Static-Analysis Technology for Security

Thomas Reps

University of Wisconsin

Goals of the Briefing

- Relevance of static analysis to security issues
- Tutorial on some static-analysis issues
- Some forward pointers to other presentations

The Role of Static Analysis

- De-obfuscation to detect . . .
  - Undesirable information flows
  - Buffer-overrun attacks
  - Actions of a virus
- More precise = less freedom for attacker

Dependence Analysis and Analysis of Malicious Code

- Post-mortem analysis
  - What affects what?
- Information Flow
  - Data dependences + control dependences
- Program Differencing
  - Compare slices to identify changes
- CodeSurfer
  - UW (1986-96): NSF, DARPA, ONR, Packard Foundation
  - GT (1997-): DARPA, AF, ONR, NSF, NASA

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”

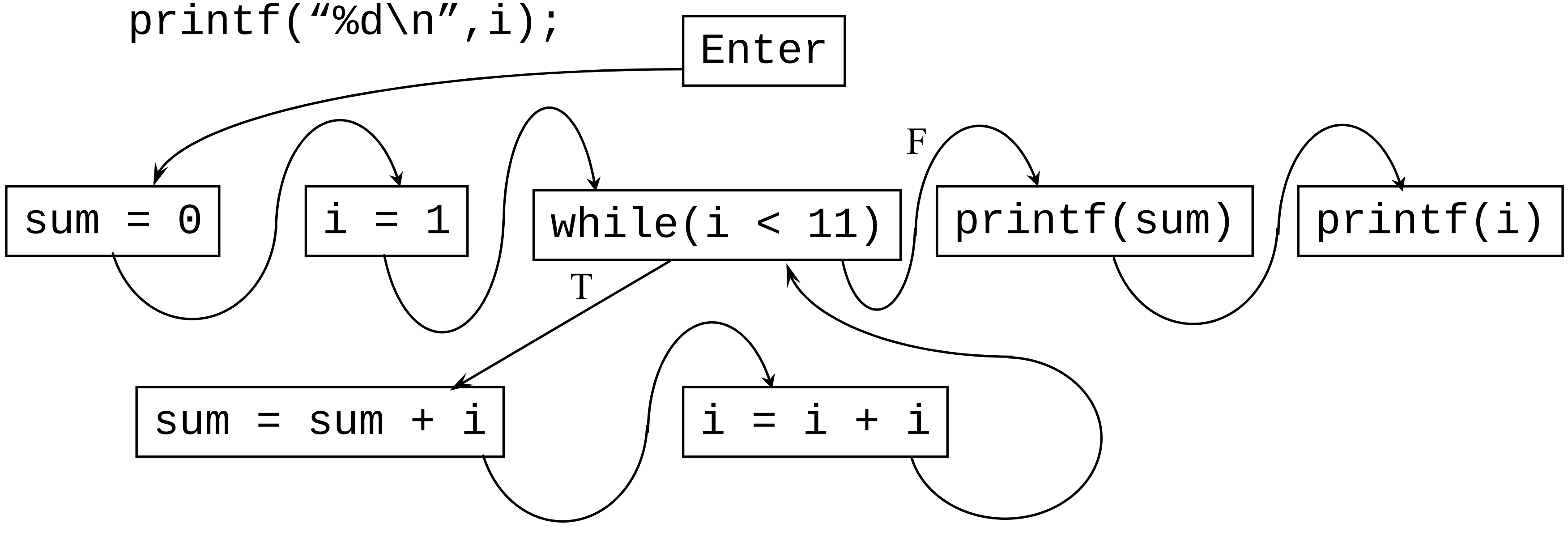
Forward Slice

```
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"

Control Flow Graph

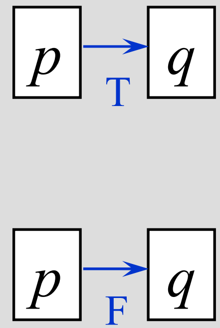
```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf(“%d\n”,sum);
      printf(“%d\n”,i);
}
```



Control Dependence Graph

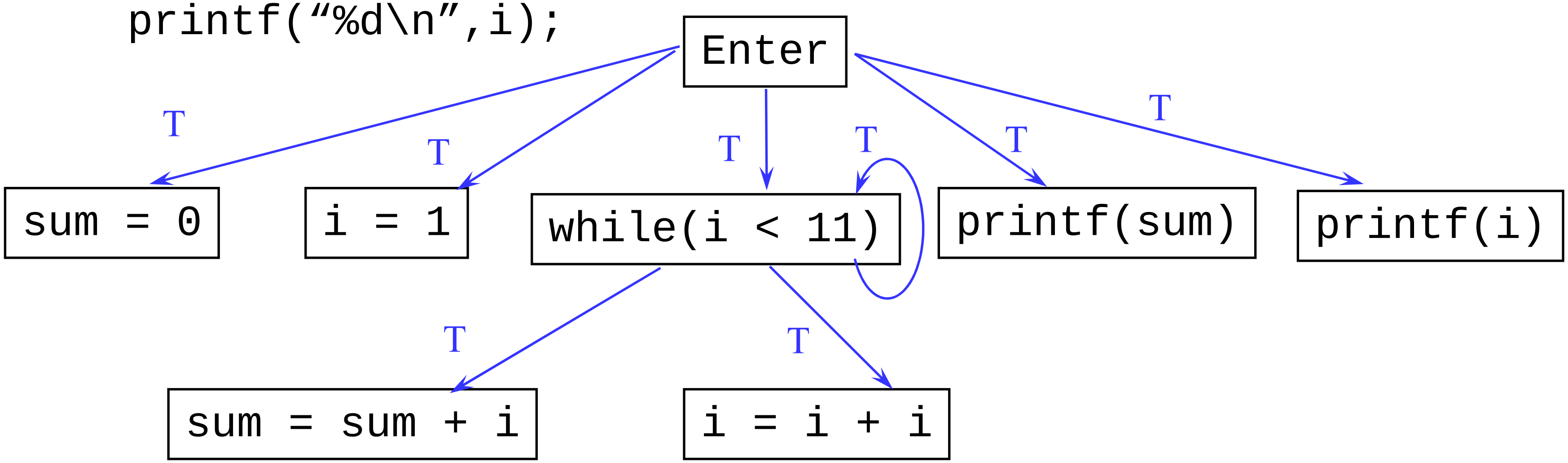
```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf(“%d\n”,sum);
      printf(“%d\n”,i);
}
```

Control dependence



q is reached from *p* if condition *p* is true (T), not otherwise.

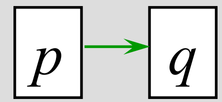
Similar for false (F).



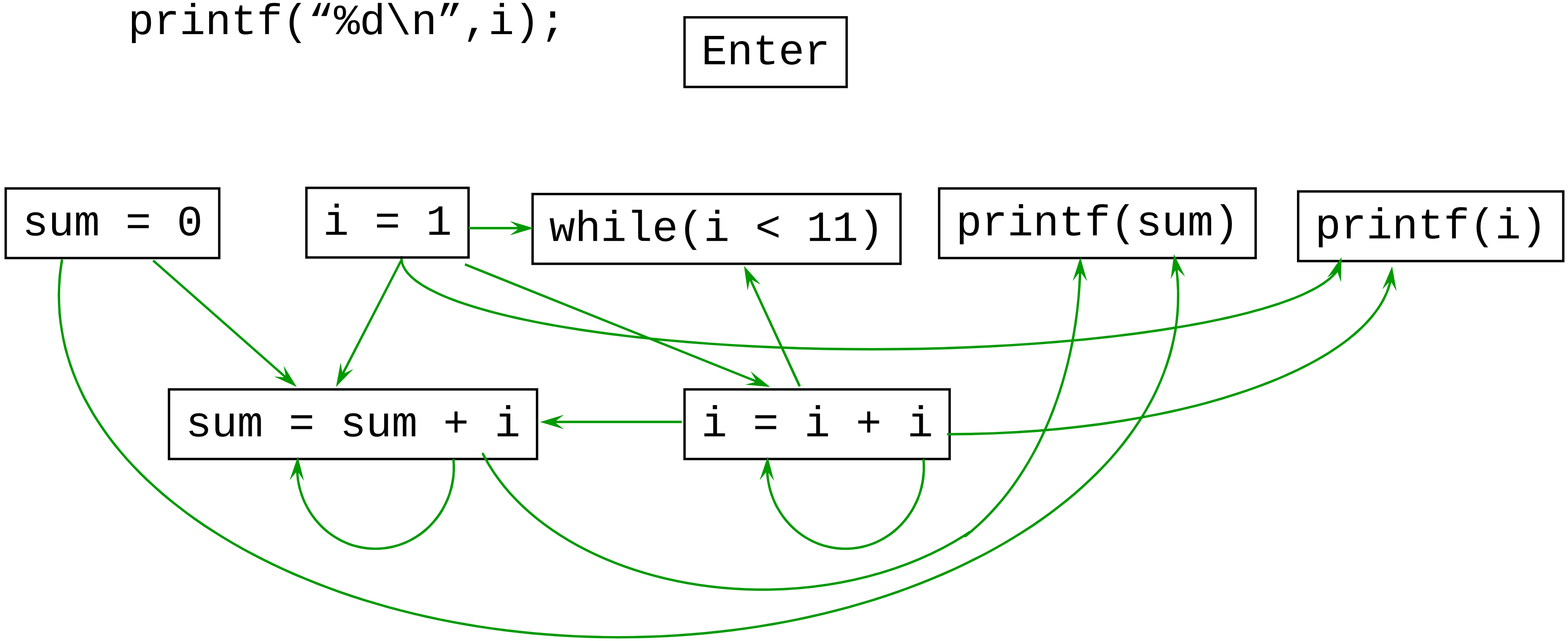
Flow Dependence Graph

```
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.

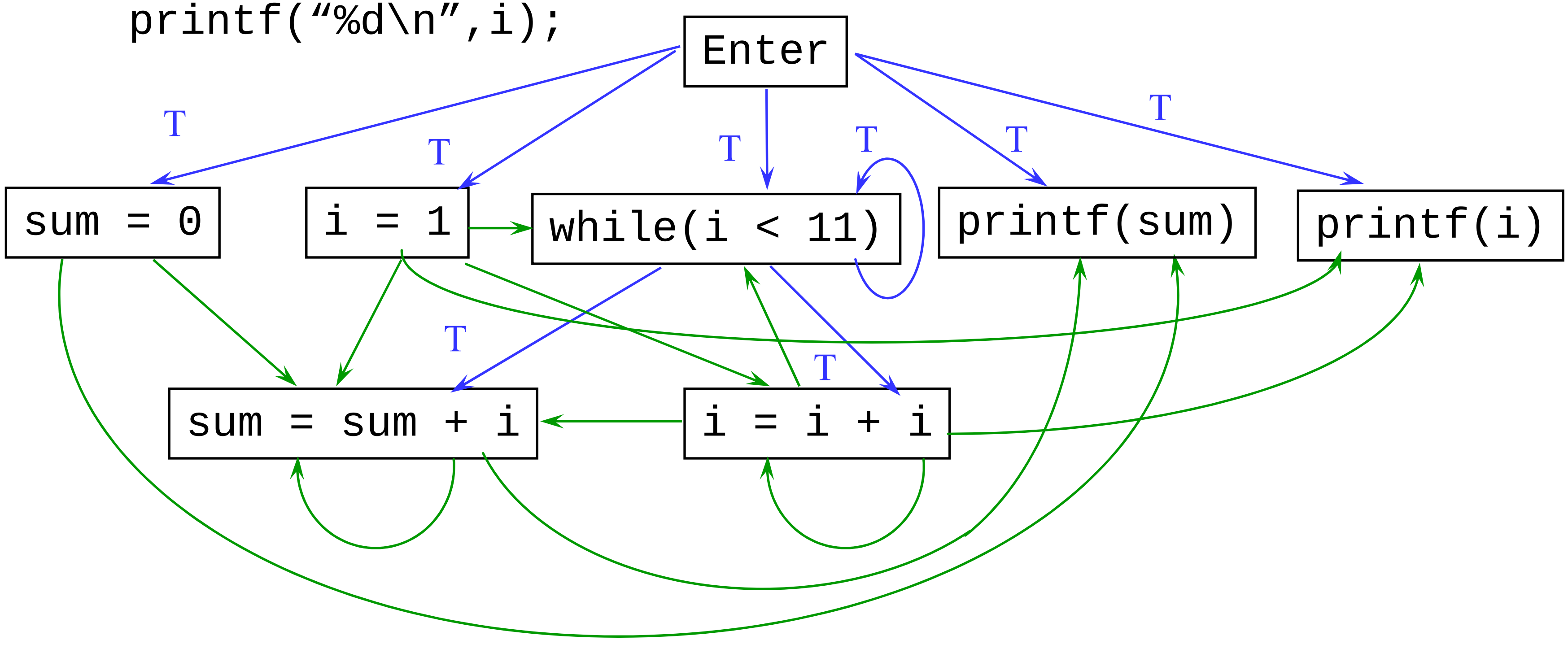


Program Dependence Graph

```
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

Flow dependence

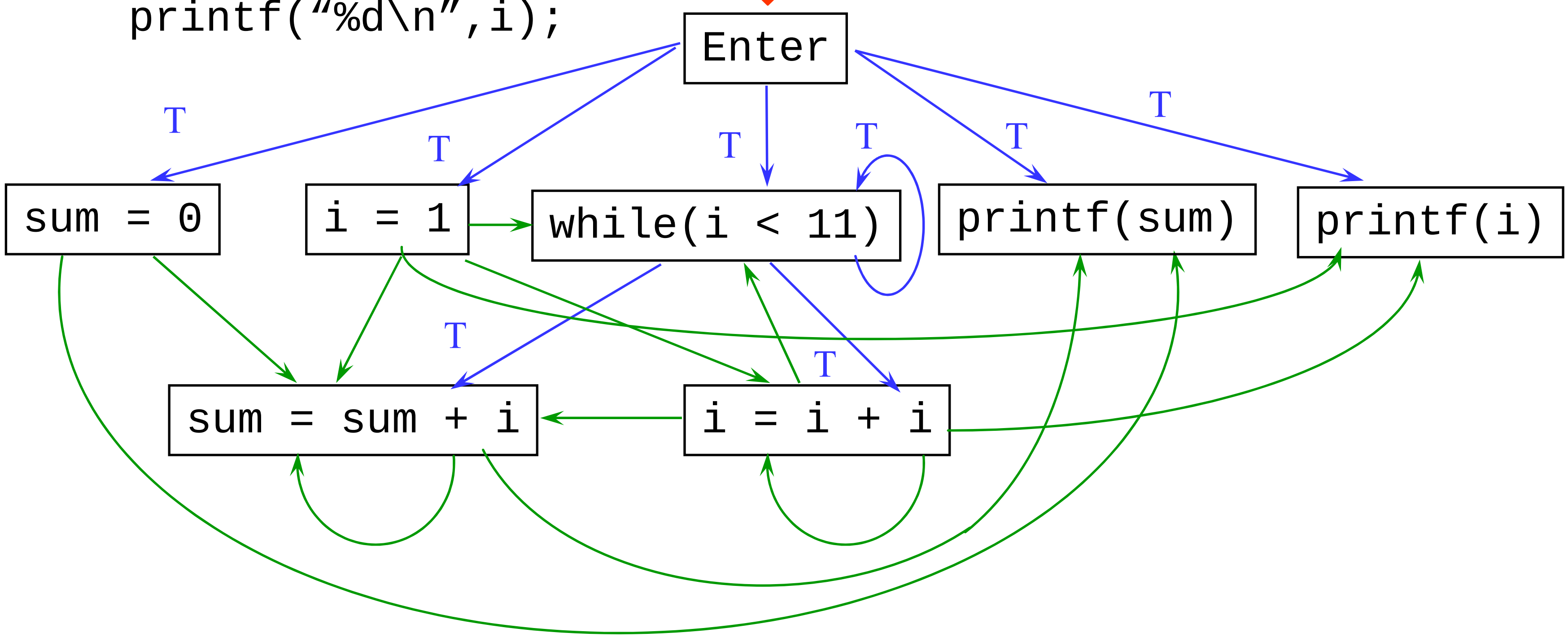


Program Dependence Graph

```
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);
}
```

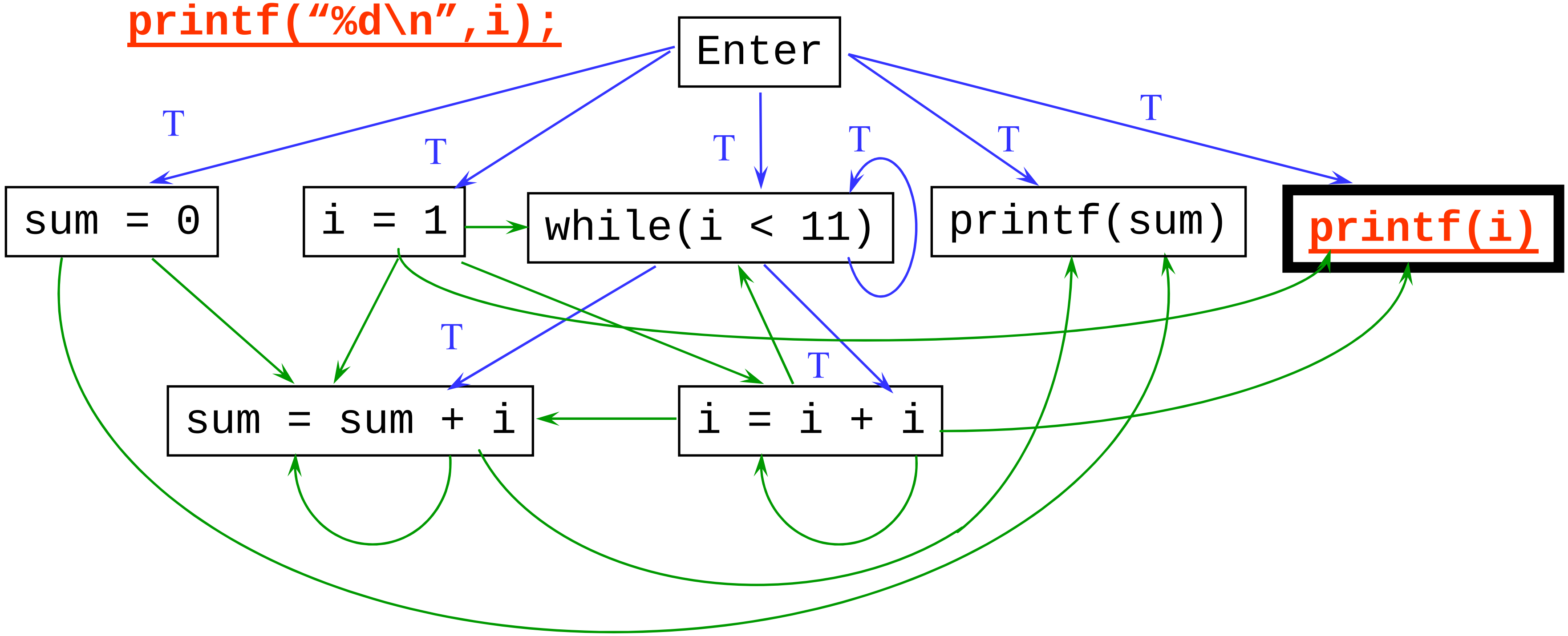
Opposite Order

Same PDG



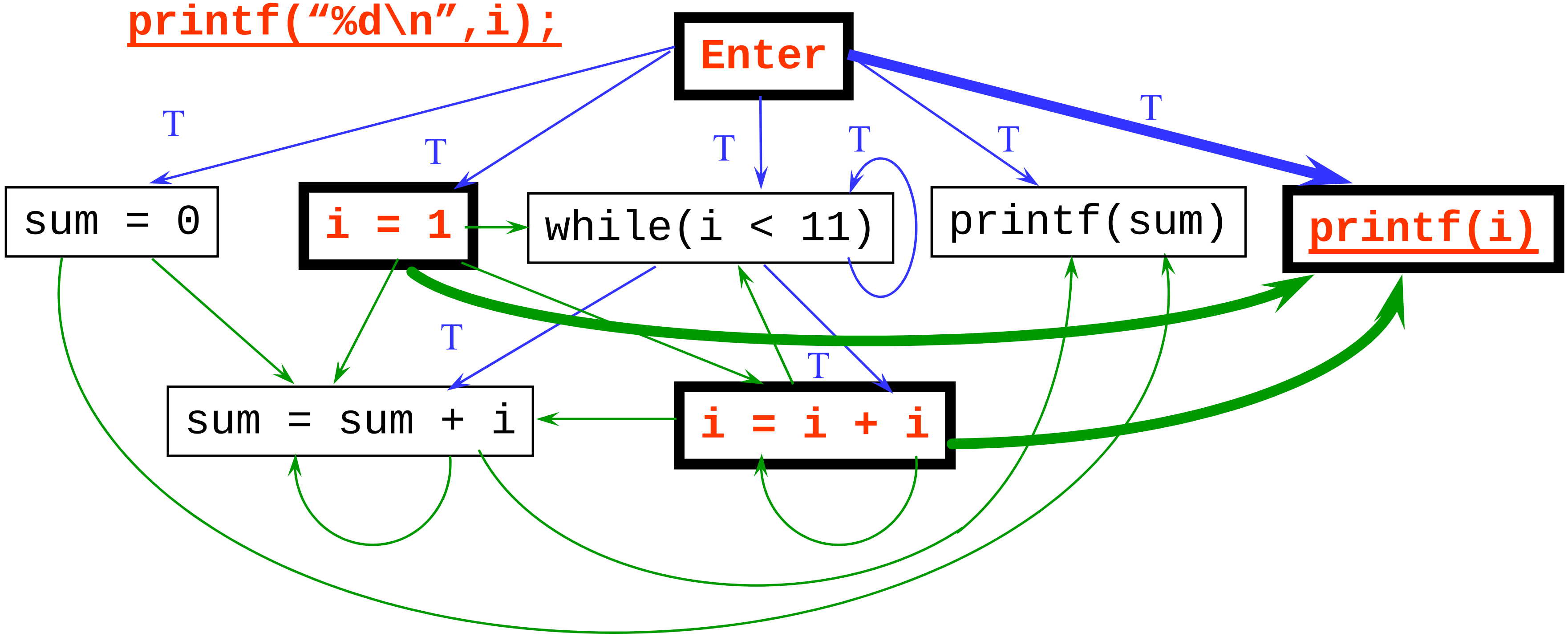
Backward Slice

```
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);
}
```



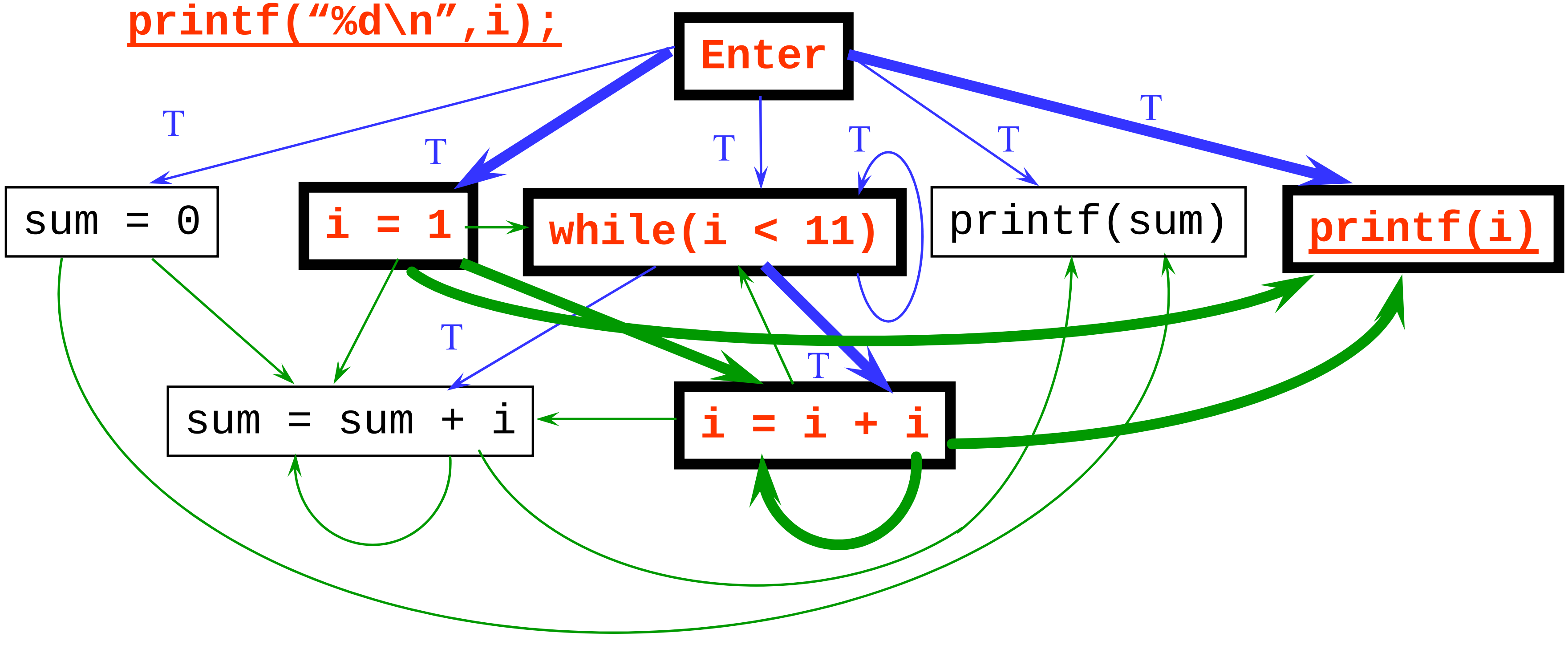
Backward Slice

```
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);
}
```



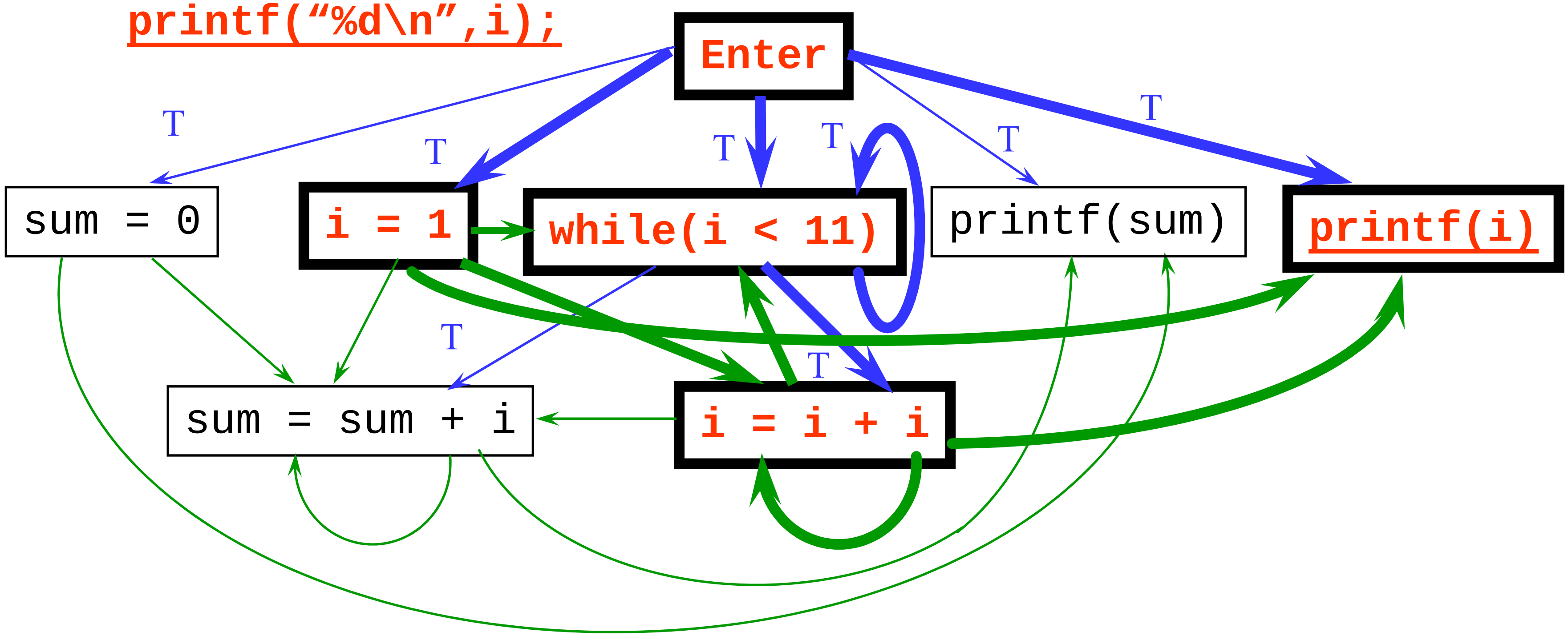
Backward Slice

```
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

```
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);
}
```



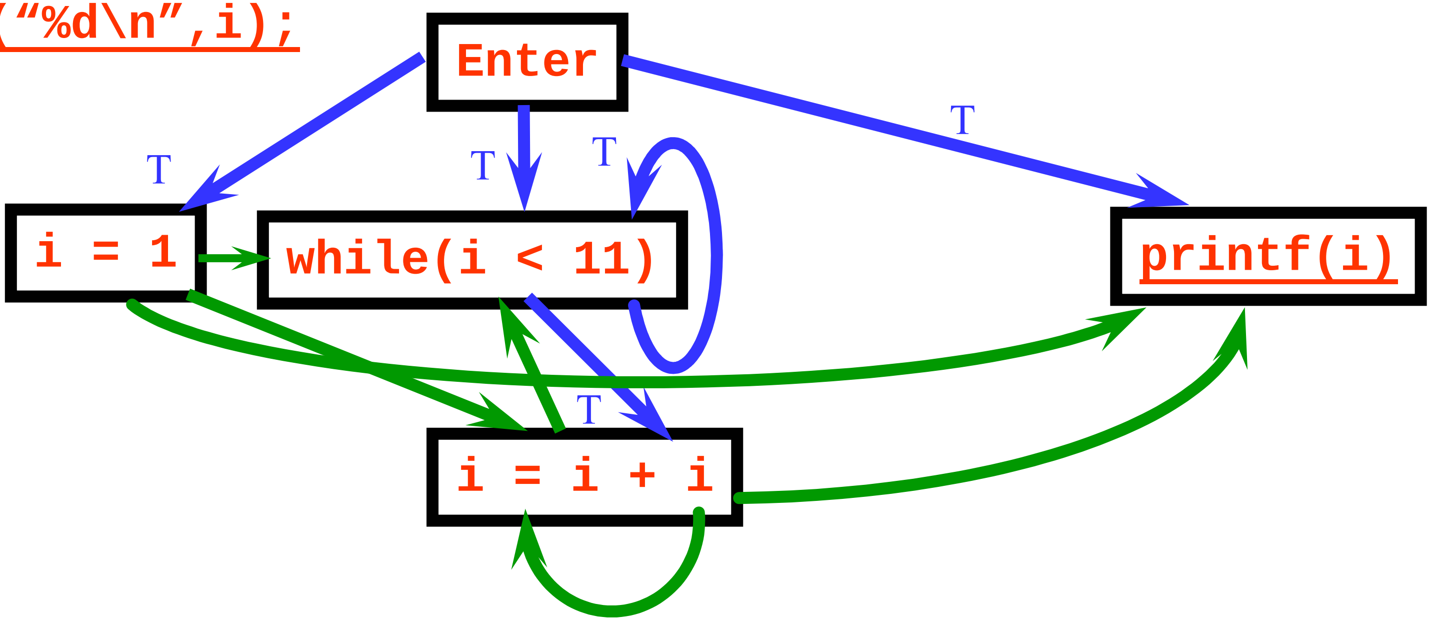
Slice Extraction

```
int main() {

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

            i = i + 1;
      }

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



Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers

Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers

*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);
}
```

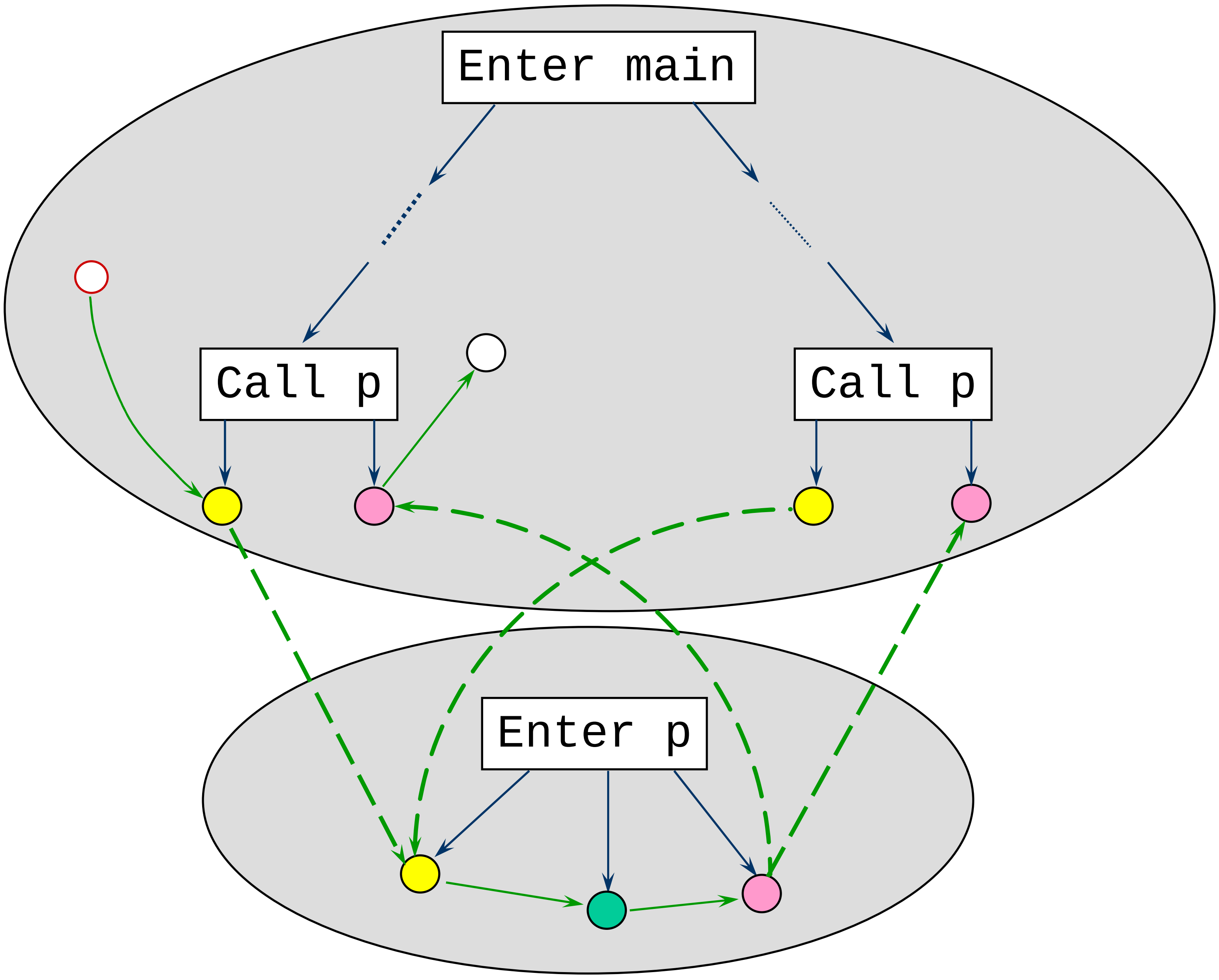
```
int add(int x, int y) {
        return x + y;
}
```

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

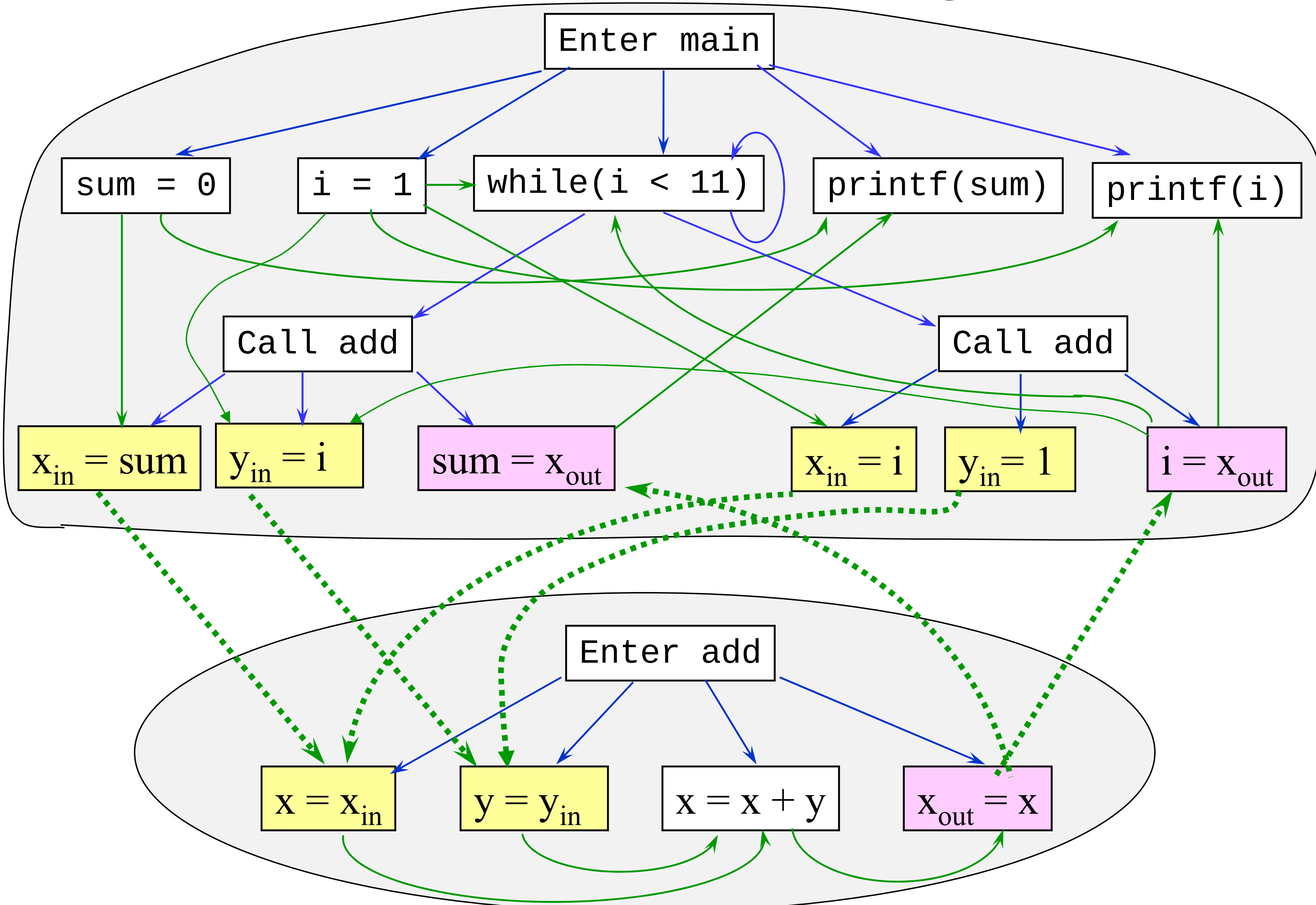
How is an SDG Created

- 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

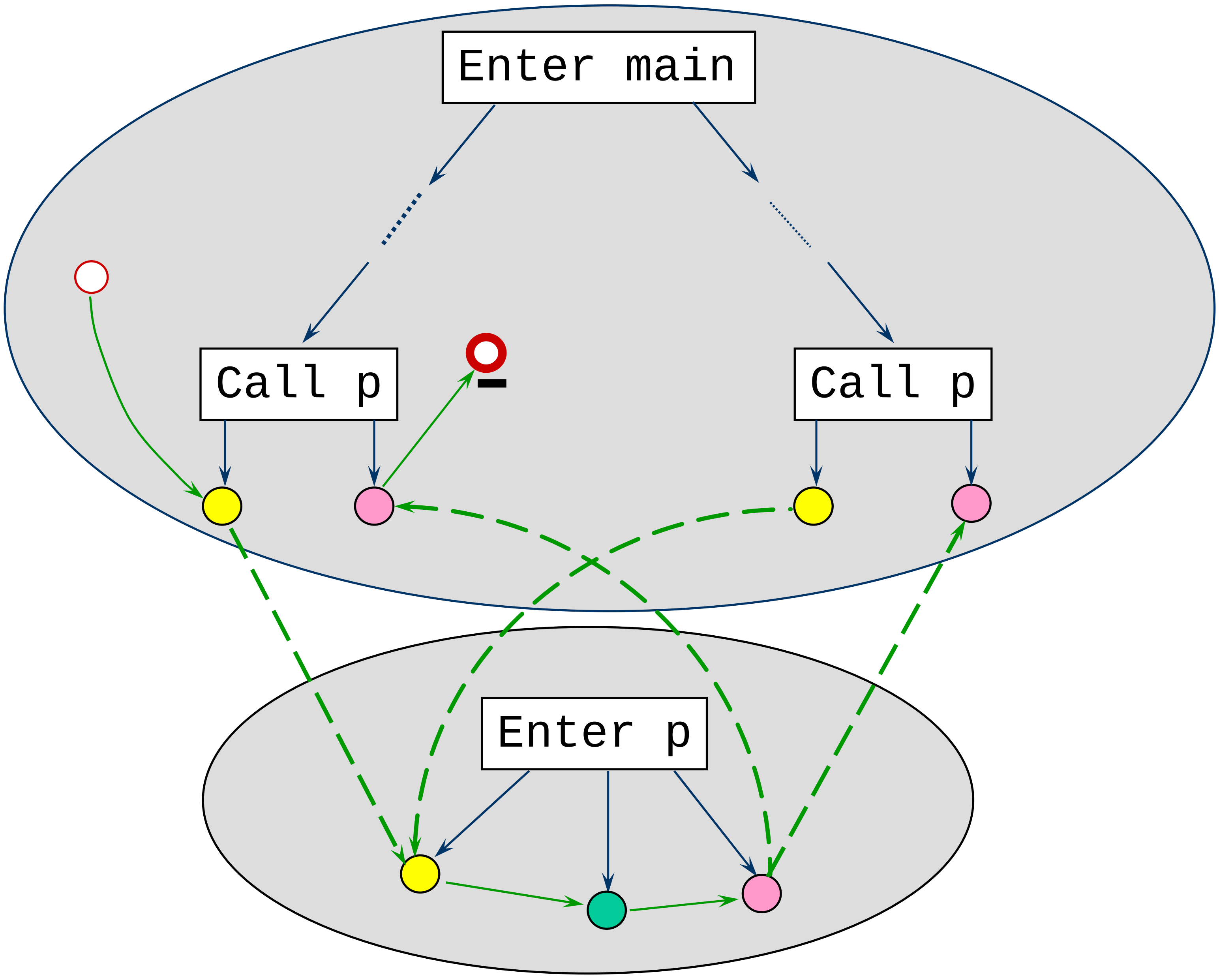
System Dependence Graph (SDG)



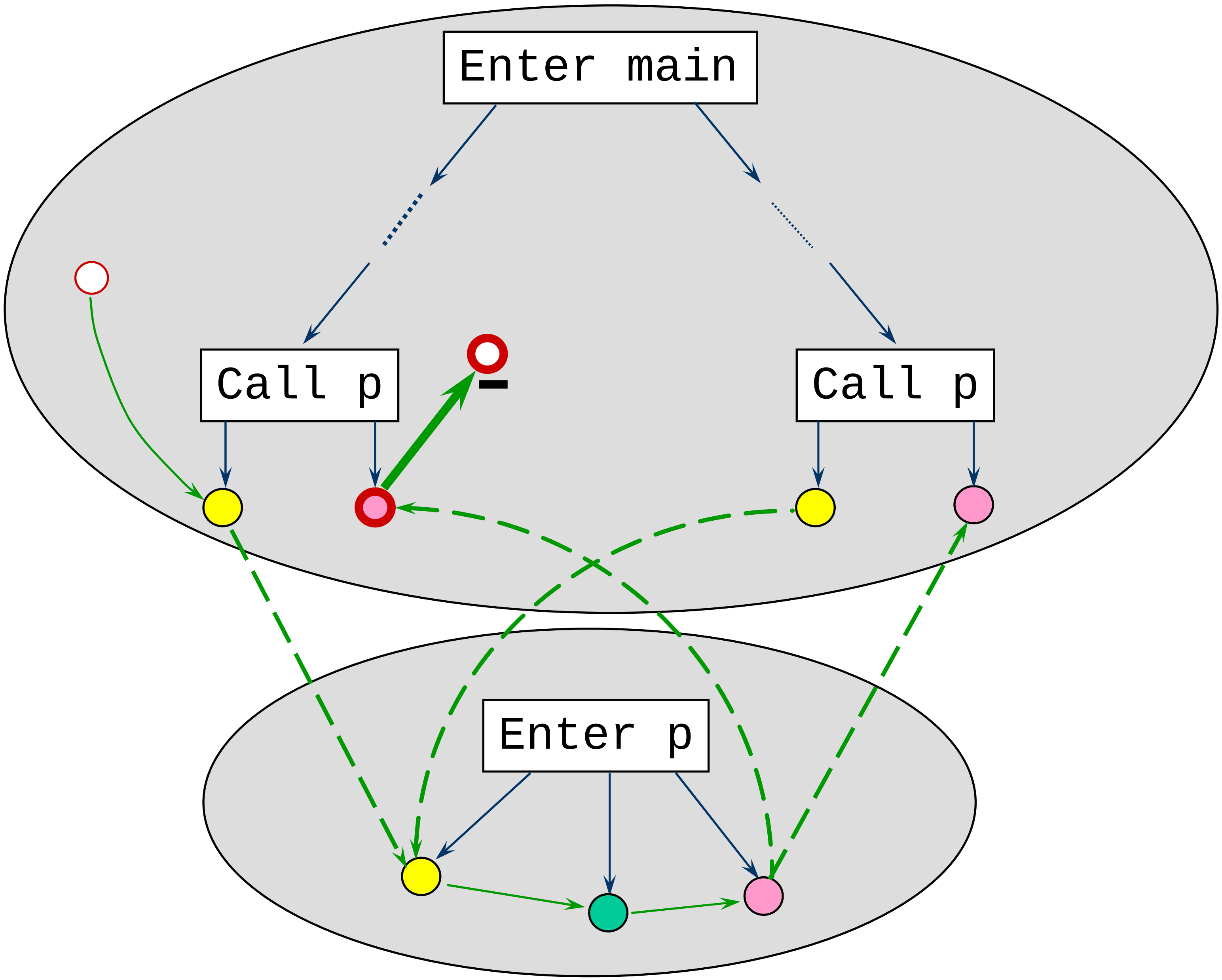
SDG for the Sum Program



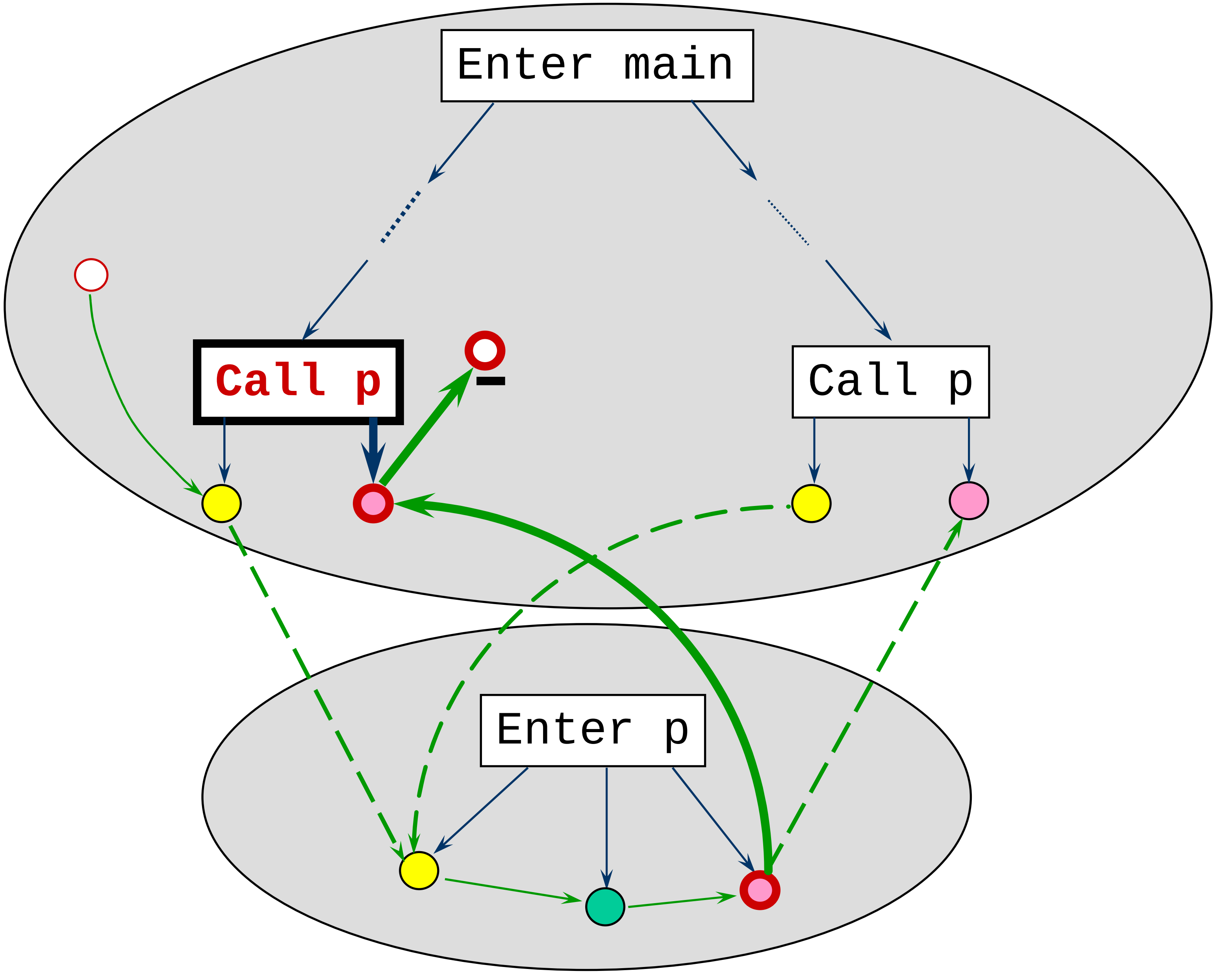
*Inter*procedural Backward Slice



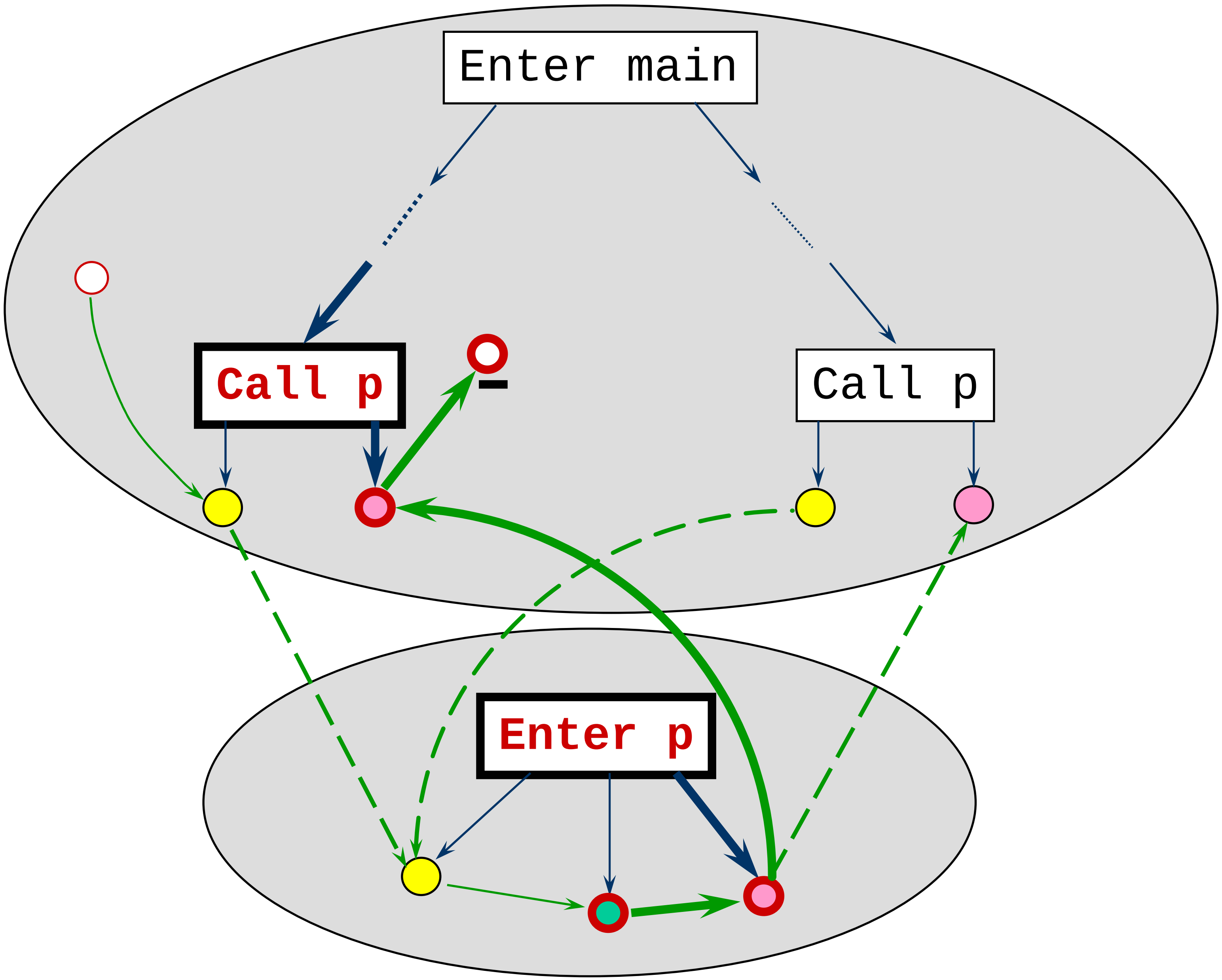
*Inter*procedural Backward Slice (2)



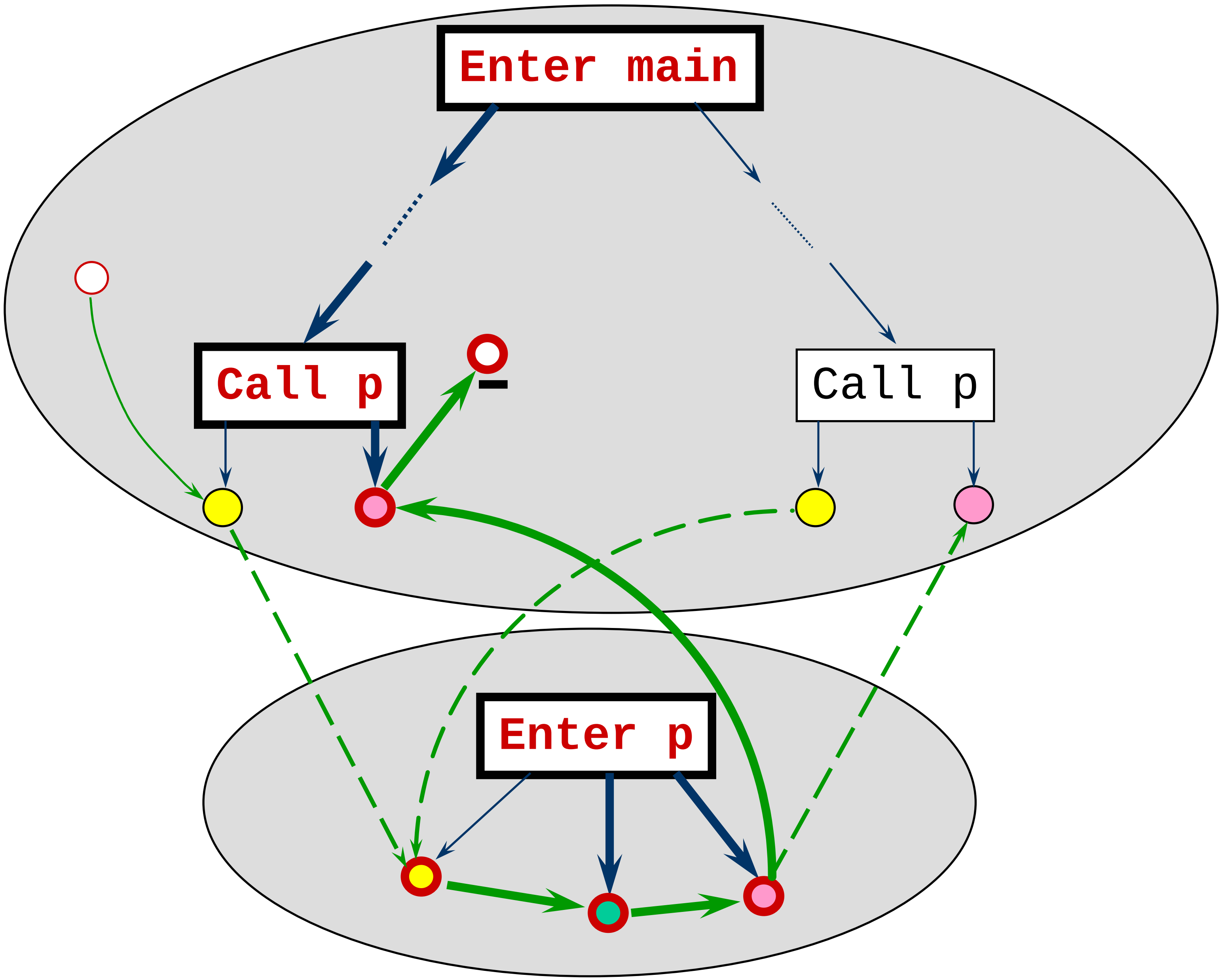
*Inter*procedural Backward Slice (3)



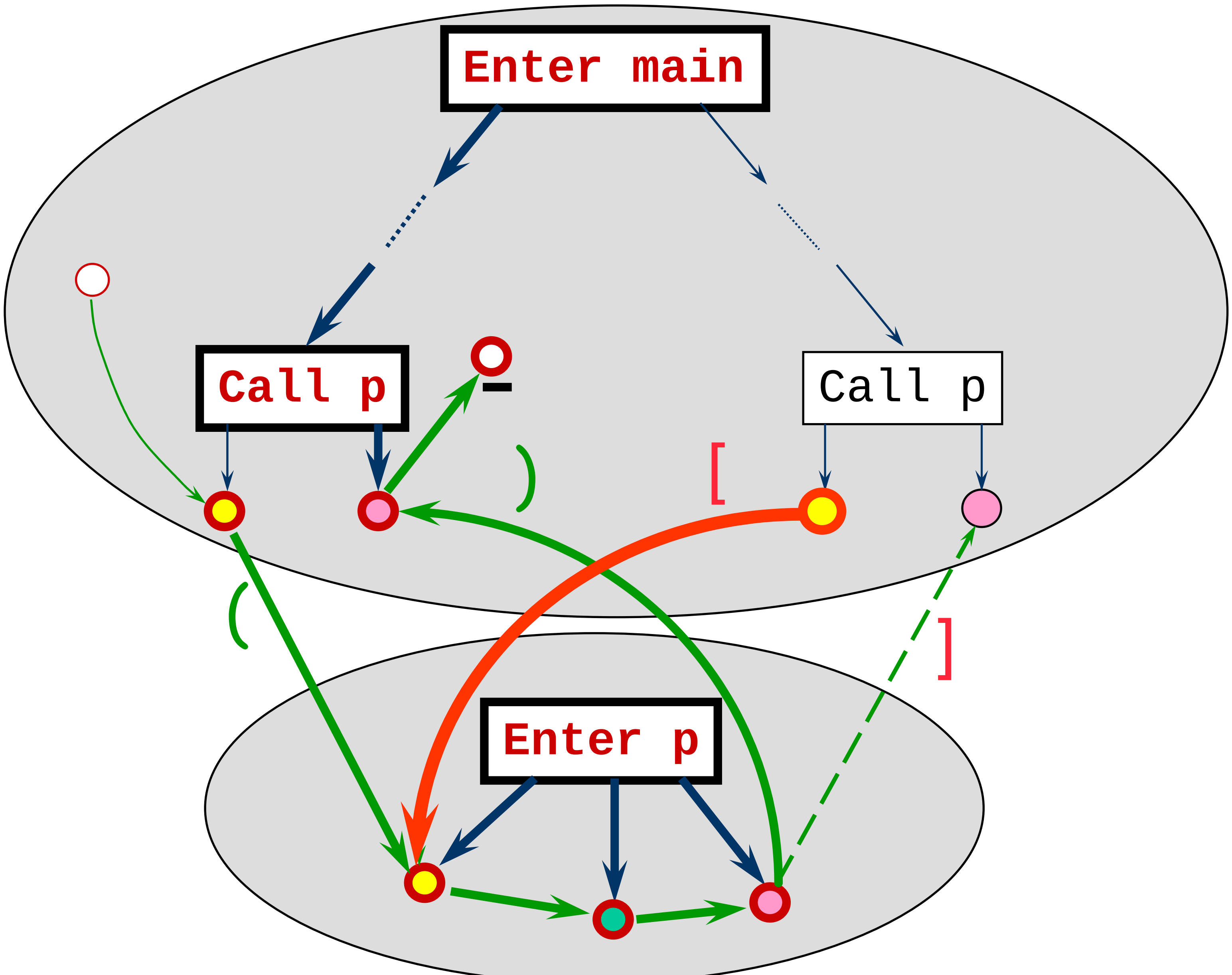
*Inter*procedural Backward Slice (4)



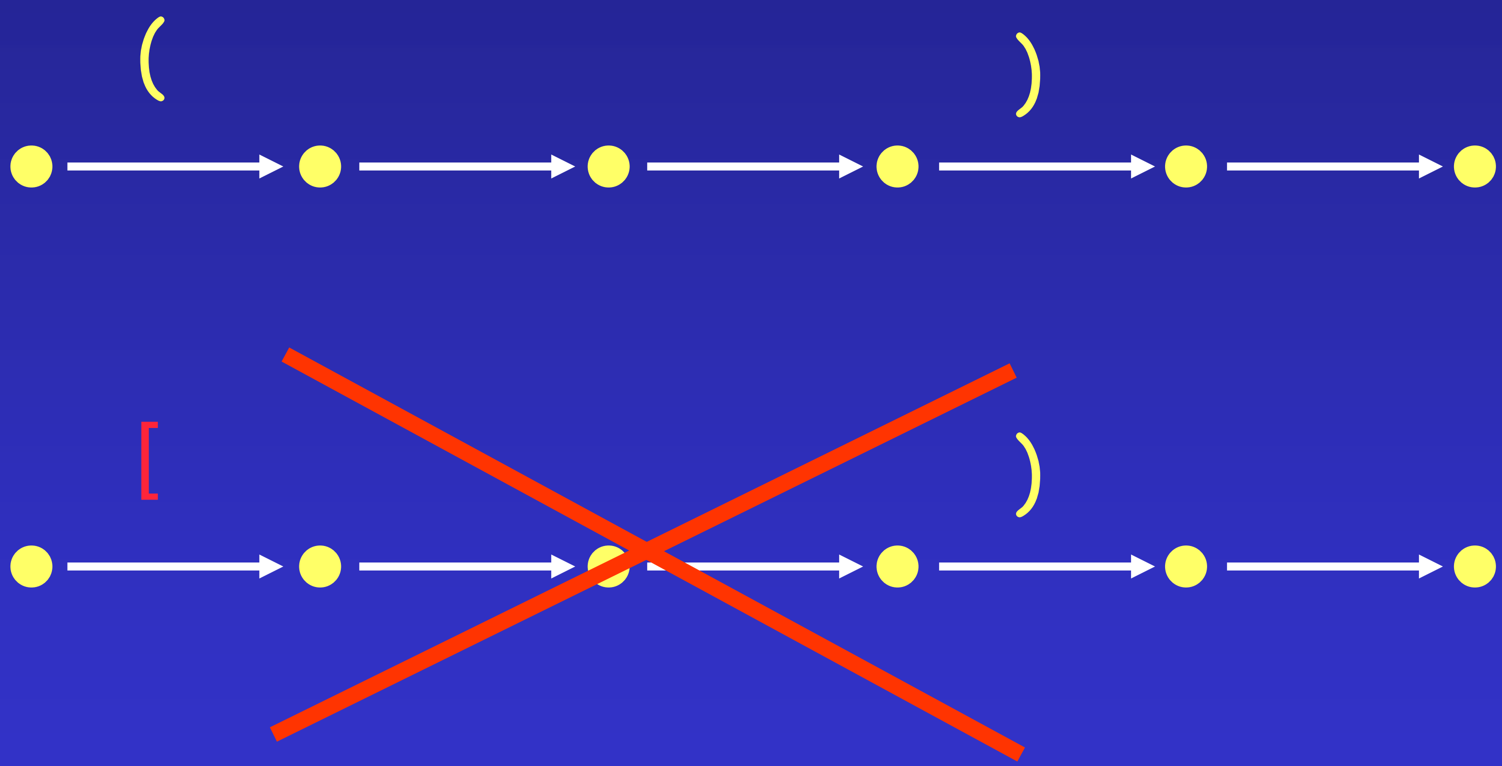
*Inter*procedural Backward Slice (5)



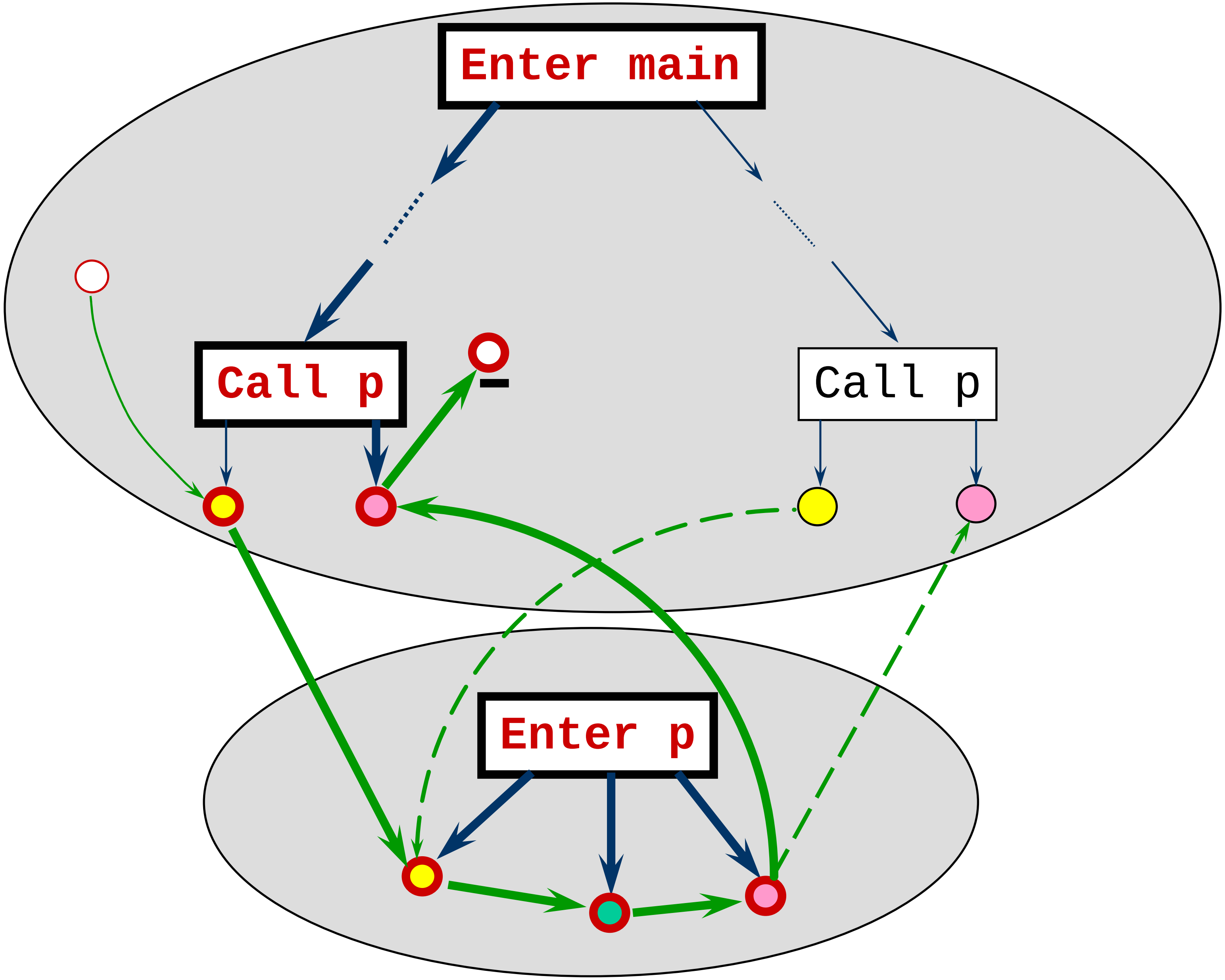
*Inter*procedural Backward Slice (6)



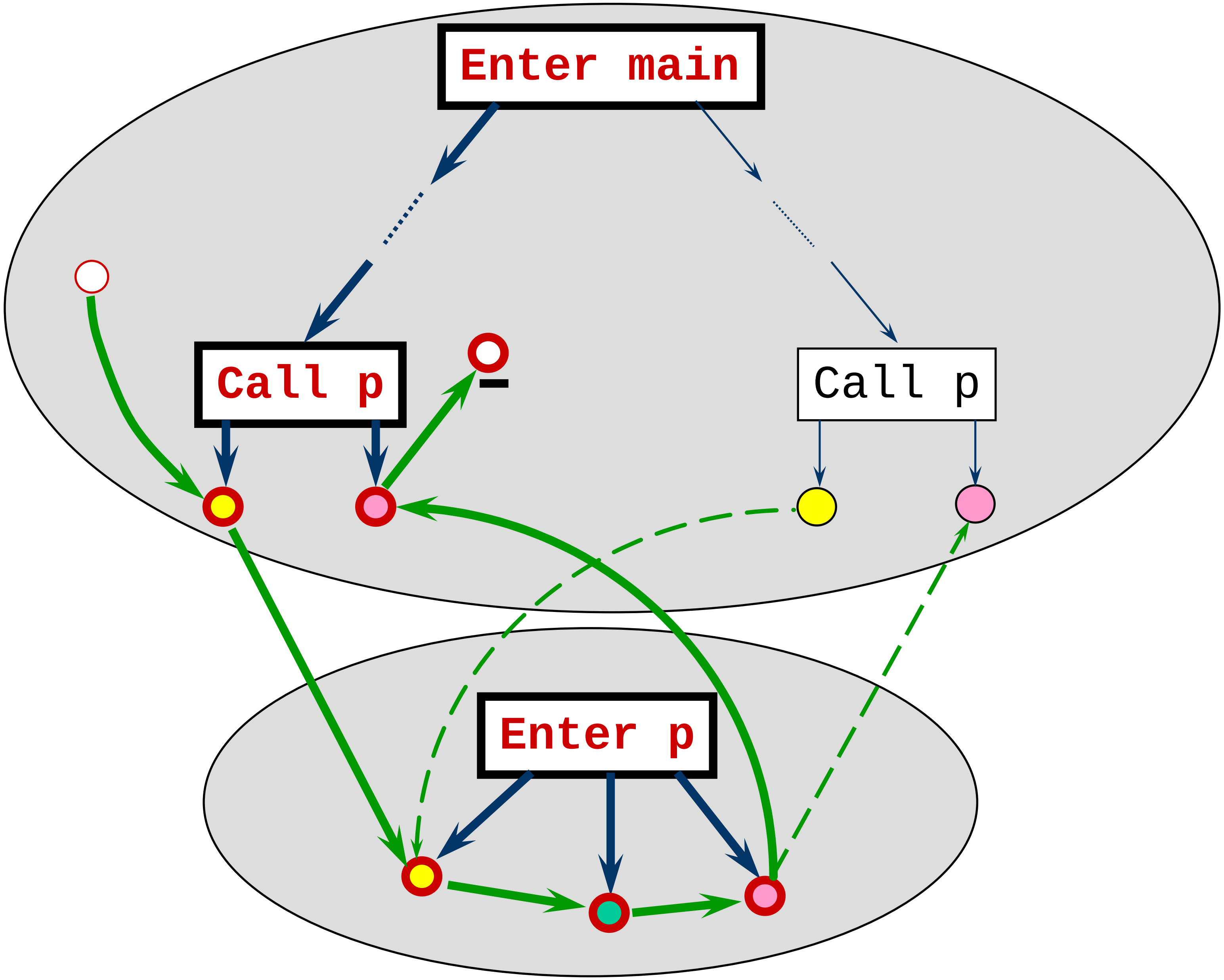
Matched-Parenthesis Path



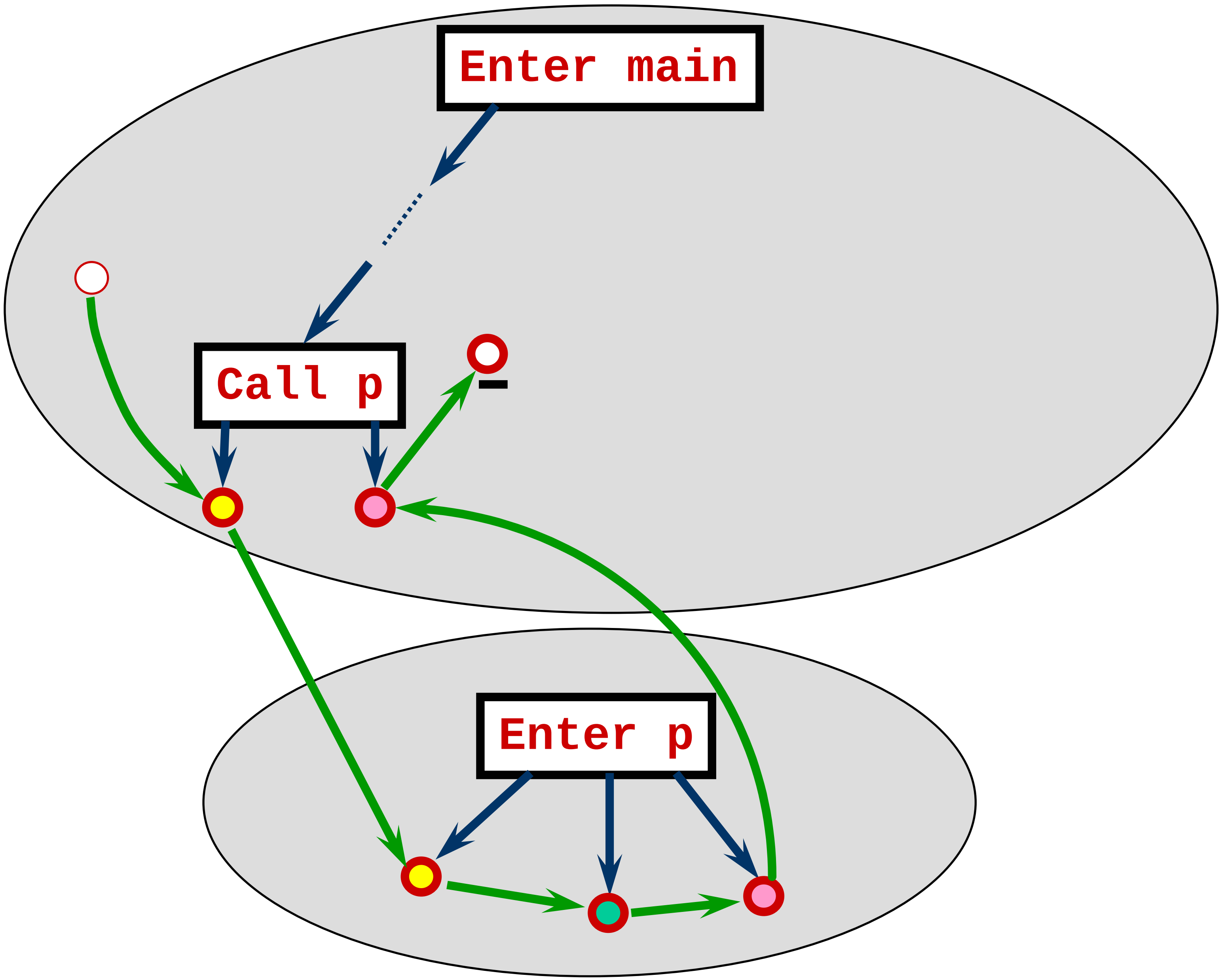
*Inter*procedural Backward Slice (6)



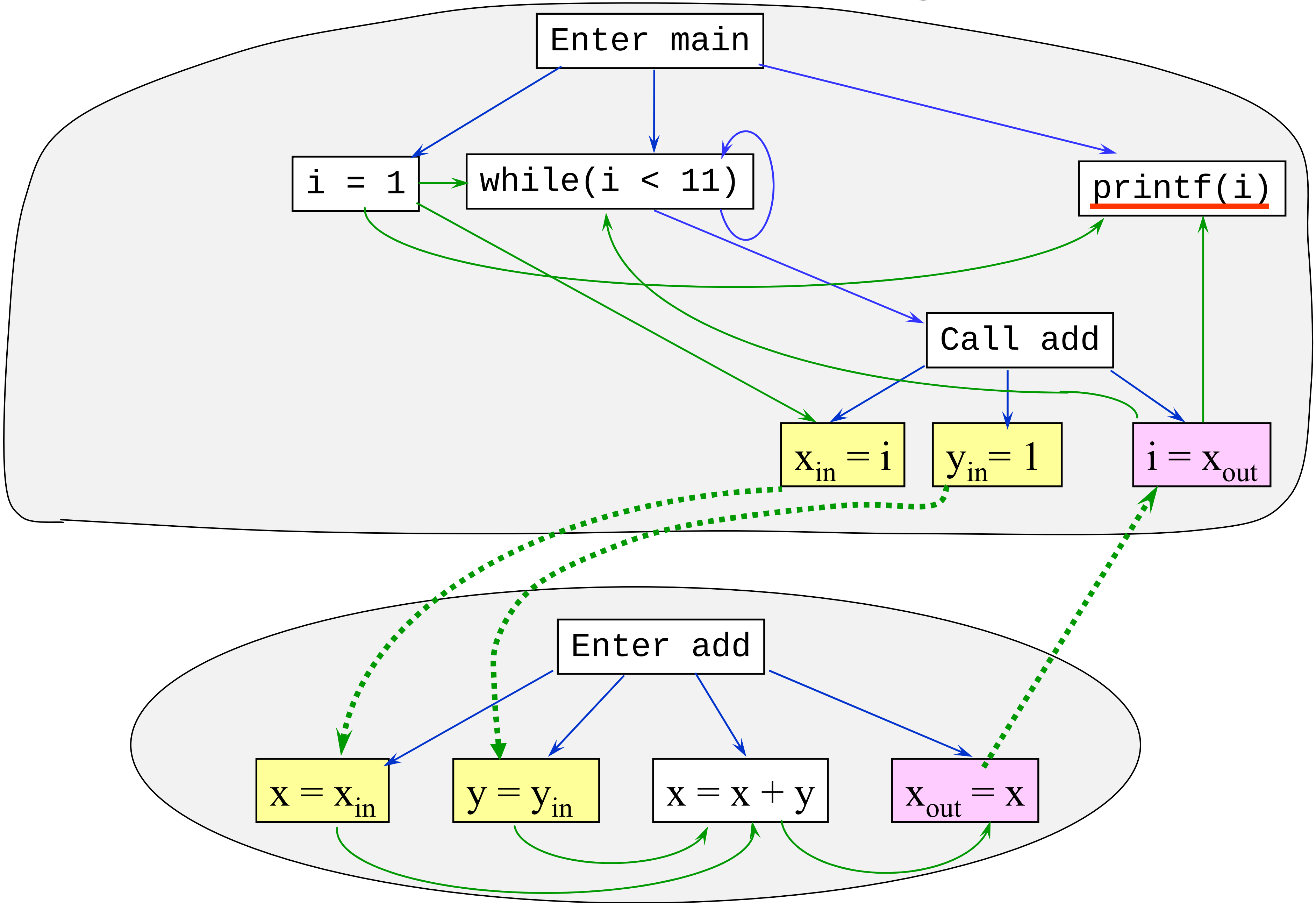
*Inter*procedural Backward Slice (7)



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*

Asymptotic Running Time

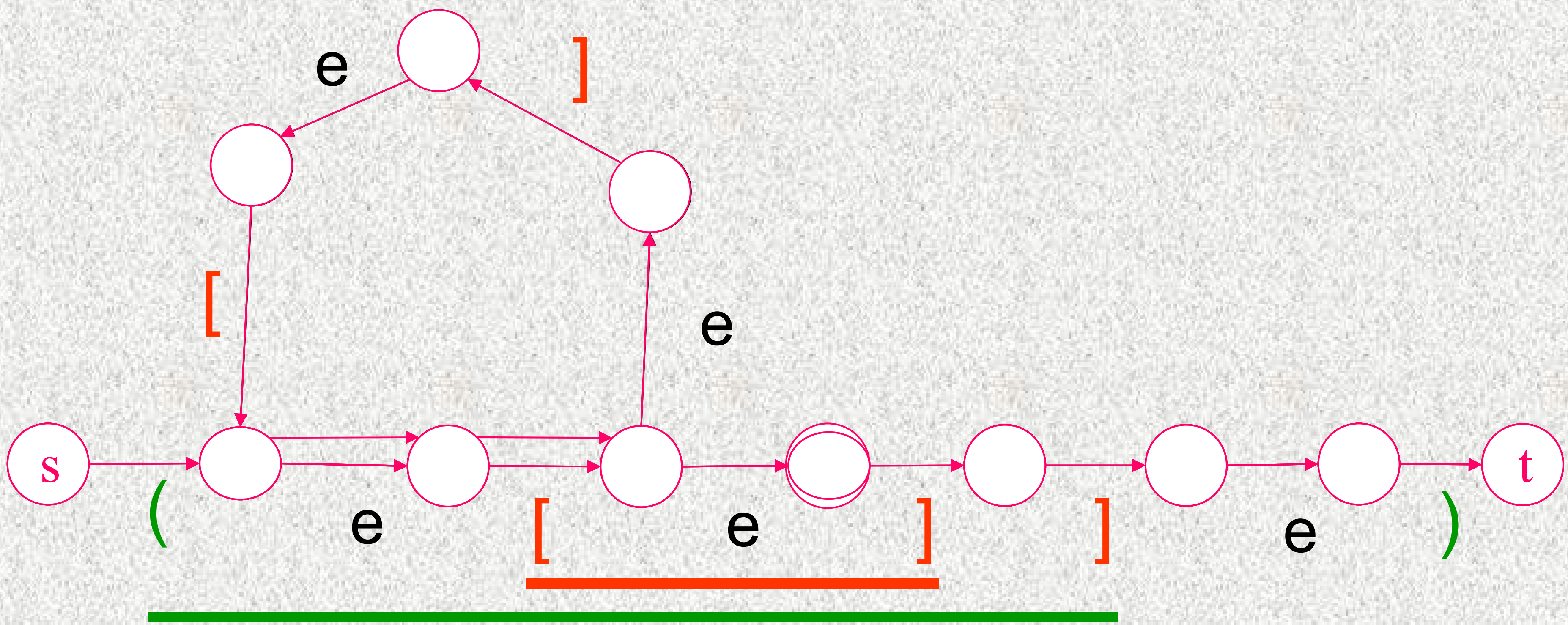
[Reps, Horwitz, Sagiv, & Rosay 94]

- CFL-reachability
  - System dependence graph: N nodes, E edges
  - Running time: $O(N^3)$
- System dependence graph ⇒ Special structure

Running time: $O(E + CallSites$ 🔔 $MaxParams^3)$

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

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

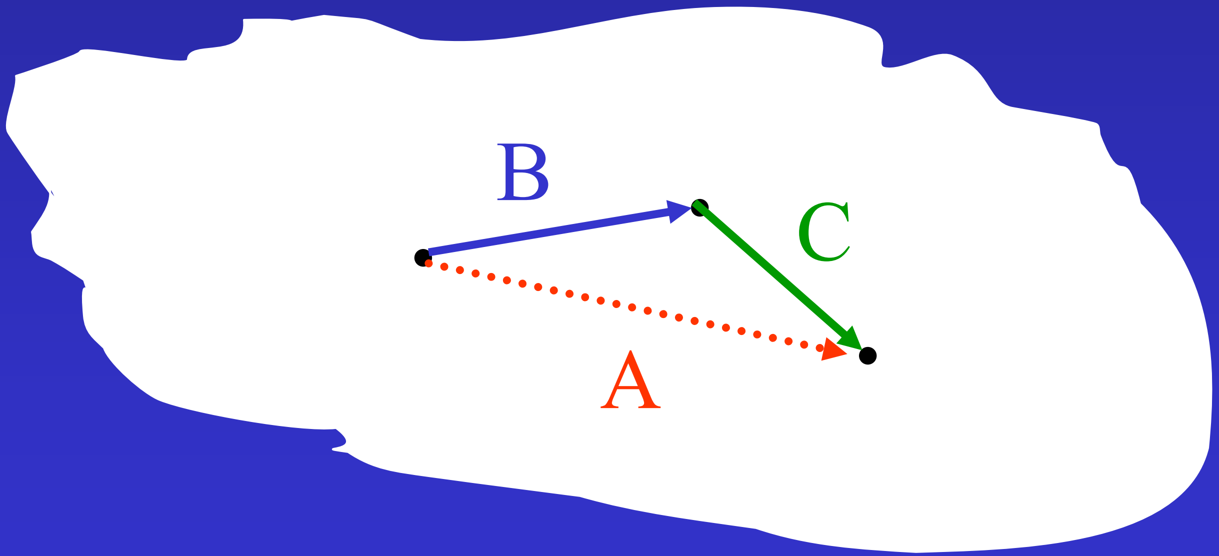


Ordinary Graph Reachability

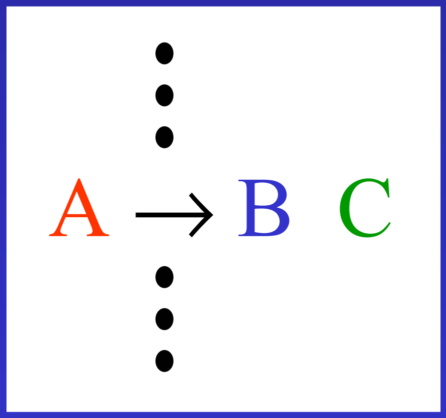
CFL-Reachability

CFL-Reachability via Dynamic Programming

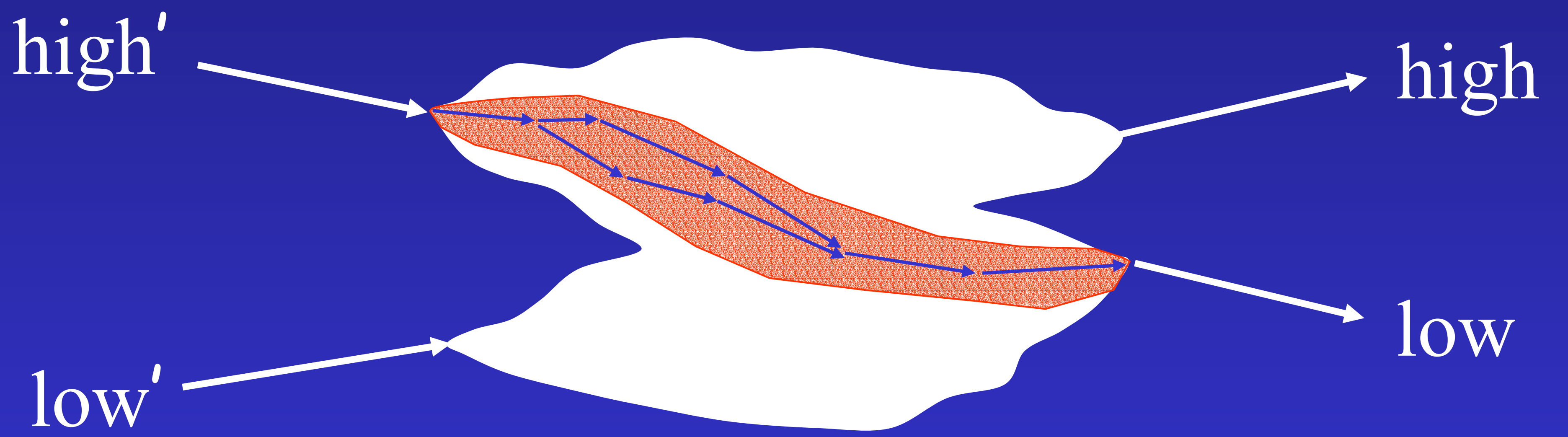
Graph



Grammar



Static Analysis for Mandatory Access Control

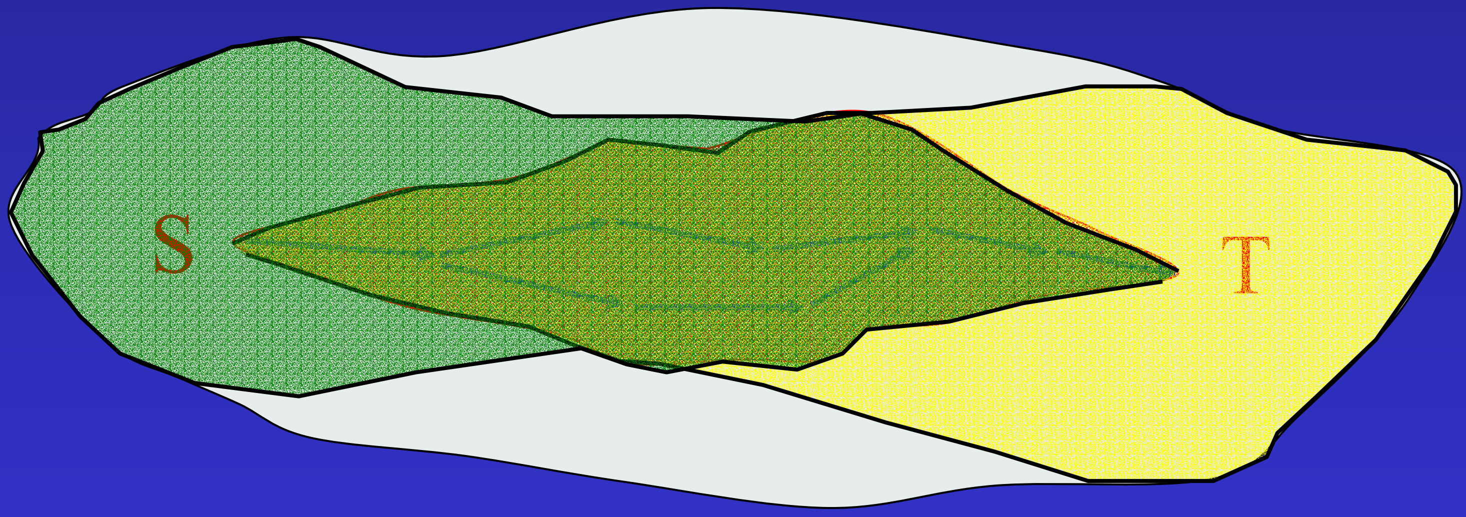


Flow of information from high′ to low?

Program Chopping: Chop(high′, low) = ∅?

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?

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]

Context-Sensitivity and Chopping

```
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"

Context-Sensitivity and Chopping

```
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”

Context-Sensitivity and Chopping

```
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)”

Context-Sensitivity and Chopping

```
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)"

Context-Sensitivity and Chopping

```
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)”

Context-Sensitivity and Chopping

```
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"

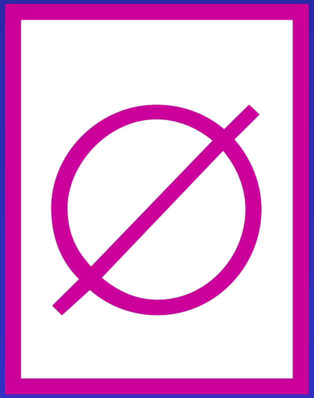
∩

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

Context-Sensitivity and Chopping

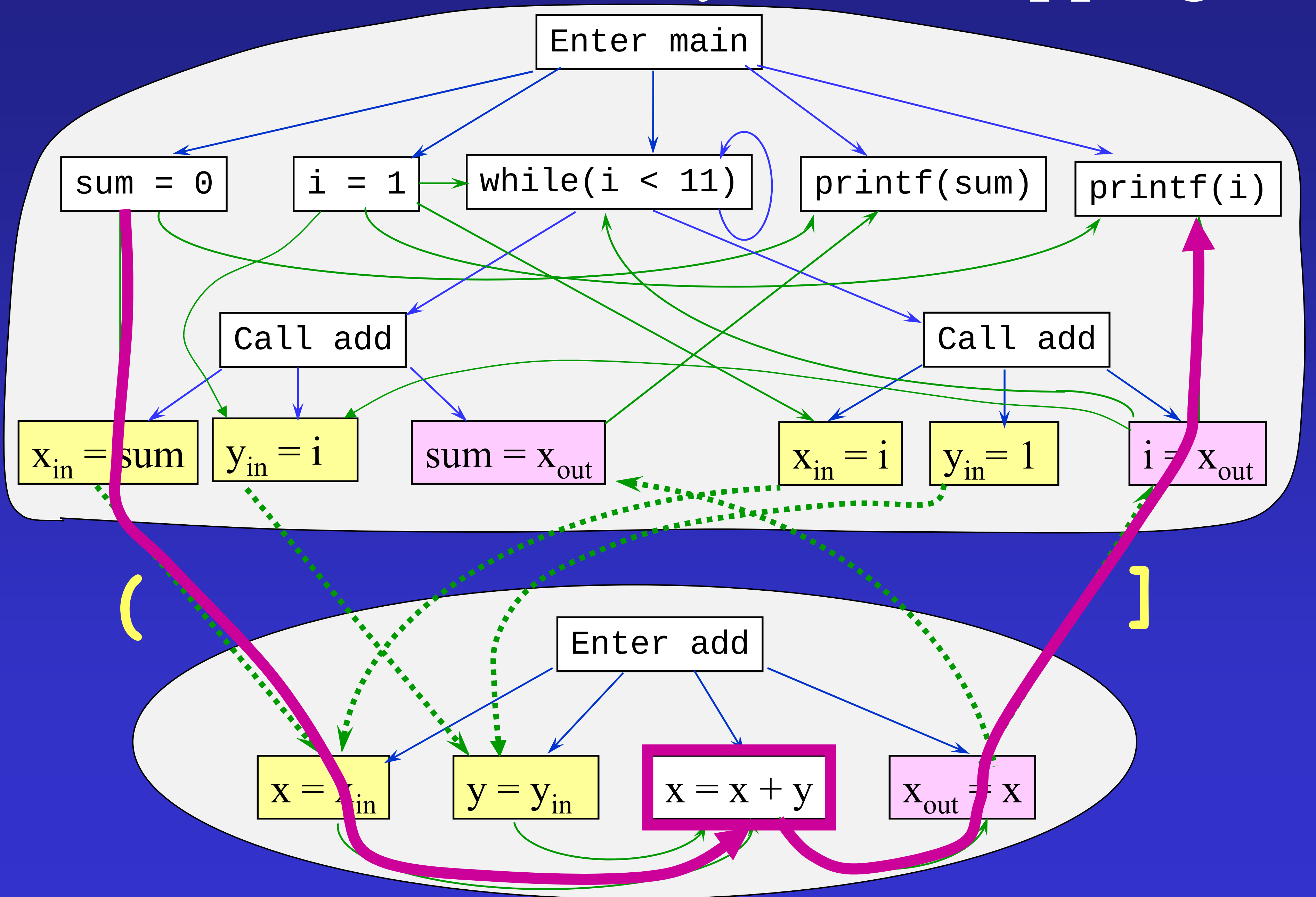
```
int main() {
     int sum = 0;
     int i = 1;
     while (i < 11) {
          sum = add(sum,i);
          i = add(i,1);
     }
     printf(“%d\n”,sum);
     printf(“%d\n”,i);
}
```

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



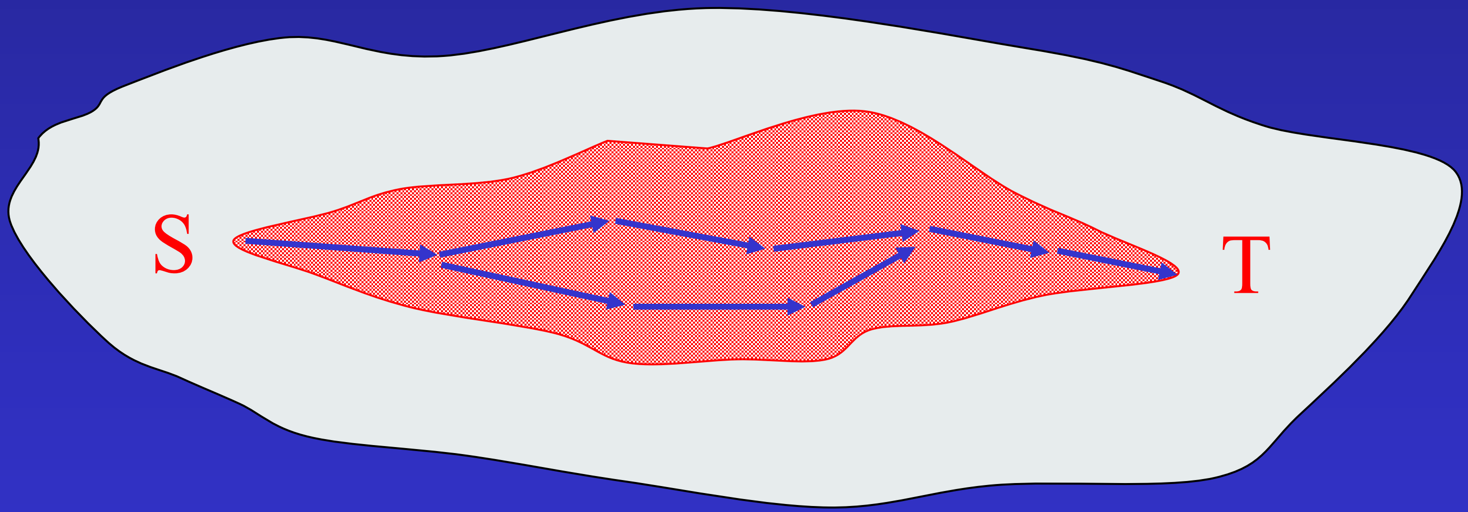
Chop with respect to “sum = 0” and “printf(“%d\n”,i)”

Context-Sensitivity and Chopping



Program Chopping

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



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

More Expressive Memory-Safety Policies

Opportunity: “Checking System Rules” [Engler]

v.unknown:

[(v = malloc(_)) == 0] $\rightarrow_t$ v.null

$\rightarrow_f$ v.notNull

[v = malloc(_)] $\rightarrow$ v.unknown

v.unknown, v.null, v.notNull:

[free(v)] $\rightarrow$ v.freed

v.freed:

[free(v)] $\rightarrow$ “double free!”

[v] $\rightarrow$ “use after free!”

More Expressive Access Policies

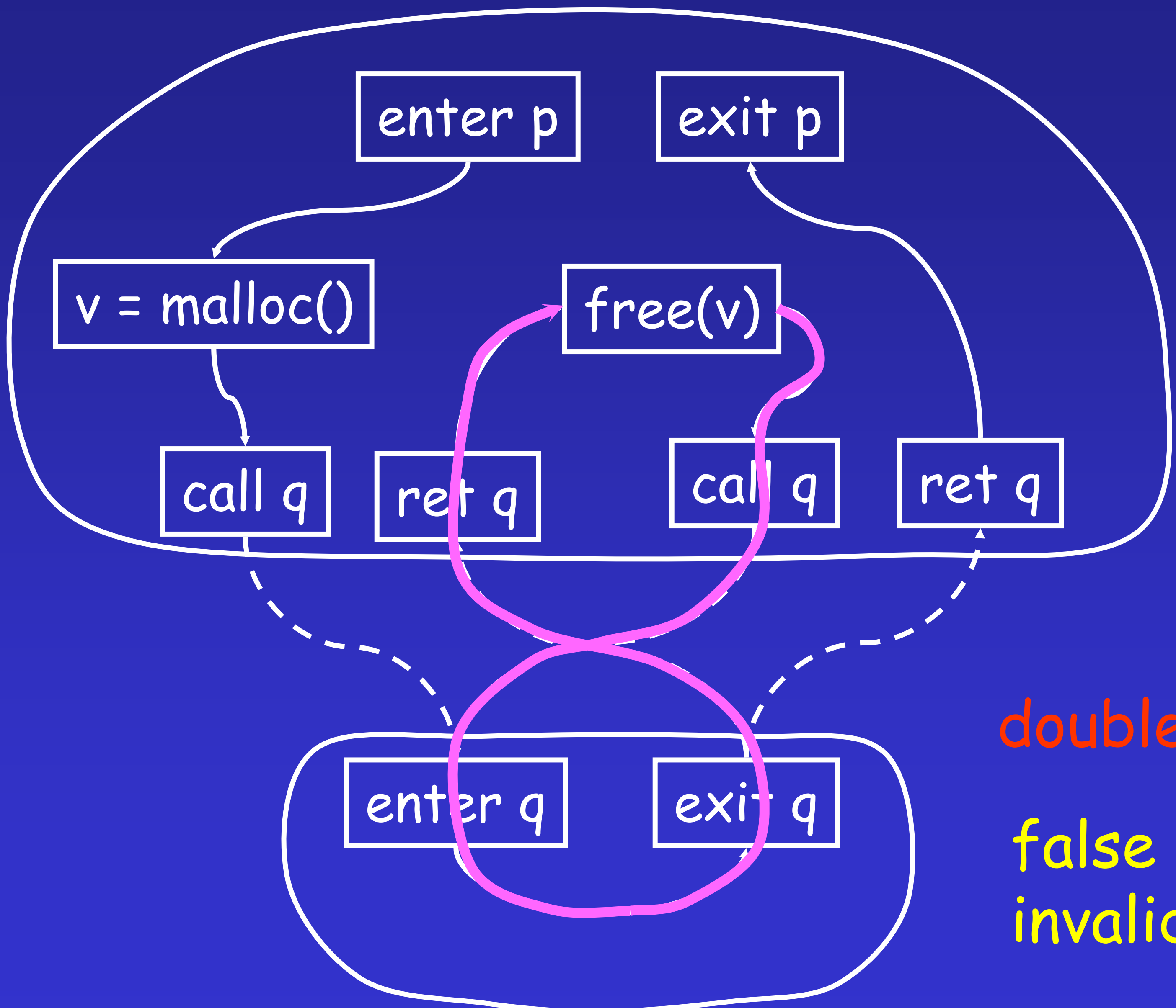
"No send after file read"

f.hasNotBeenRead:
[read(f)] → f.hasBeenRead
[send] → f.hasNotBeenRead

f.hasBeenRead:
[read(f)] → f.hasBeenRead
[send] → "send after file f was read!"

A model-checking problem:
Control-flow graph ⑥ Security automaton?

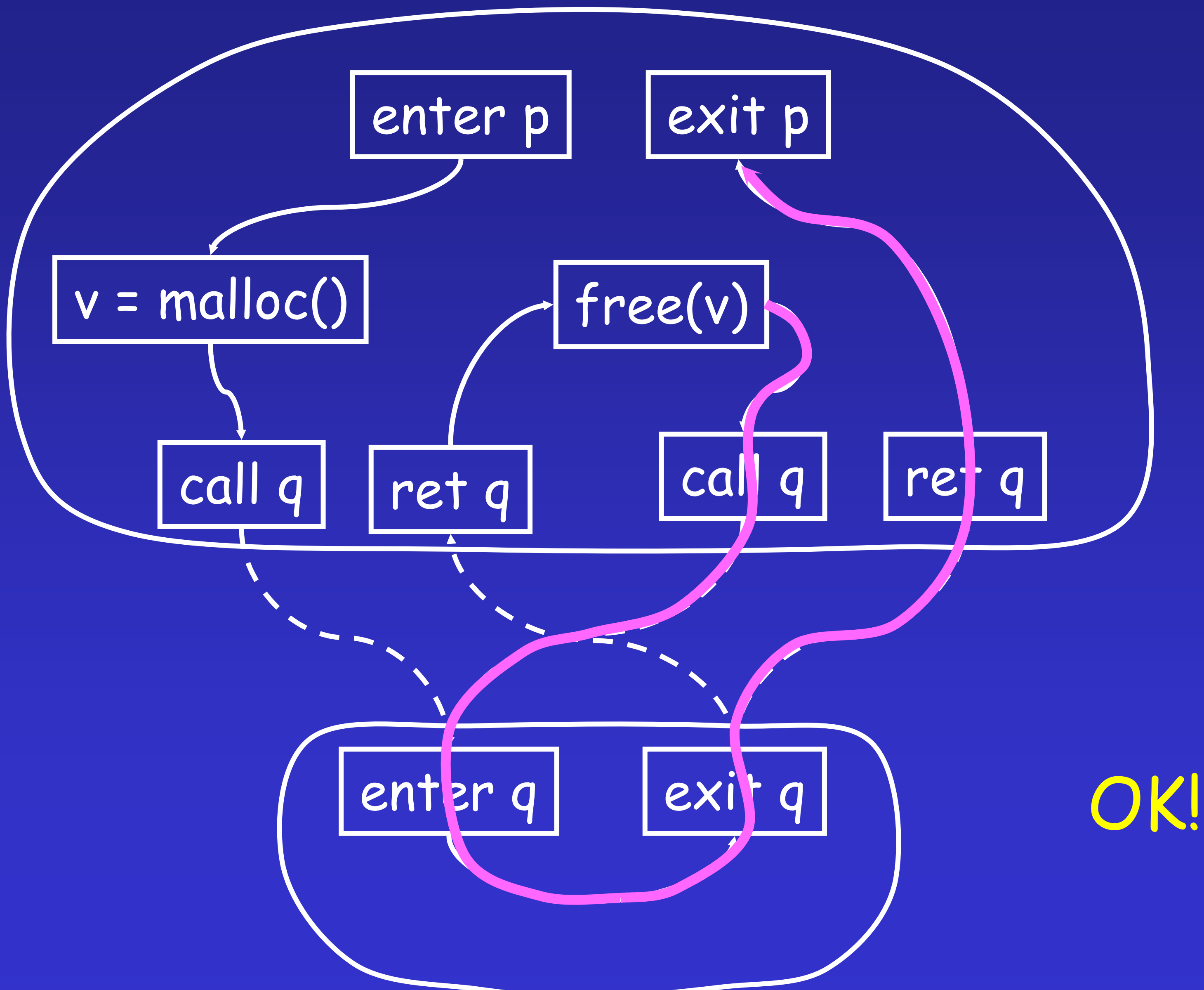
The Need for Context Sensitivity



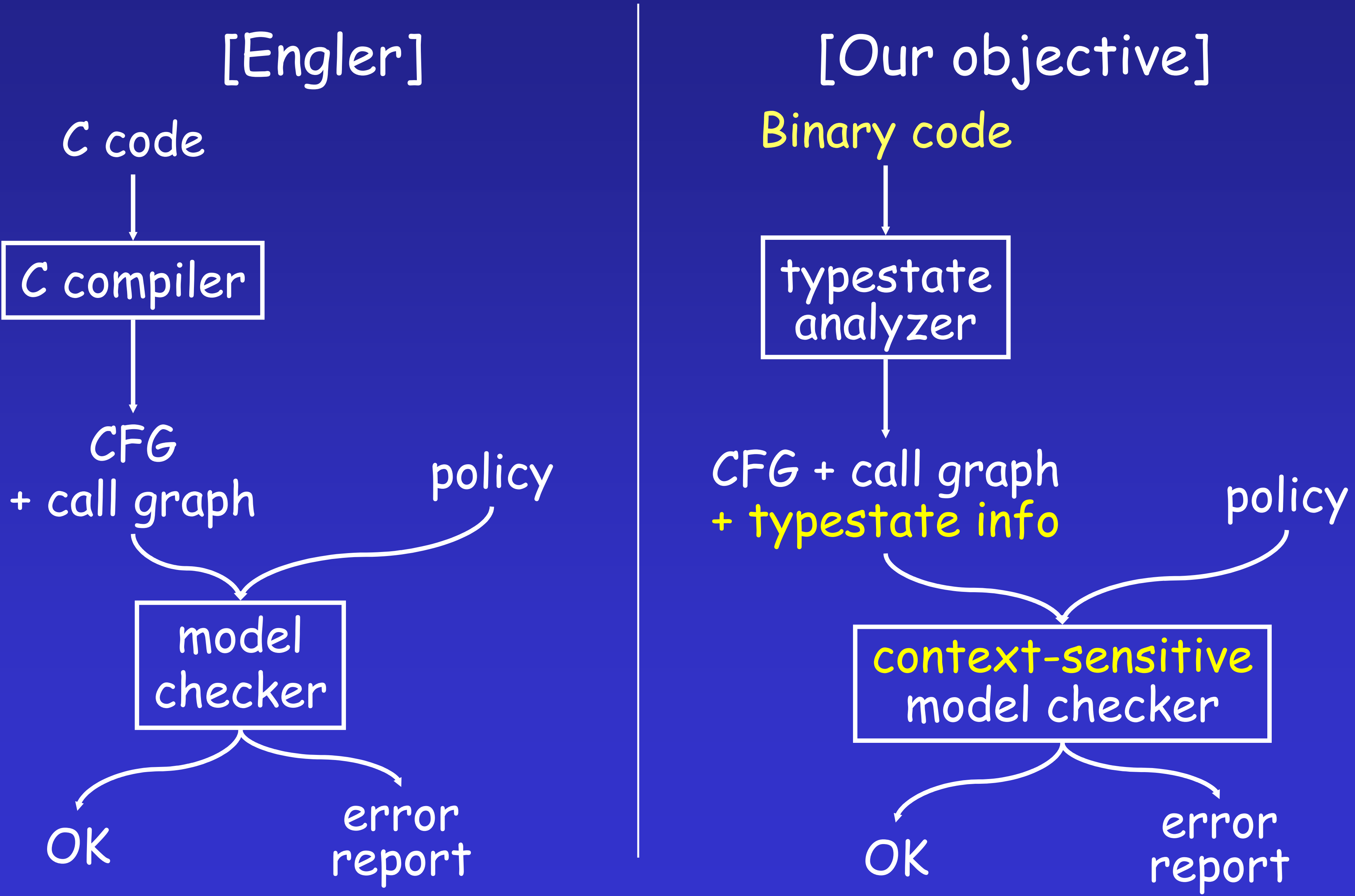
double free!

false alarm: invalid path!

The Need for Context Sensitivity



A Richer Setting



The Need for Context Sensitivity

- Information-flow analysis
- Virus scanning via model checking [Mihai]
- Buffer-overrun detection [Vinod]
- Dynamic traces of system calls [Jon]

Reps et al: Context Sensitivity

- Program slicing and chopping
  - Horwitz, Reps, & Binkley, TOPLAS 90
  - Reps, Horwitz, Sagiv, & Rosay, FSE 94
- Dataflow analysis
  - Reps, Horwitz, & Sagiv, POPL 95
- Survey of a collection of analyses
  - Reps, IST 98
- Context-sensitive model checking
  - Benedikt, Godefroid, & Reps, ICALP 01

Outline

- Slicing and Dependence Graphs
- CodeSurfer
- Interprocedural Slicing and Chopping
- <u>Points-to Analysis</u>
- Demos

Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers

Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers
  - CodeSurfer: flow-insensitive points-to analysis
  - TVLA: flow-sensitive analysis of heap-allocated storage

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);
}
```

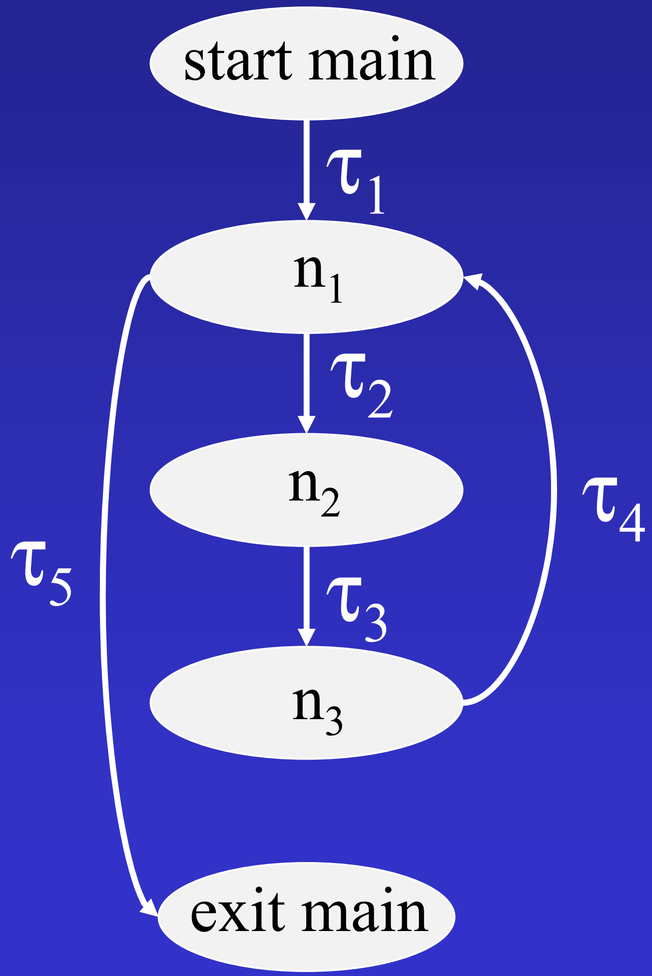
```
int add(int x, int y)
{
  return x + y;
}
```

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 Dataflow Analysis



$$V[n_1] = \tau_1(V[main]) \otimes \tau_4(V[n_3])$$

$$V[n_2] = \tau_2(V[n_1])$$

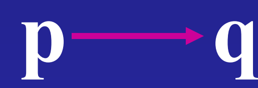
$$V[n_3] = \tau_3(V[n_2])$$

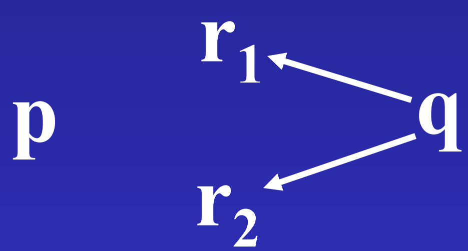
$$V[exit] = \tau_5(V[n_1])$$

$$V[main] = \varnothing$$

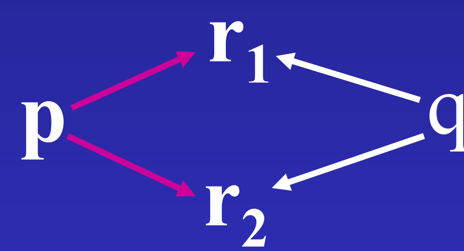
Flow-Sensitive Points-To Analysis

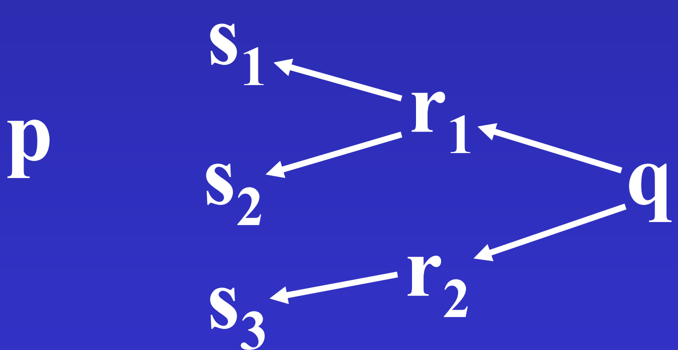
p = &q;



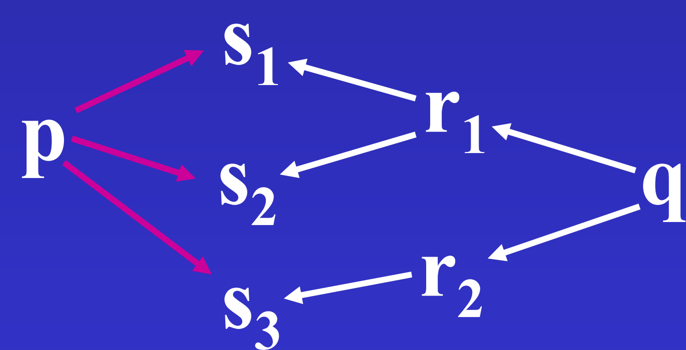


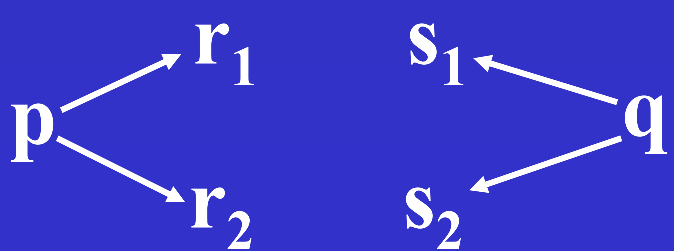
p = q;



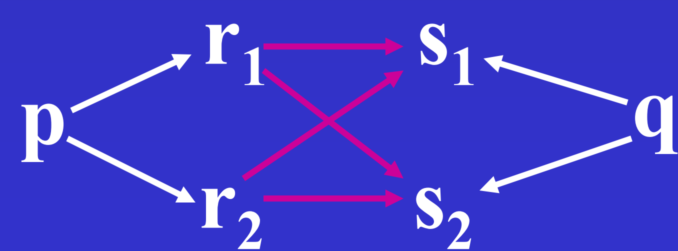


p = *q;

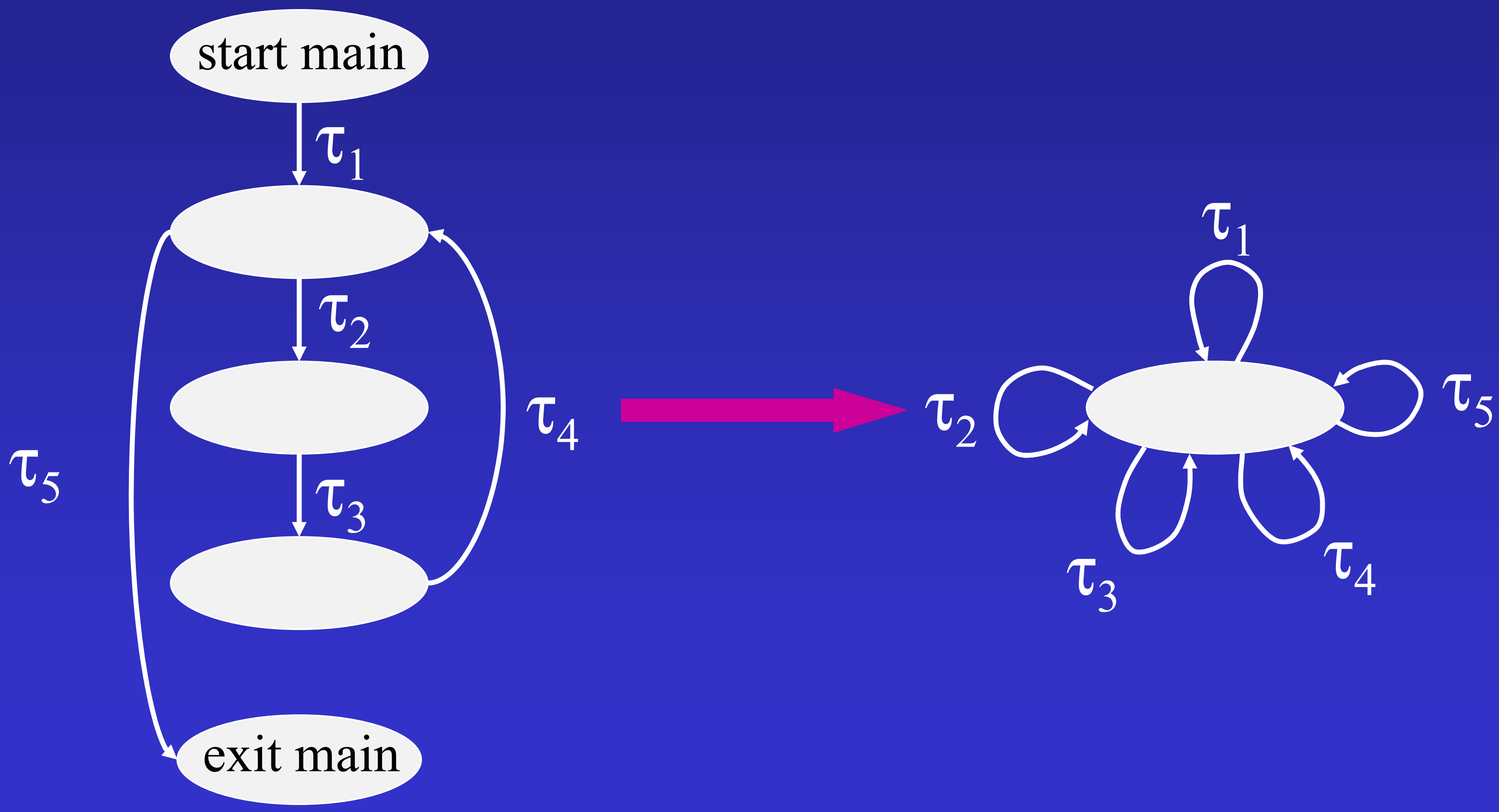




*p = q;



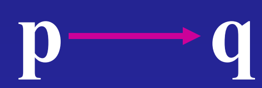
Flow-Sensitive → Flow-Insensitive



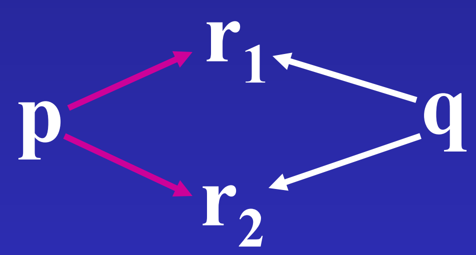
Flow-Insensitive Points-To Analysis

[Andersen 94, Shapiro & Horwitz 97]

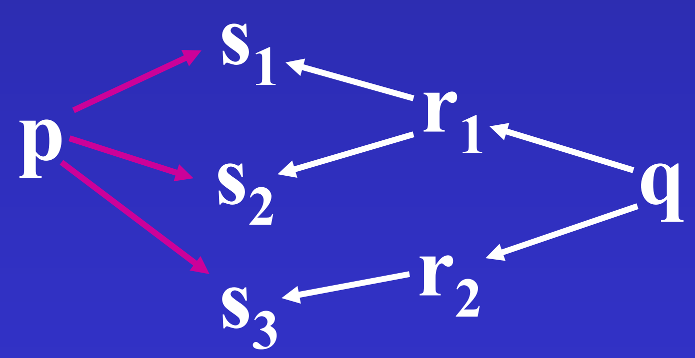
p = &q;



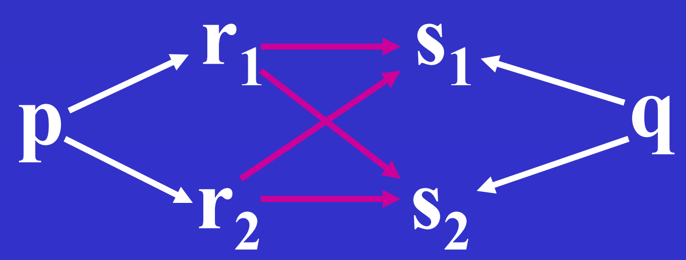
p = q;



p = *q;

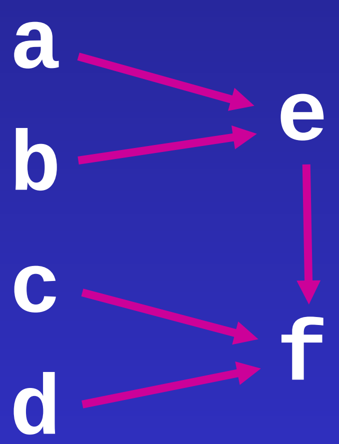


***p = q;**

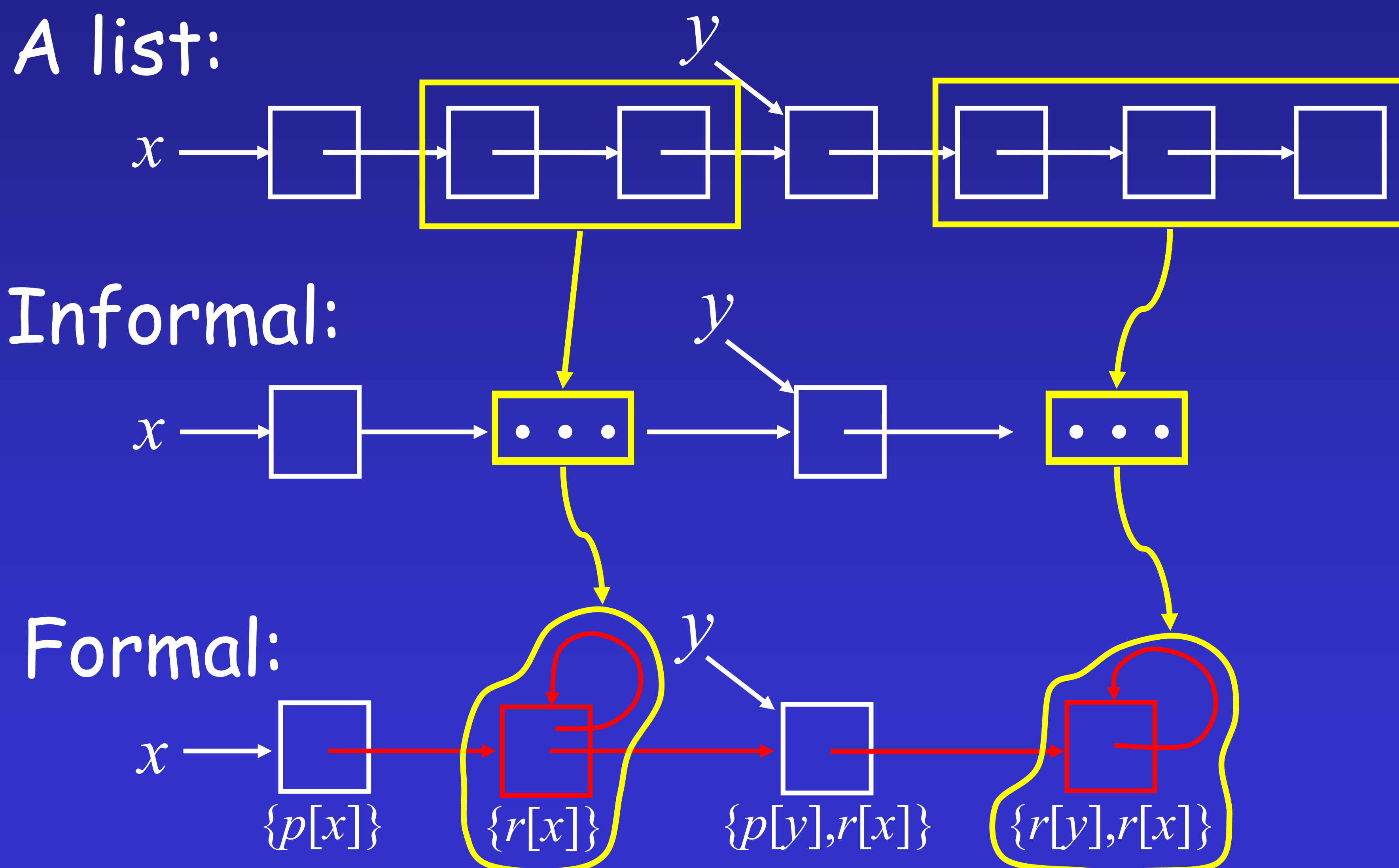


Flow-Insensitive Points-To Analysis

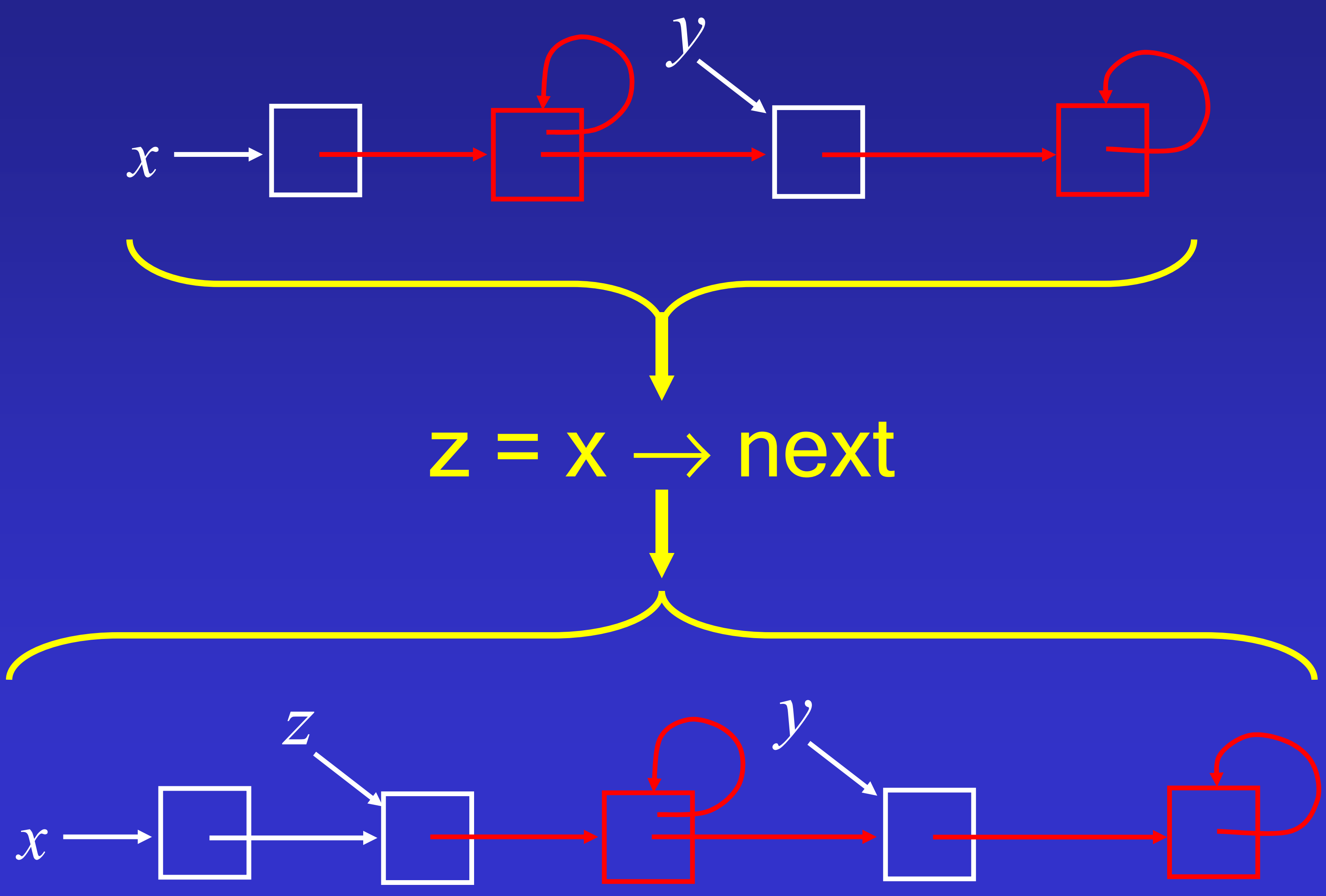
```
 a =  &e;
 b =   a;
 c =  &f;
*b =   c;
 d =  *a;
```



Shape Analysis: Formalizing ". . ."



Shape Analysis: Formalizing "..."



The Need for Pointer/Shape Analysis

- Indirect function calls
  - (*fp)(e1,e2,e3);
  - What does fp point to?
  - [Mihai, Vinod, Jon]
- "Shape" of heap-allocated data structures
  - Information flow through data structures [Reps]
  - Relating behavior patterns [Mulhern]

Limitations of Static Analysis

- Undecidable questions (yes/no)
- Safe
  - yes/maybe
  - no/maybe
  - yes/no/maybe
- Time/space tradeoffs
  - context-sensitive/context-insensitive
  - flow-sensitive/flow-insensitive
  - Find a sweet spot

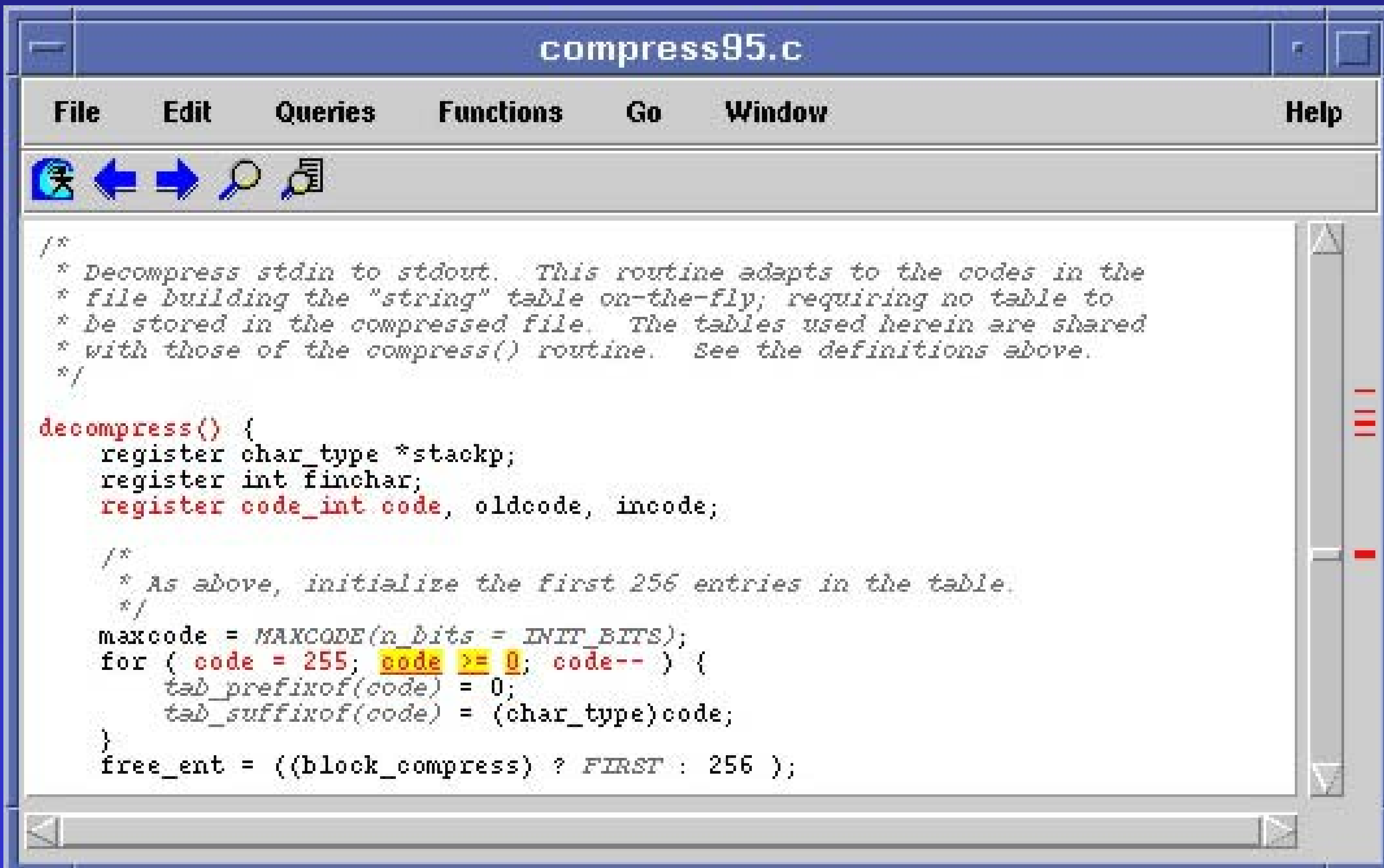
Static Analysis of Binaries

+Single language to deal with

–Some “inherent” obfuscation

- operations on registers (vs. variables)
- very weak types (8-bit, 16-bit, 32-bit quantities)
- jump tables
- . . .

CodeSurfer


compress95.c
File
Edit
Queries
Functions
Go
Window
Help
/*
 * Decompress stdin to stdout. This routine adapts to the codes in the
 * file building the "string" table on-the-fly; requiring no table to
 * be stored in the compressed file. The tables used herein are shared
 * with those of the compress() routine. See the definitions above.
 */
decompress() {
    register char_type *stackp;
    register int finchar;
    register code_int code, oldcode, incode;
    /*
     * As above, initialize the first 256 entries in the table.
     */
    maxcode = MAXCODE(n_bits = INIT_BITS);
    for (code = 255; code >= 0; code--) {
        tab_prefixof(code) = 0;
        tab_suffixof(code) = (char_type)code;
    }
    free_ent = ((block_compress) ? FIRST : 256);

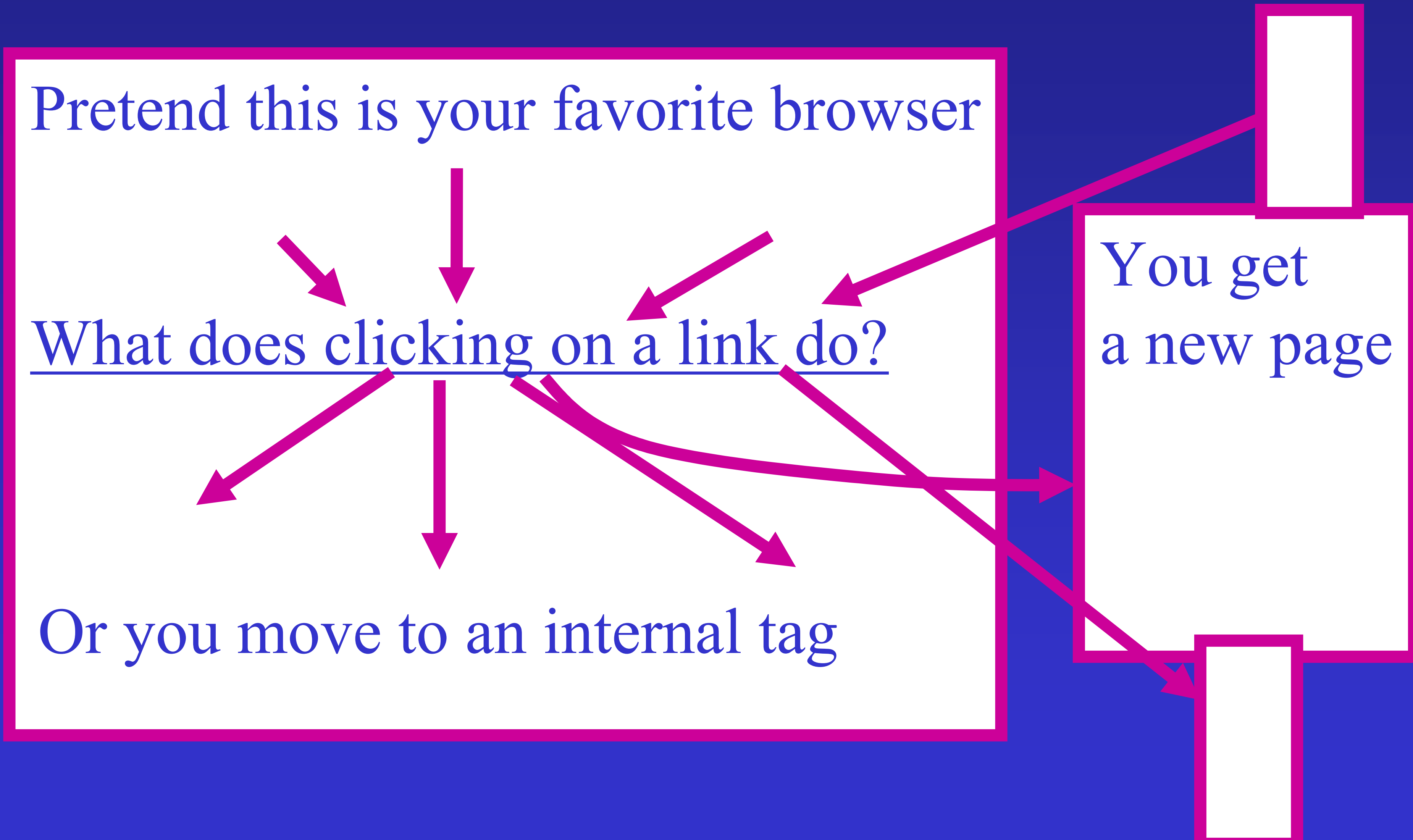
compress95.c

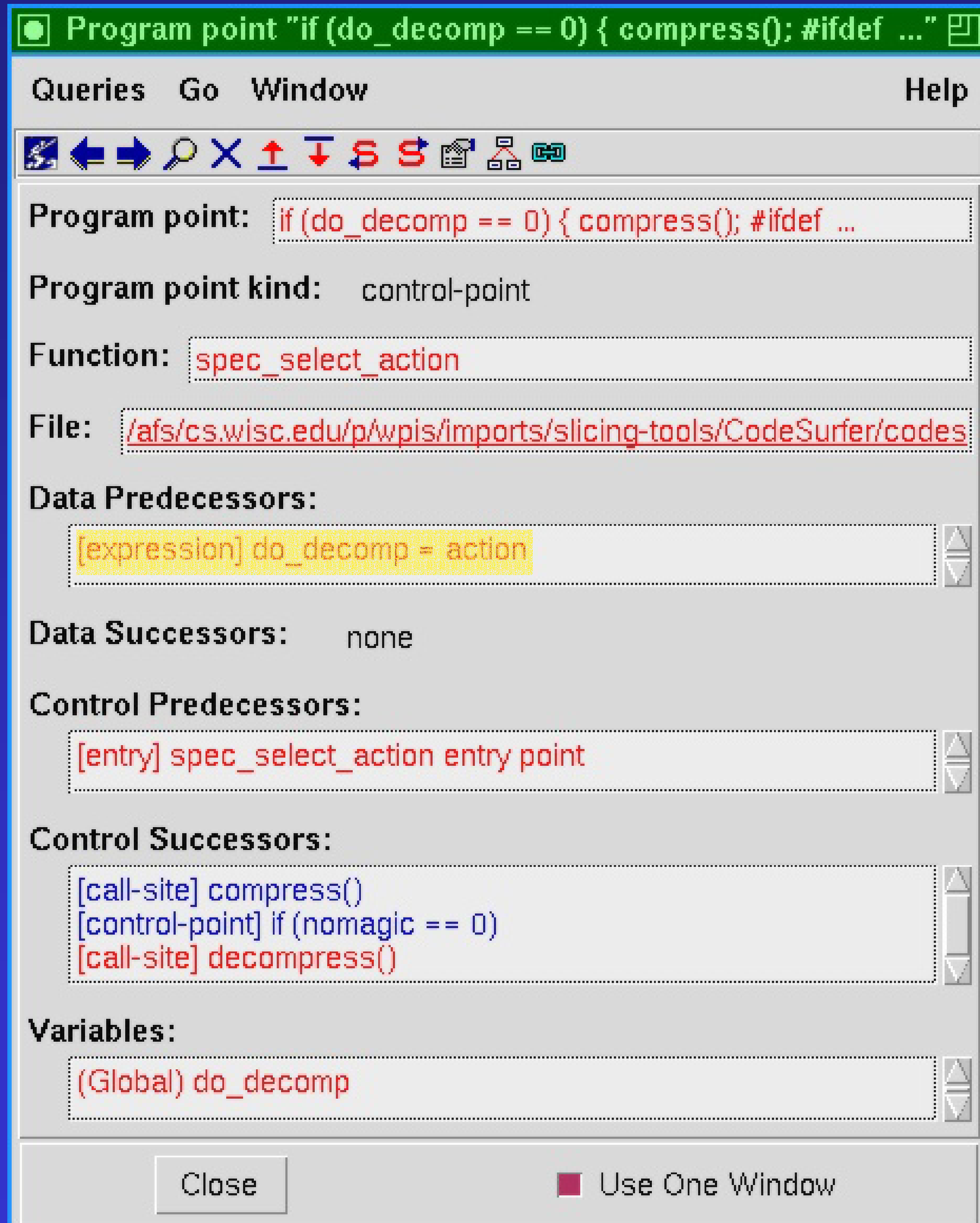
File Edit Queries Functions Go Window Help

```
    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
```

Browsing a Dependence Graph




Program point "if (do_decomp == 0) { compress(); #ifdef ..."
Queries
Go
Window
Help
Program point:
if (do_decomp == 0) { compress(); #ifdef ...
Program point kind:
control-point
Function:
spec_select_action
File:
/afs/cs.wisc.edu/p/wpis/imports/slicing-tools/CodeSurfer/codes
Data Predecessors:
[expression] do_decomp = action
Data Successors:
none
Control Predecessors:
[entry] spec_select_action entry point
Control Successors:
[call-site] compress()
[control-point] if (nomagic == 0)
[call-site] decompress()
Variables:
(Global) do_decomp
Close
Use One Window

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
```

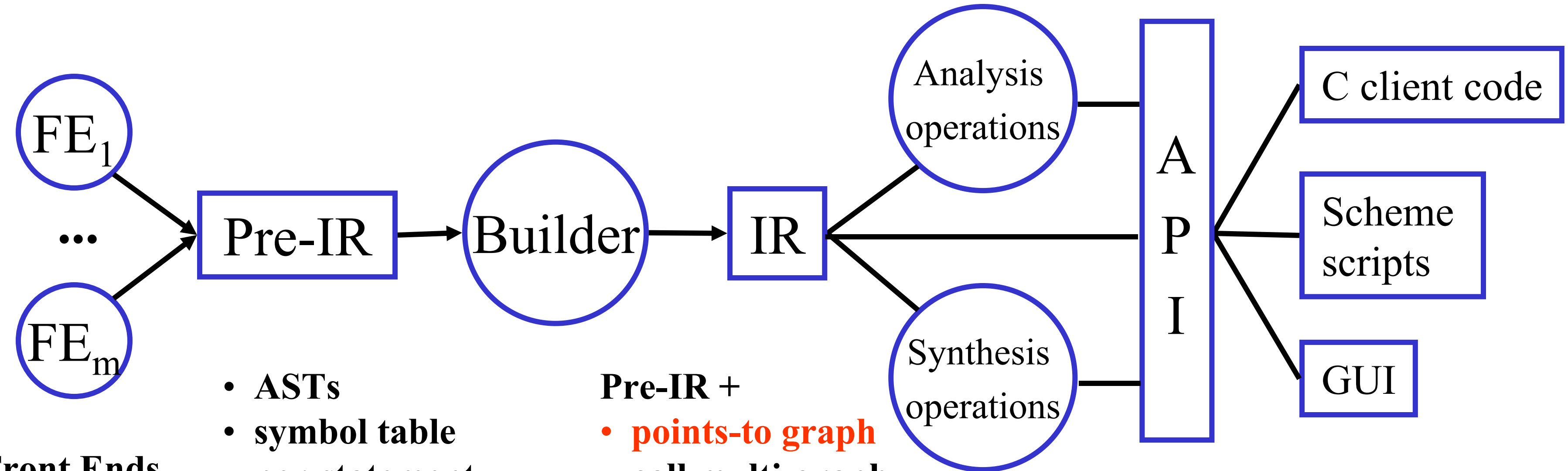
harness.c

File Edit Queries Functions Go Window Help

```
int main(int argc, char *argv[])

{
int count, i, oper;
int comp_count, new_count;
char    start_char;
int N;
char C;

        printf("SPEC 129.compress harness\n");
        scanf("%i       %c      %i", &count, &start_char, &seedi);
        printf("Initial File Size:%i    Start character:%c\n", count, start_char
        fill_text_buffer(count, start_char, orig_text_buffer);
        for (i = 1; i <= 25; i++)
        {
          new_count=add_line(orig_text_buffer, count, i, start_char);
          count=new_count;
          oper=COMPRESS;
          printf("The starting size is: %d\n", count);
          comp_count=spec_select_action(orig_text_buffer, count, oper, comp_text
          printf("The compressed size is: %d\n", comp_count);
          oper=UNCOMPRESS;
          new_count=spec_select_action(comp_text_buffer, comp_count, oper, new_t
          printf("The compressed/uncompressed size is: %d\n", new_count);
          compare_buffer(orig_text_buffer, count, new_text_buffer, new_count);
        }
```



Front Ends

EDG

- **ANSI C**
- **C++**

IdaPro

- **x86 binary code**

Java

UML (Rose/RT)

Verilog

VHDL

Jovial

- **ASTs**
- **symbol table**
- **per-statement variable usage**
- **pointer uses**
- **CFGs**

Pre-IR +

- **points-to graph**
- **call multi-graph**
- **GMOD / GREF**
- **program-wide variable usage**
- **PDGs / SDG**

- **Precise interprocedural**
  - **predecessors and successors**
  - **slicing and chopping**
- **Boolean operations on sets of PDG nodes**
- **model checking**
- **constraint generation**

Other infrastructure: **command-line, preprocessor, include-file instances, library, and loader support**

Demos

wc

x86

sum

information flow

hello

script example