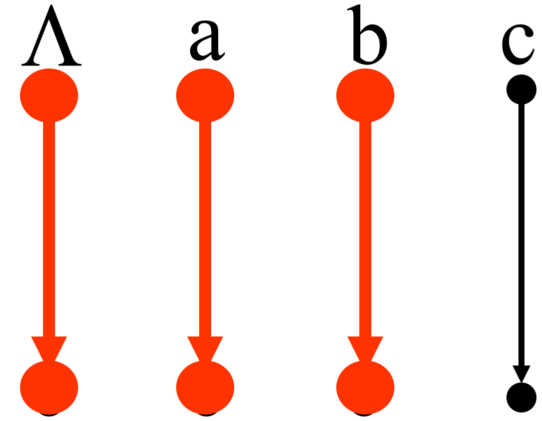
[Sharir & Pnueli 81]

Representing Dataflow Functions

Identity Function

$f = \lambda V.V$

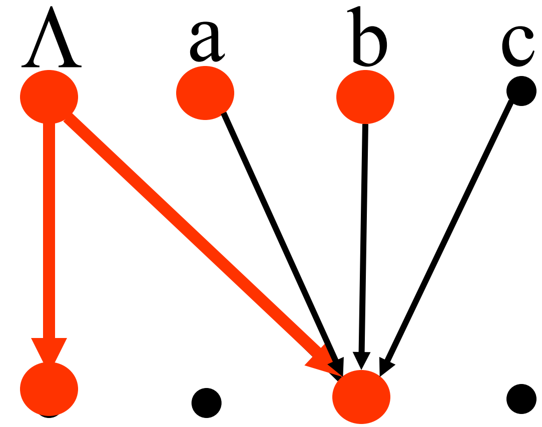
$\boxed{f(\{a,b\}) = \{a,b\}}$



Constant Function

$f = \lambda V.\{b\}$

$\boxed{f(\{a,b\}) = \{b\}}$

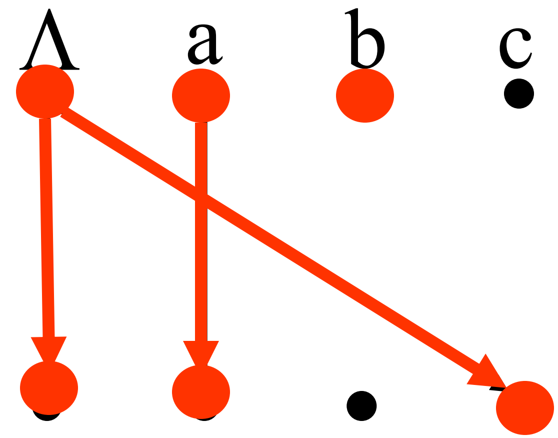


Representing Dataflow Functions

"Gen/Kill" Function

$$f = \lambda V.(V - \{b\}) \cup \{c\}$$

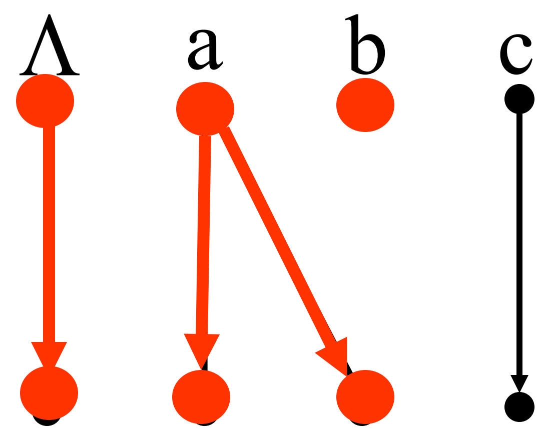
$$\boxed{f(\{a,b\}) = \{a,c\}}$$

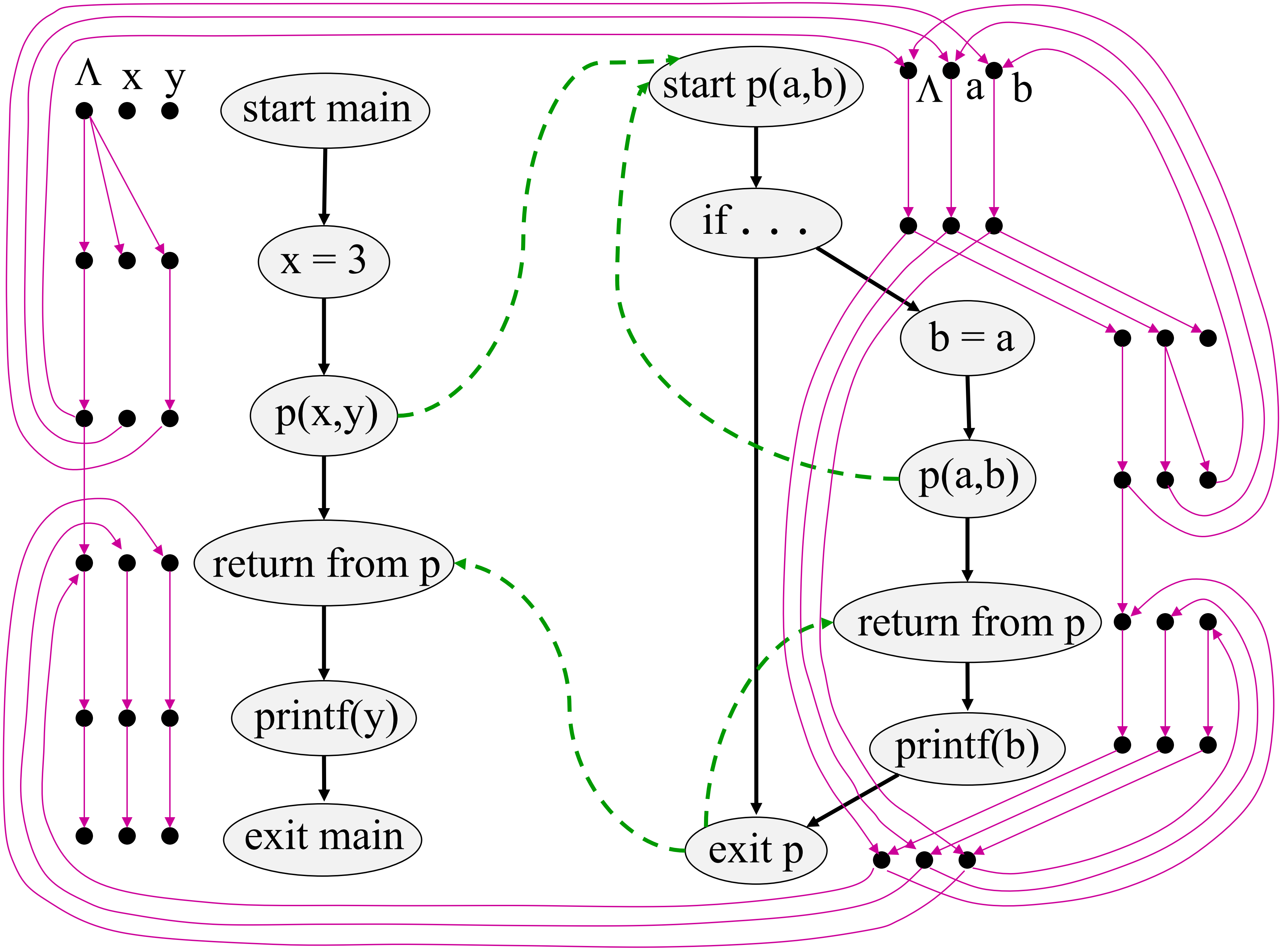


Non-"Gen/Kill" Function

$$\begin{aligned} f = \lambda V.\ & \text{if } a \in V \\ & \text{then } V \cup \{b\} \\ & \text{else } V - \{b\} \end{aligned}$$

$$\boxed{f(\{a,b\}) = \{a,b\}}$$



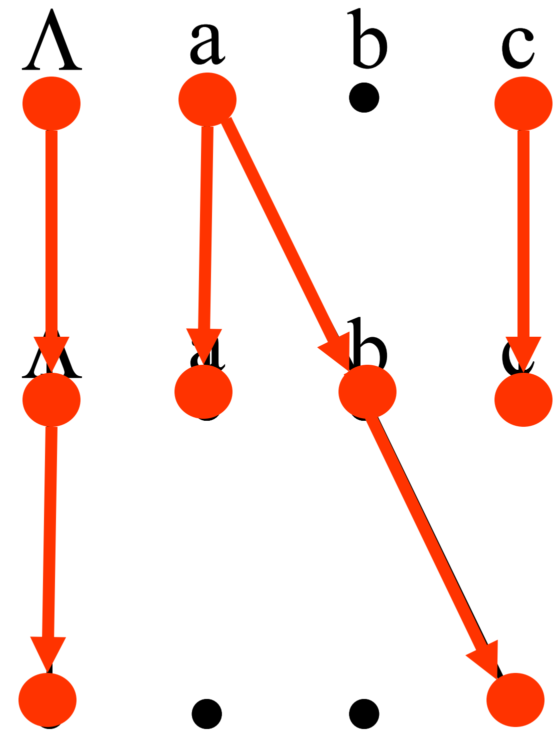


Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
Λ a b
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p

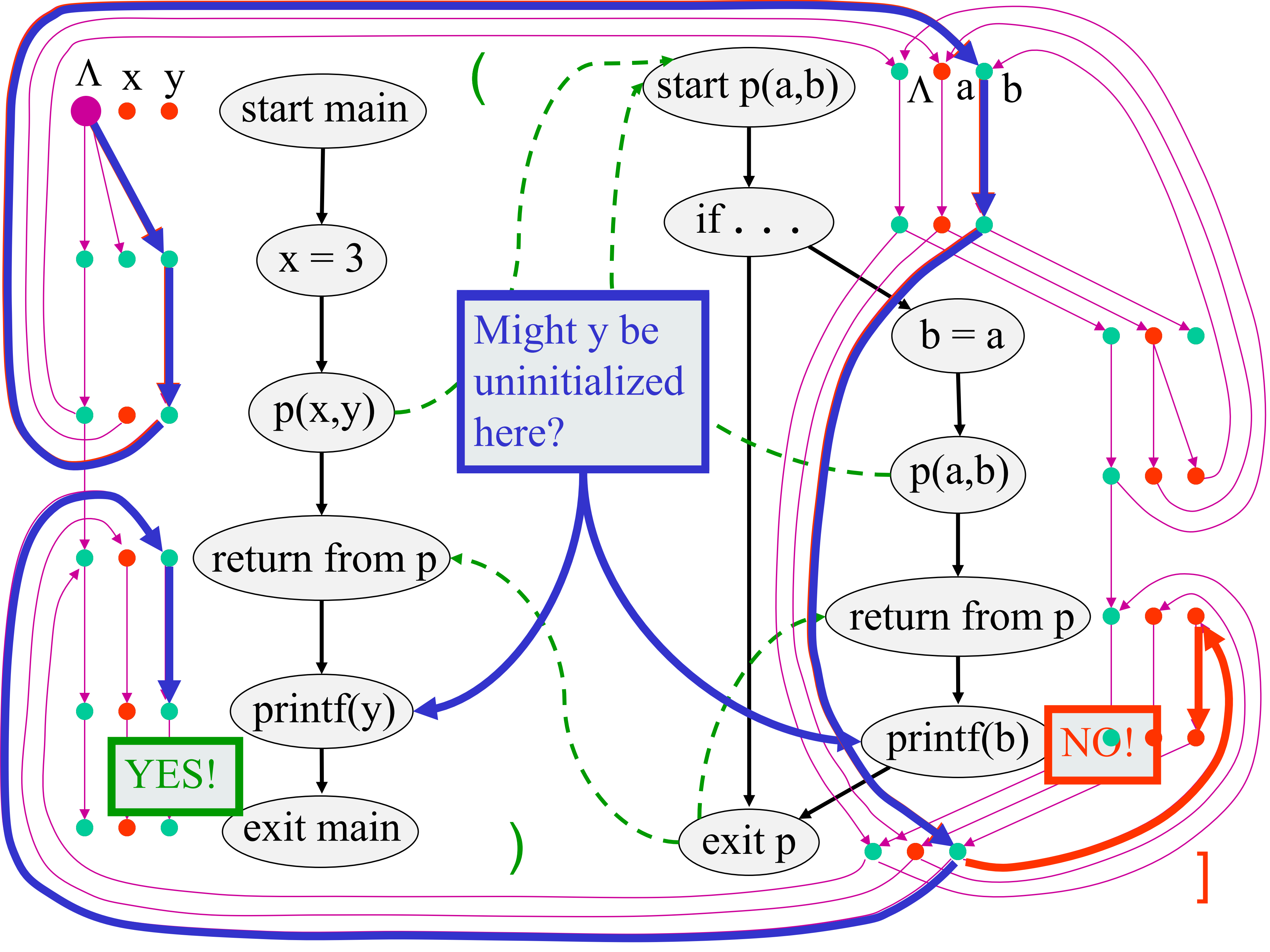
Composing Dataflow Functions

$$f_1 = \lambda V.\text{if } a \in V \text{ then } V \cup \{b\} \text{ else } V - \{b\}$$

$$f_2 = \lambda V.\text{if } b \in V \text{ then } \{c\} \text{ else } \phi$$

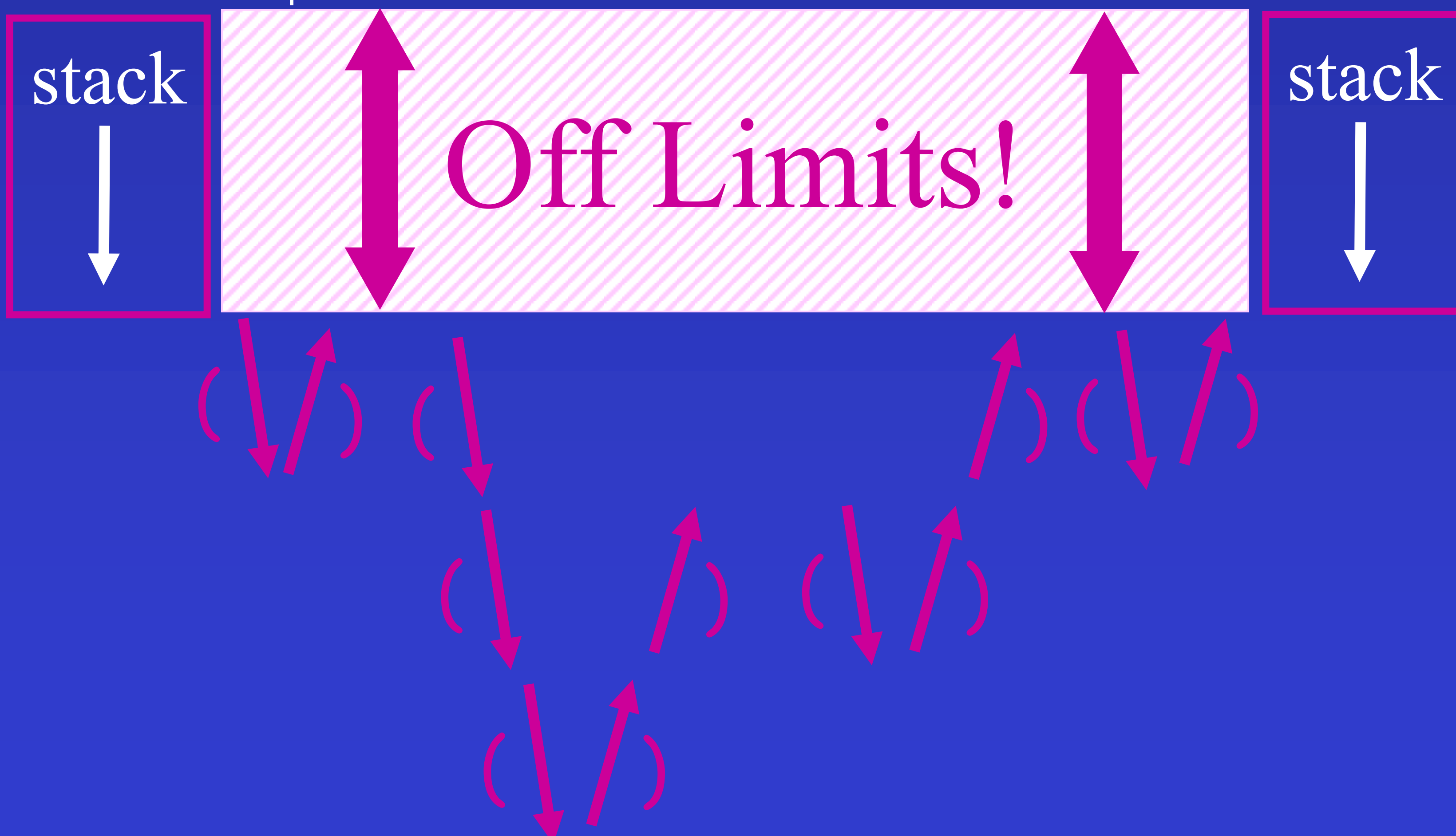


$$f_2 \circ f_1(\{a, c\}) = \boxed{\{c\}}$$

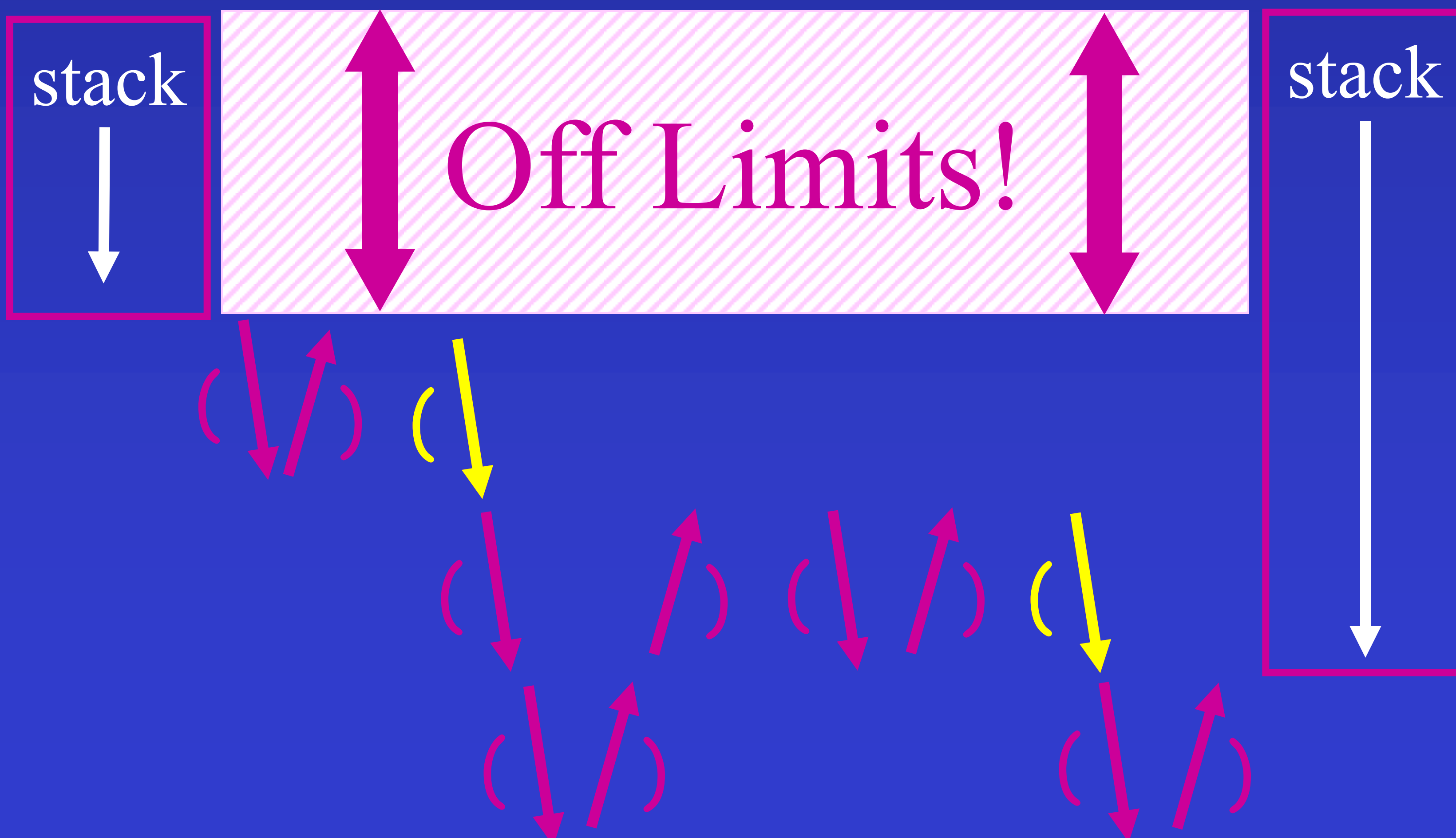


Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
YES!
(
)
Might y be uninitialized here?
start p(a,b)
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
Λ a b
NO!
]

$$
\begin{aligned}
matched \;\rightarrow\;& matched\ matched \\
\mid\;& (_i\ matched\ )_i \qquad 1 \le i \le \text{CallSites} \\
\mid\;& \text{edge} \\
\mid\;& \varepsilon
\end{aligned}
$$



$$
\begin{aligned}
\textit{unbalLeft} \;&\rightarrow\; \textit{matched}\ \textit{unbalLeft} \\
&\;\;|\;\; (_i\ \textit{unbalLeft} \qquad 1 \le i \le \text{CallSites} \\
&\;\;|\;\; \varepsilon
\end{aligned}
$$



Interprocedural Dataflow Analysis via CFL-Reachability

- Graph: Exploded control-flow graph
- *L*: *L*(*unbalLeft*)
- Fact *d* holds at *n* iff there is an *L*(*unbalLeft*)-path from $\langle start_{main}, \Lambda \rangle$ to $\langle n, d \rangle$

Asymptotic Running Time

[Reps, Horwitz, & Sagiv 95]

- CFL-reachability
  - Exploded control-flow graph: ND nodes
  - Running time: $O(N^3D^3)$
- Exploded control-flow graph $\rightarrow$ Special structure

Running time: $O(ED^3)$

Typically: E | N, hence $O(ED^3)$ | $O(ND^3)$

"Gen/kill" problems: $O(ED)$

Why Bother?

"We're only interested in million-line programs"

- Know thy enemy!
  - "Any" algorithm must do these operations
  - Avoid pitfalls (e.g., claiming $\boldsymbol{O(N^2)}$ algorithm)
- The essence of "context sensitivity"
- Special cases
  - "Gen/kill" problems: $\boldsymbol{O(ED)}$
- Compression techniques
  - Basic blocks
  - SSA form, sparse evaluation graphs
- Demand algorithms

Relationship to Other Analysis Paradigms

- Dataflow analysis
  - reachability versus equation solving
- Deduction
- Set constraints

The Need for Pointer Analysis

```
int main() {
  int sum = 0;
  int i = 1;
  int *p = &sum;
  int *q = &i;
  int (*f)(int,int) = add;
  while (*q < 11) {
    *p = (*f)(*p,*q);
    *q = (*f)(*q,1);
  }
  printf("%d\n",*p);
  printf("%d\n",*q);
}
```

```
int add(int x, int y)
{
  return x + y;
}
```

The Need for Pointer Analysis

```
int main() {
  int sum = 0;
  int i = 1;
  int *p = &sum;
  int *q = &i;
  int (*f)(int,int) = add;
  while (*q < 11) {
    *p = (*f)(*p,*q);
    *q = (*f)(*q,1);
  }
  printf("%d\n",*p);
  printf("%d\n",*q);
}
```

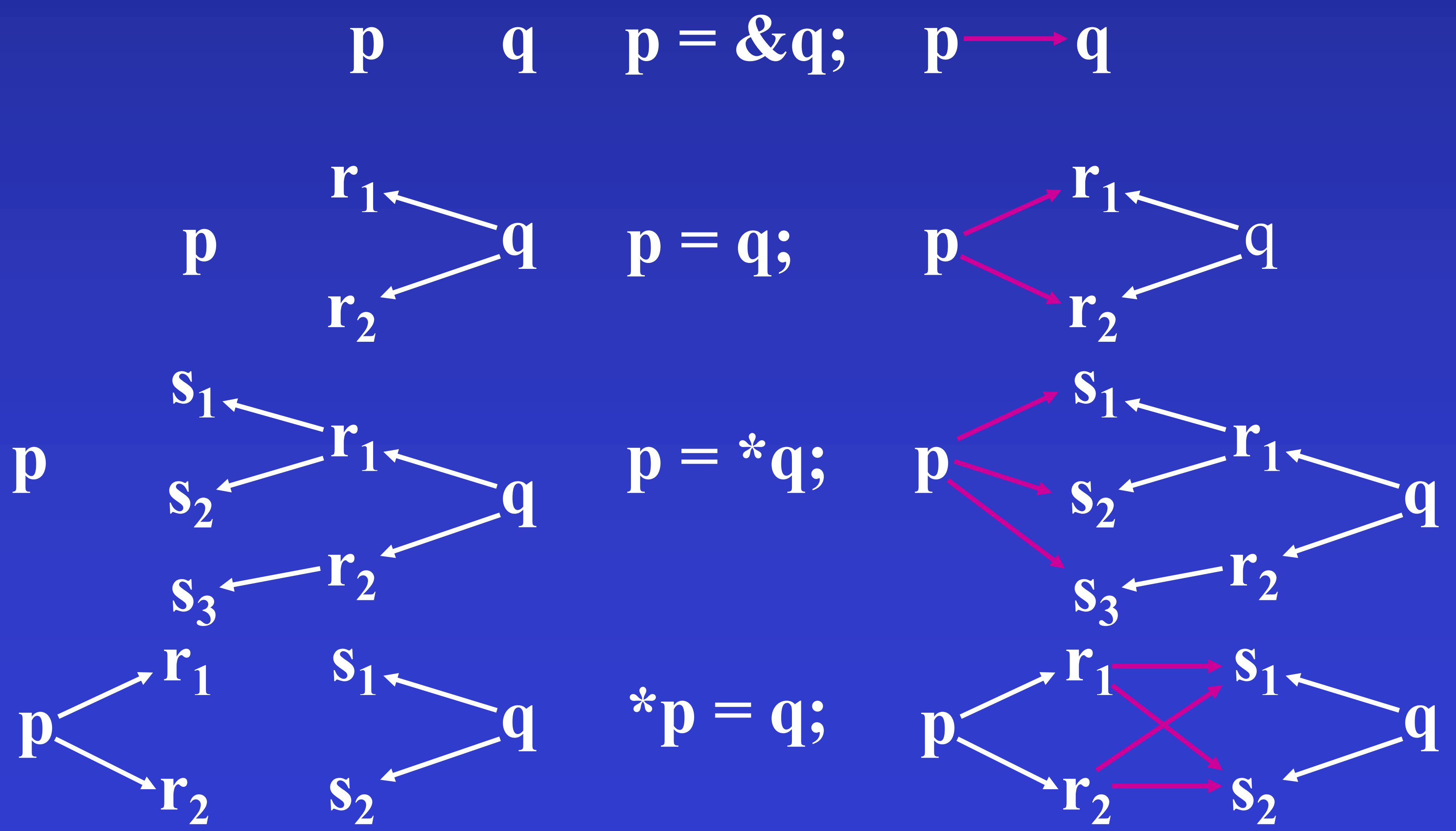
```
int add(int x, int y)
{
  return x + y;
}
```

The Need for Pointer Analysis

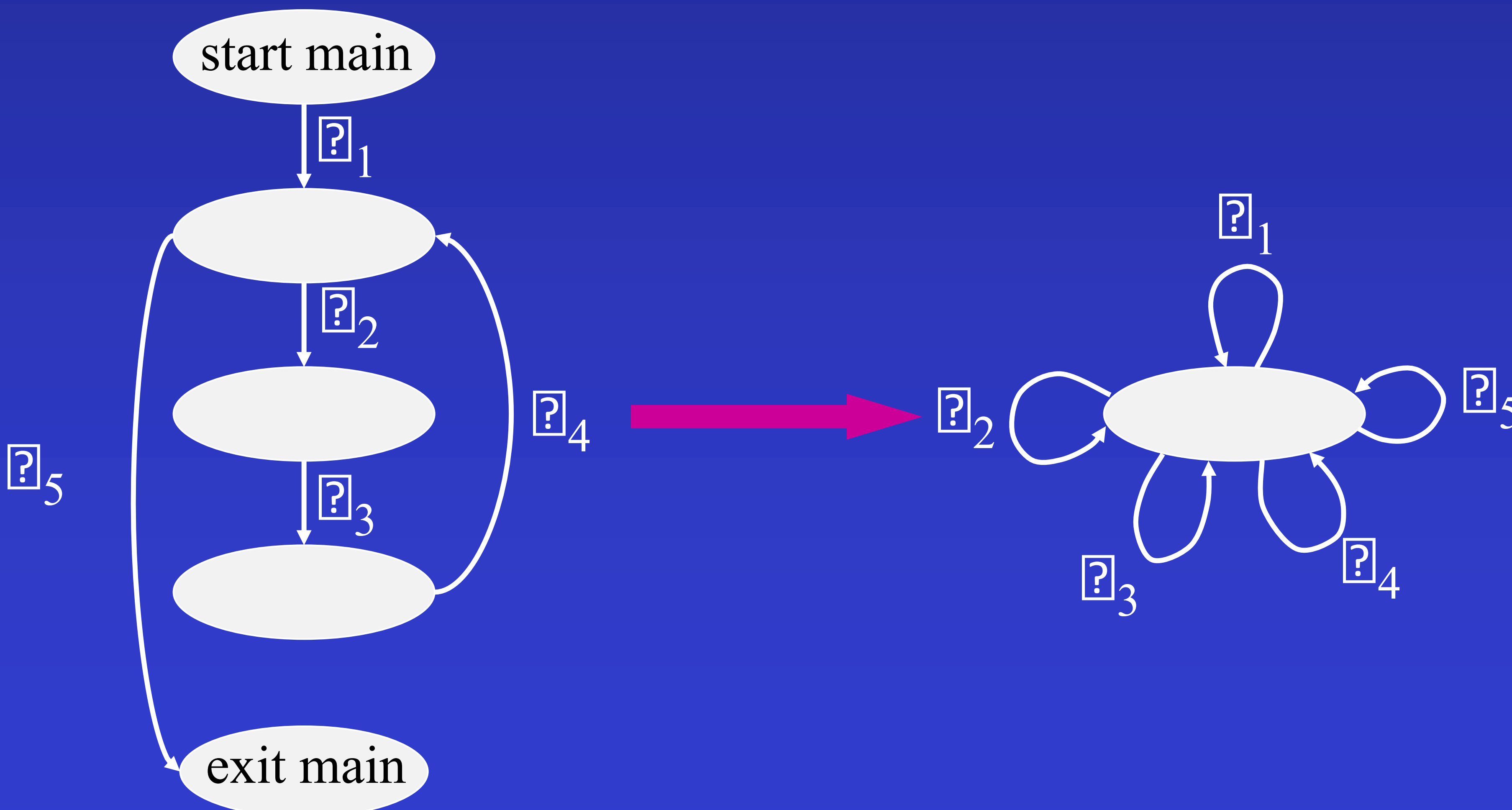
```
int main() {
  int sum = 0;
  int i = 1;
  int *p = &sum;
  int *q = &i;
  int (*f)(int,int) = add;
  while (i < 11) {
    sum = add(sum,i);
    i = add(i,1);
  }
  printf("%d\n",sum);
  printf("%d\n",i);
}
```

```
int add(int x, int y)
{
  return x + y;
}
```

Flow-**Sensitive** Points-To Analysis



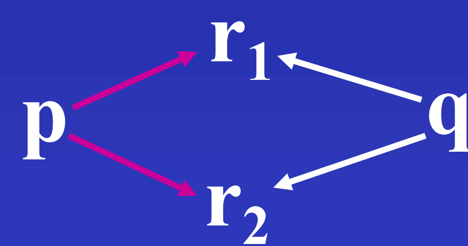
Flow-Sensitive → Flow-Insensitive



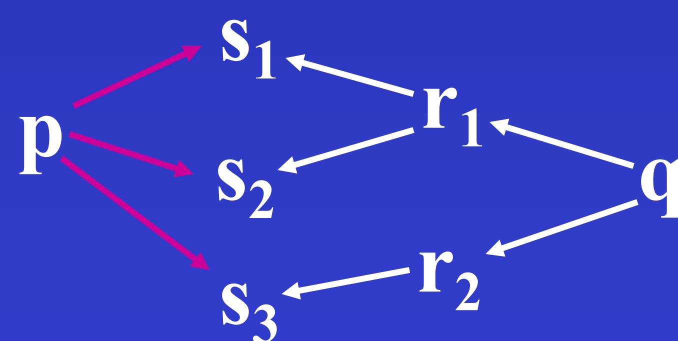
Flow-**Insensitive** Points-To Analysis
[Andersen 94, Shapiro & Horwitz 97]

p = &q;

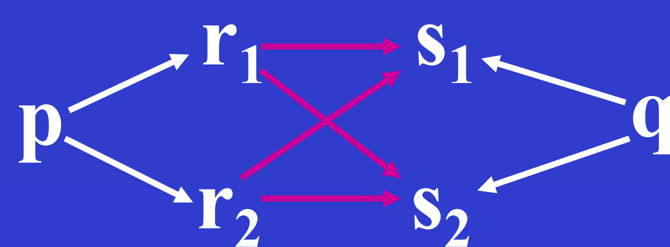
p = q;



p = *q;

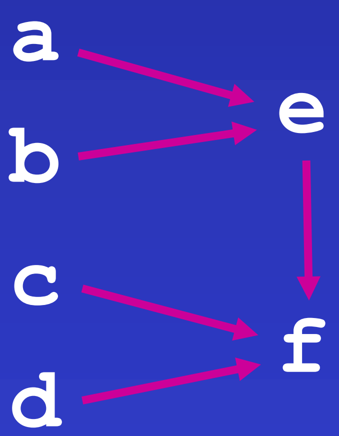


***p = q;**



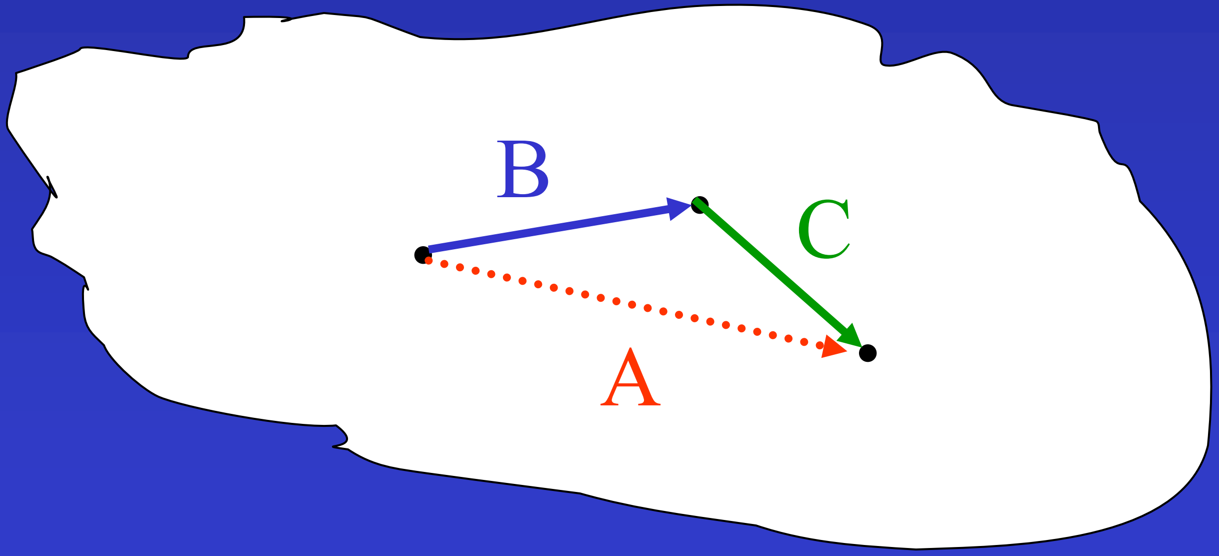
Flow-Insensitive Points-To Analysis

```
 a = &e;
 b =  a;
 c = &f;
*b =  c;
 d = *a;
```

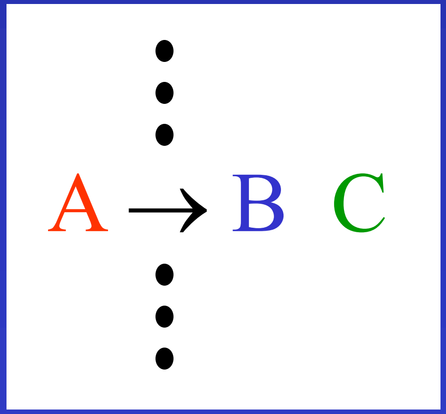


CFL-Reachability via Dynamic Programming

Graph

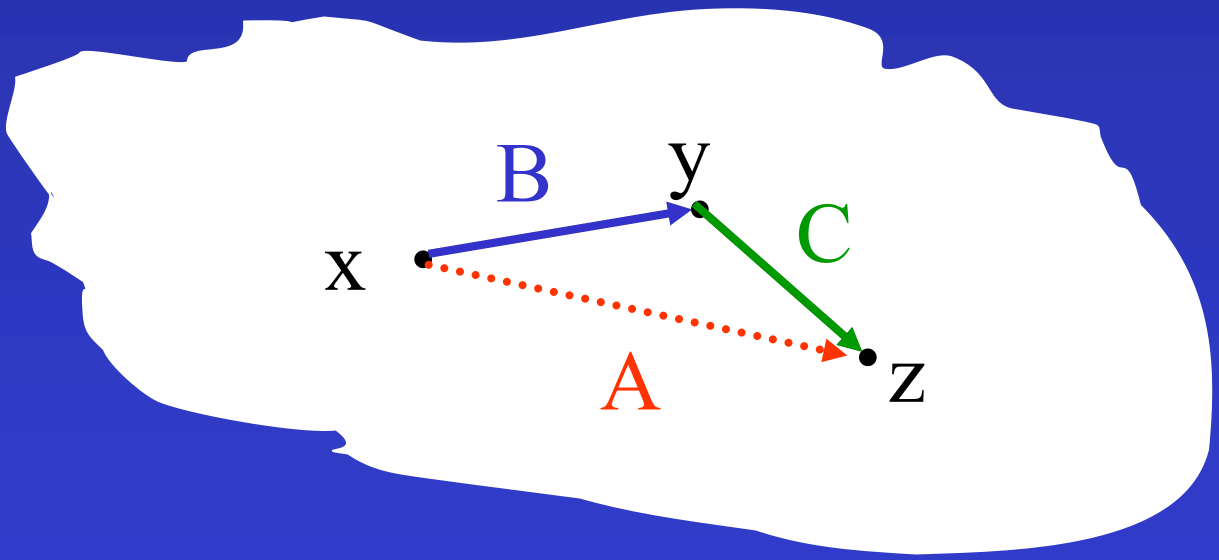


Grammar



CFL-Reachability = Chain Programs

Graph



Grammar

$$\vdots$$
$$A \rightarrow B\ C$$
$$\vdots$$

a(X,Z) :- b(X,Y), c(Y,Z).

Base Facts for Points-To Analysis

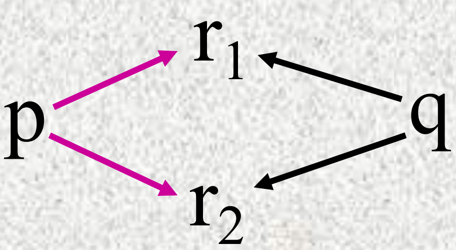
p = &q;	assignAddr(p,q).
p = q;	assign(p,q).
p = *q;	assignStar(p,q).
*p = q;	starAssign(p,q).

Rules for Points-To Analysis (I)

p = &q; $p \rightarrow q$

pointsTo(P,Q) :- assignAddr(P,Q).

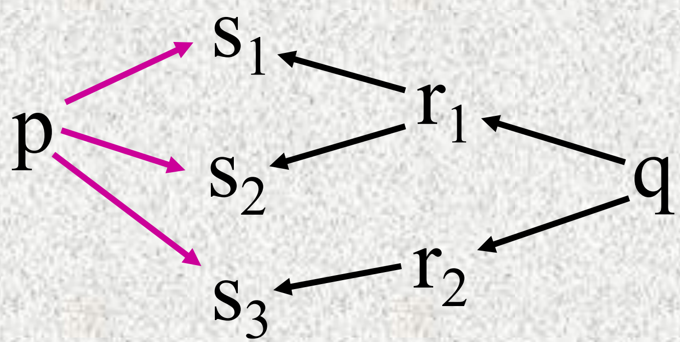
p = q;



pointsTo(P,R) :- assign(P,Q), pointsTo(Q,R).

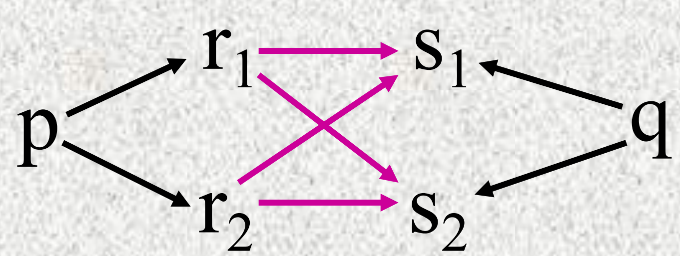
Rules for Points-To Analysis (II)

p = *q;



pointsTo(P,S) :- assignStar(P,Q),pointsTo(Q,R),pointsTo(R,S).

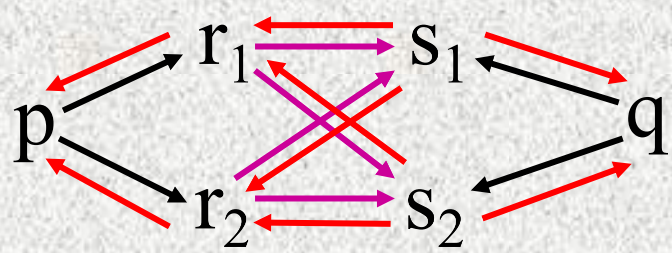
*p = q;



pointsTo(R,S) :- starAssign(P,Q),pointsTo(P,R),pointsTo(Q,S).

Creating a Chain Program

*p = q;



pointsTo(R,S) :- starAssign(P,Q),pointsTo(P,R),pointsTo(Q,S).

pointsTo(R,S) :- pointsTo(P,R),starAssign(P,Q),pointsTo(Q,S).

pointsTo(R,S) :- pointsTo(R,P),starAssign(P,Q),pointsTo(Q,S).

~~pointsTo(R,P) :- pointsTo(P,R).~~

Base Facts for Points-To Analysis

p = &q;	assignAddr(p,q). ~~assignAddr~~(q,p).
p = q;	assign(p,q). ~~assign~~(q,p).
p = *q;	assignStar(p,q). ~~assignStar~~(q,p).
*p = q;	starAssign(p,q). ~~starAssign~~(q,p).

Creating a Chain Program

pointsTo(P,Q) :- assignAddr(P,Q).

pointsTo(Q,P) :- assignAddr(Q,P).

pointsTo(P,R) :- assign(P,Q), pointsTo(Q,R).

pointsTo(R,P) :- pointsTo(R,Q), assign(Q,P).

pointsTo(P,S) :- assignStar(P,Q),pointsTo(Q,R),pointsTo(R,S).

pointsTo(S,P) :- pointsTo(S,R),pointsTo(R,Q),assignStar(Q,P).

pointsTo(R,S) :- pointsTo(R,P),starAssign(P,Q),pointsTo(Q,S).

pointsTo(S,R) :- pointsTo(S,Q),starAssign(Q,P),pointsTo(P,R).

. . . and now to CFL-Reachability

$$\text{pointsTo} \to \text{assignAddr}$$

$$\overline{\text{pointsTo}} \to \overline{\text{assignAddr}}$$

$$\text{pointsTo} \to \text{assign} \;\; \text{pointsTo}$$

$$\overline{\text{pointsTo}} \to \overline{\text{pointsTo}} \;\; \overline{\text{assign}}$$

$$\text{pointsTo} \to \text{assignStar} \;\; \text{pointsTo} \;\; \text{pointsTo}$$

$$\overline{\text{pointsTo}} \to \overline{\text{pointsTo}} \;\; \overline{\text{pointsTo}} \;\; \overline{\text{assignStar}}$$

$$\text{pointsTo} \to \overline{\text{pointsTo}} \;\; \text{starAssign} \;\; \text{pointsTo}$$

$$\overline{\text{pointsTo}} \to \overline{\text{pointsTo}} \;\; \overline{\text{starAssign}} \;\; \text{pointsTo}$$

Themes

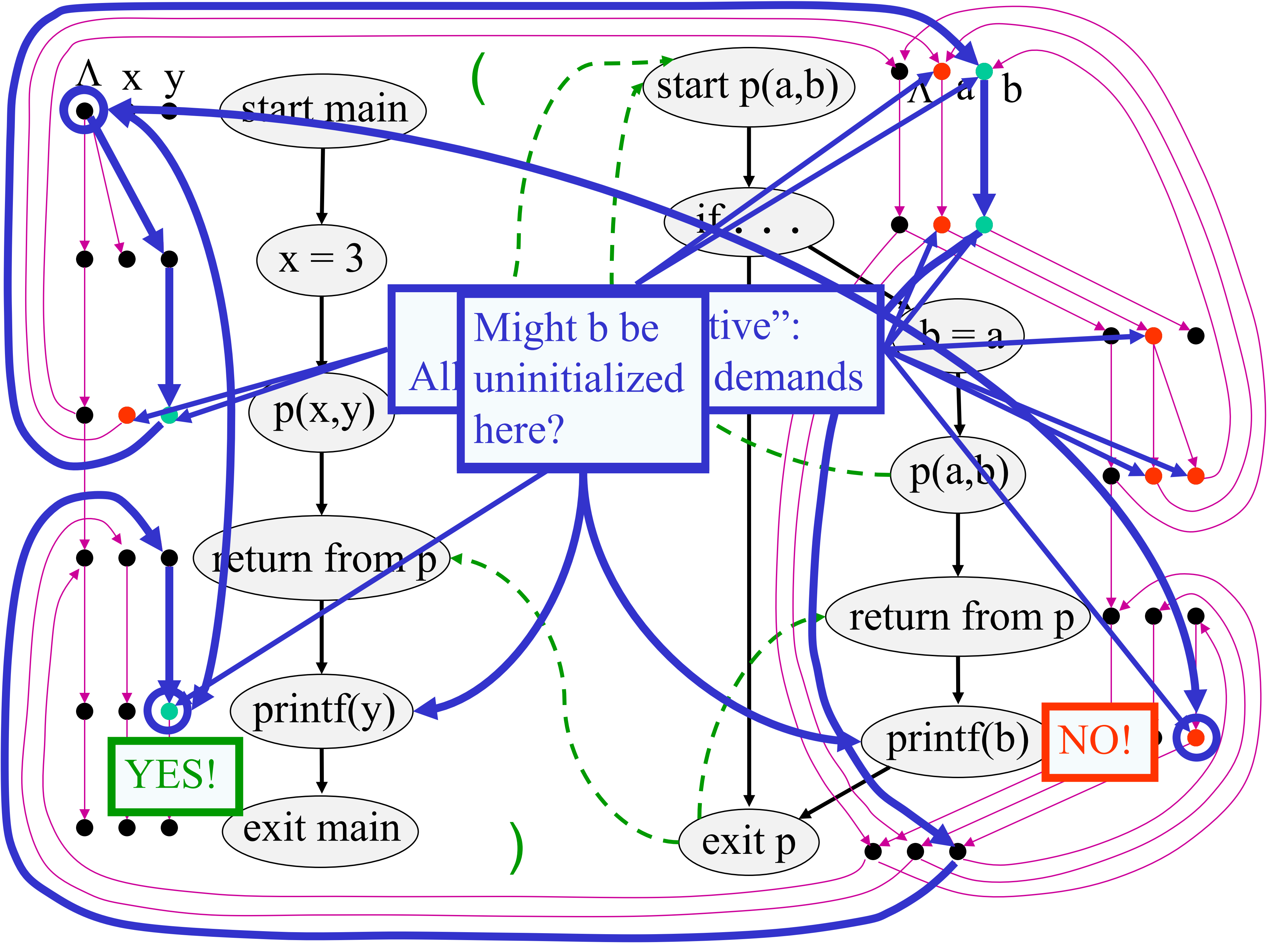
- Harnessing CFL-reachability
- Relationship to other analysis paradigms
- Exhaustive alg. $\Rightarrow$ Demand alg.
- Understanding complexity
  - Linear . . . cubic . . . undecidable
- Beyond CFL-reachability

Exhaustive Versus Demand Analysis

- Exhaustive analysis: **All** facts at **all** points
- Optimization: Concentrate on inner loops
- Program-understanding tools: Only some facts are of interest

Exhaustive Versus Demand Analysis

- Demand analysis:
  - Does a **given** fact hold at a **given** point?
  - **Which** facts hold at a **given** point?
  - At **which** points does a **given** fact hold?
- Demand analysis via CFL-reachability
  - single-source/single-target CFL-reachability
  - single-source/multi-target CFL-reachability
  - multi-source/single-target CFL-reachability



Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
(
)
start p(a,b)
Λ a b
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
All ... tive": ... demands
Might b be uninitialized here?
YES!
NO!

Experimental Results

[Horwitz , Reps, & Sagiv 1995]

- 53 C programs (200-6,700 lines)
- For a single fact of interest:
  - demand **always better** than exhaustive
- All "appropriate" demands beats exhaustive when percentage of "yes" answers is high
  - Live variables
  - Truly live variables
  - Constant predicates
  - . . .

Demand Analysis and LP Queries (I)

- Flow-insensitive points-to analysis
  - Does variable *p* point to *q*?
    - Issue query: ?- pointsTo(p, q).
    - Solve single-source/single-target *L*(pointsTo)-reachability problem
  - What does variable *p* point to?
    - Issue query: ?- pointsTo(p, Q).
    - Solve single-source *L*(pointsTo)-reachability problem
  - What variables point to *q*?
    - Issue query: ?- pointsTo(P, q).
    - Solve single-target *L*(pointsTo)-reachability problem

Demand Analysis and LP Queries (II)

- Flow-sensitive analysis
  - Does a given fact *f* hold at a given point *p*?

    ?- dfFact(p, f).
  - Which facts hold at a given point *p*?

    ?- dfFact(p, F).
  - At which points does a given fact *f* hold?

    ?- dfFact(P, f).
- E.g., flow-sensitive points-to analysis

  ?- dfFact(p, pointsTo(x, Y)).

  ?- dfFact(P, pointsTo(x, y)).

  etc.