Program Representations

Xiangyu Zhang

Why Program Representations

- Initial representations
  - Source code (across languages).
  - Binaries (across machines and platforms).
  - Source code / binaries + test cases.
- They are hard for machines to analyze.

Program Representations

- Static program representations
  - Abstract syntax tree;
  - Control flow graph;
  - Program dependence graph;
  - Call graph;
  - Points-to relations.
- Dynamic program representations
  - Control flow trace, address trace and value trace;
  - Dynamic dependence graph;
  - Whole execution trace;

(1) Abstract syntax tree

- An abstract syntax tree (AST) is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the operators.

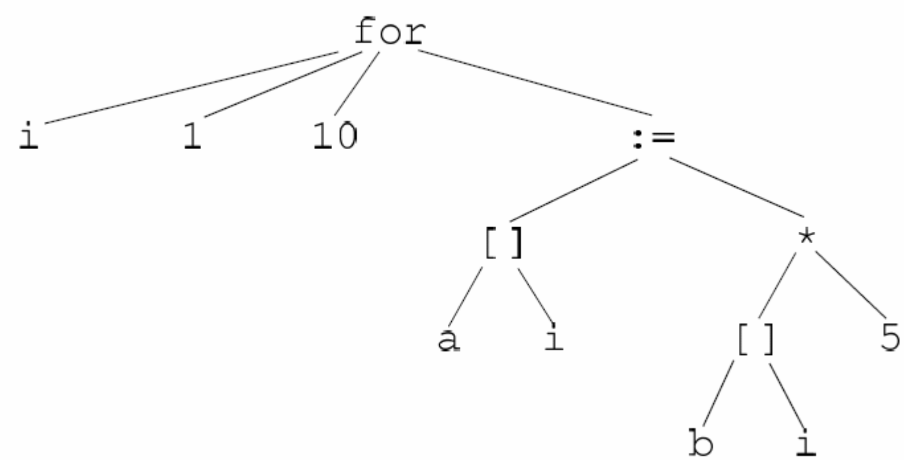
Source:

~~for i := 1 to 10 do~~

~~a[i] := b[i] * 5;~~

end

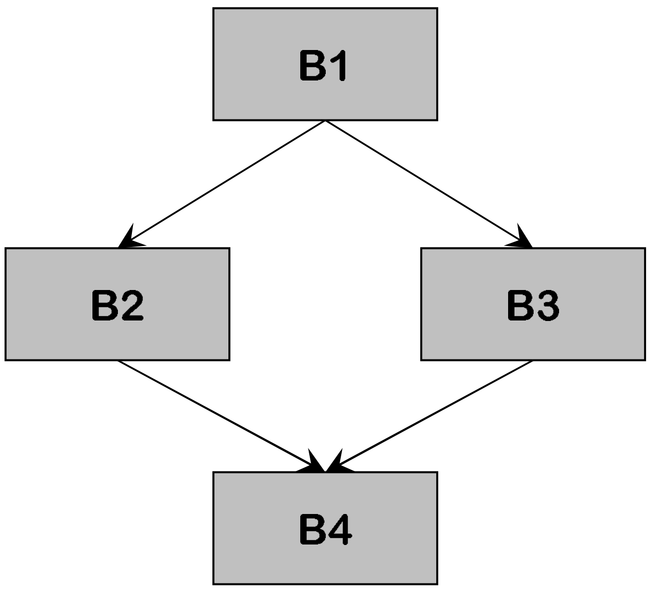
AST:



Program chipping.

(2) Control Flow Graph (CFG)

- Consists of basic blocks and edges
  - A maximal sequence of consecutive instructions such that inside the basic block an execution can only proceed from one instruction to the next (SESE).
  - Edges represent potential flow of control between BBs.
  - Program path.

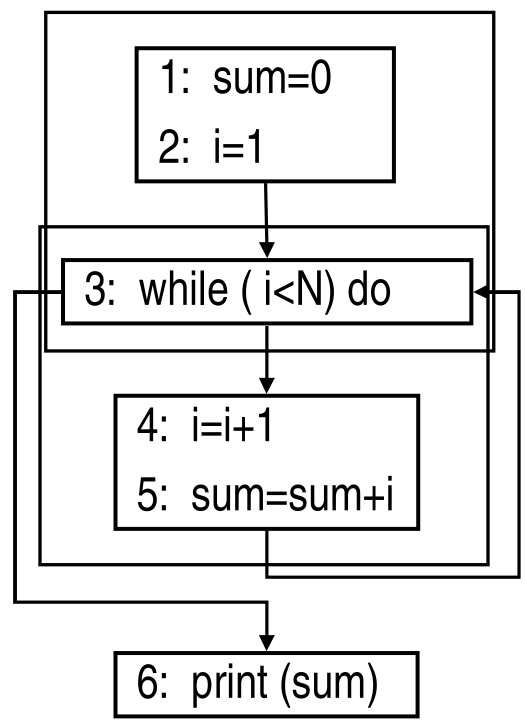


- CFG = <V, E, Entry, Exit>
- V = Vertices, nodes (BBs)
- E = Edges, potential flow of control $E \subseteq V \times V$
- Entry, Exit $\in V$, unique entry and exit

(2) An Example of CFG

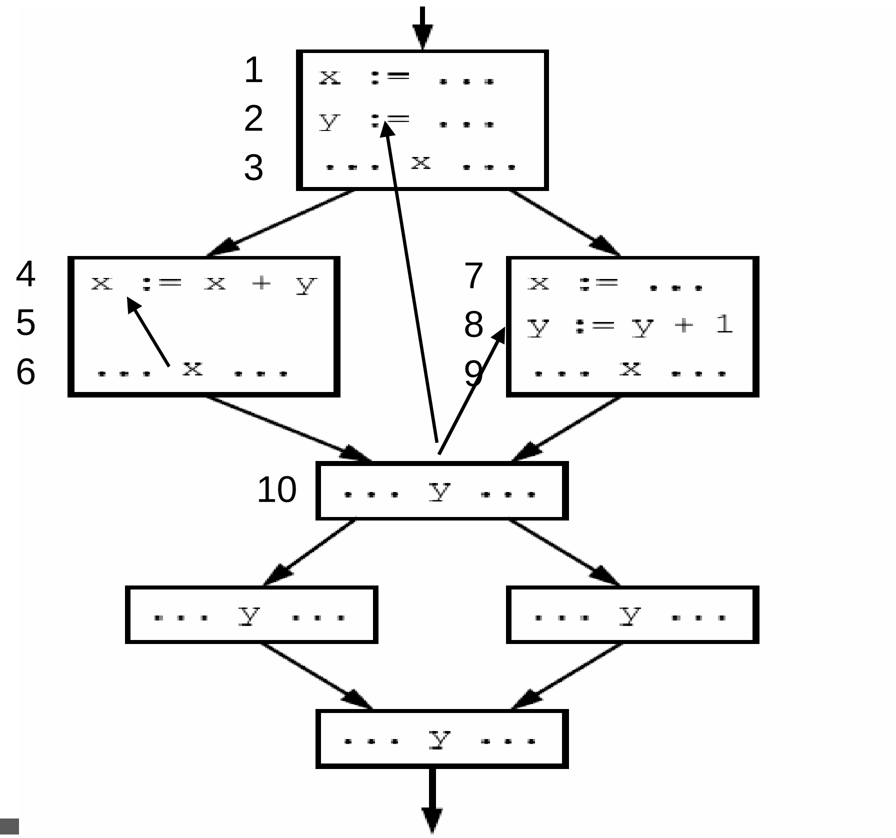
- BB- A maximal sequence of consecutive instructions such that inside the basic block an execution can only proceed from one instruction to the next (SESE).

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:            i=i+1
5:            sum=sum+i
      endwhile
6:   print(sum)
```



(3) Program Dependence Graph (PDG)– Data Dependence

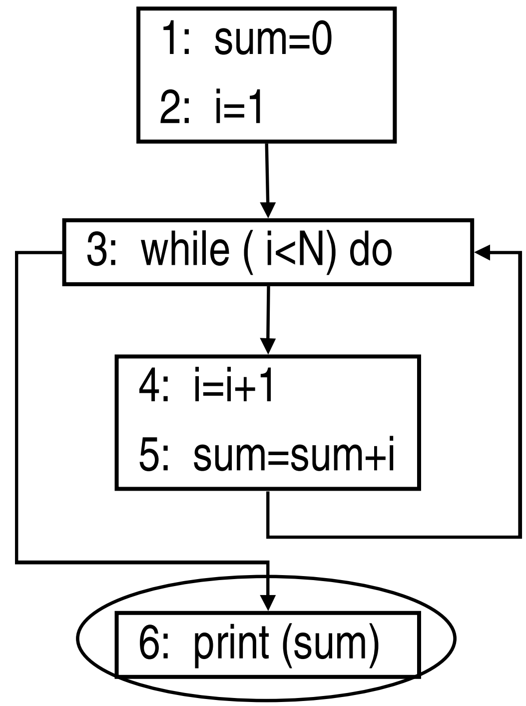
- S data depends T if there exists a control flow path from T to S and a variable is defined at T and then used at S.



(3) PDG – Control Dependence

- X dominates Y if every possible program path from the entry to Y has to pass X.
  - Strict dominance, dominator, immediate dominator.

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:            i=i+1
5:            sum=sum+i
      endwhile
6:   print(sum)
```

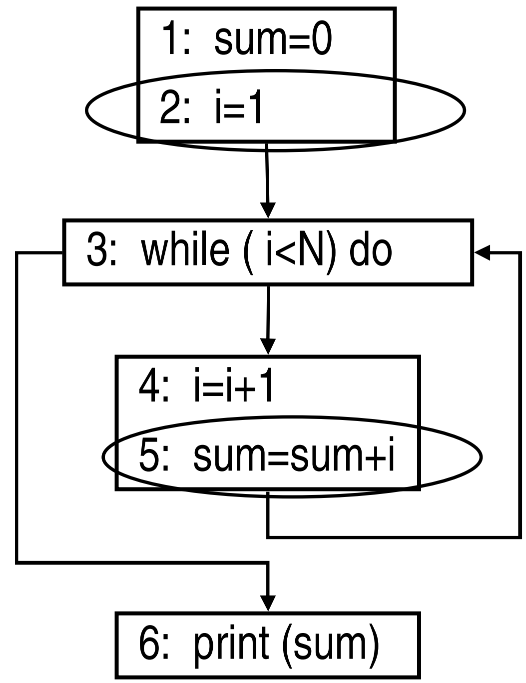


DOM(6)={1,2,3,6}   IDOM(6)=3

(3) PDG – Control Dependence

- X post-dominates Y if every possible program path from Y to EXIT has to pass X.
  - Strict post-dominance, post-dominator, immediate post-dominance.

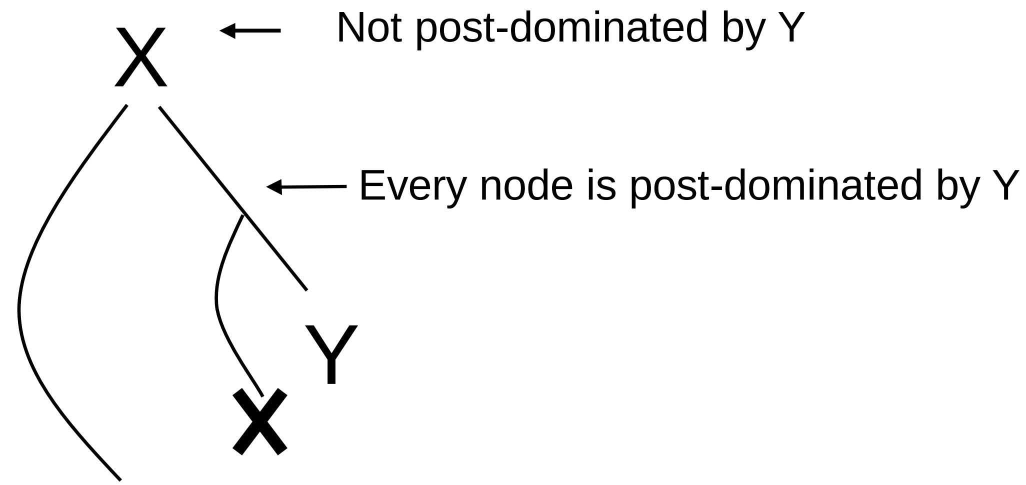
```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:             i=i+1
5:             sum=sum+i
      endwhile
6:   print(sum)
```



PDOM(5)={3,5,6}   IPDOM(5)=3

(3) PDG – Control Dependence

- Intuitively, Y is control-dependent on X iff X directly determines whether Y executes (statements inside one branch of a predicate are usually control dependent on the predicate)
  - there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y
  - X is not strictly post-dominated by Y



(3) PDG – Control Dependence

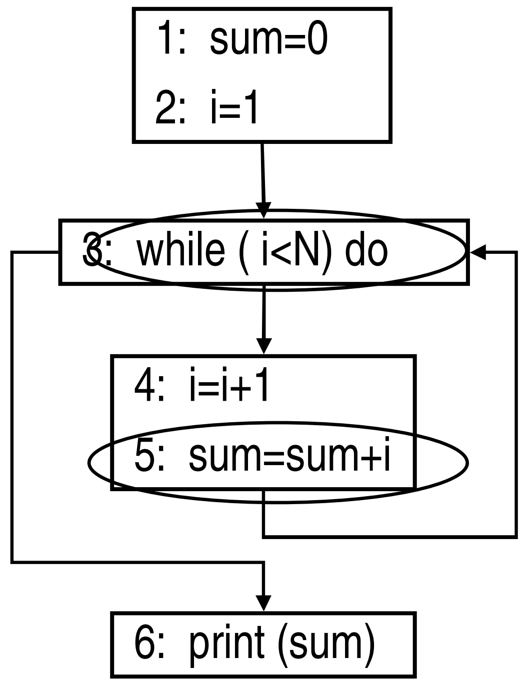
A node (basic block) Y is control-dependent on another X iff X directly determines whether Y executes

- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y
- X is not strictly post-dominated by Y

```
1:   sum=0
2:   i=1
3:   while ( i<N) do
4:        i=i+1
5:        sum=sum+i
     endwhile
6:  print(sum)
```

CD(5)=3

CD(3)=3, tricky!

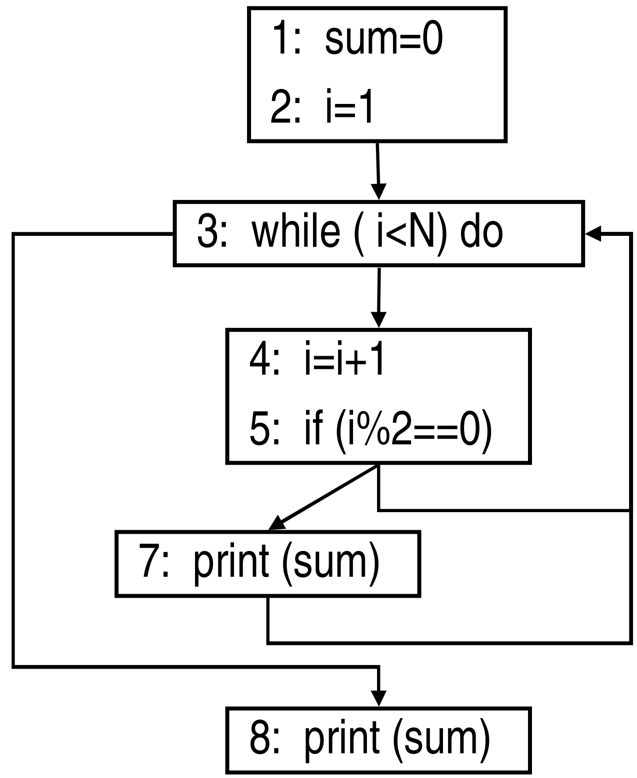


(3) PDG – Control Dependence is not Syntactically Explicit

A node (basic block) Y is control-dependent on another X iff X directly determines whether Y executes

- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y
- X is not strictly post-dominated by Y

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:            i=i+1
5:            if (i%2==0)
6:                 continue;
7:            sum=sum+i
      endwhile
8:   print(sum)
```



(3) PDG – Control Dependence is Tricky!

A node (basic block) Y is control-dependent on another X iff X directly determines whether Y executes

- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y
- X is not strictly post-dominated by Y

- ❑ Can a statement control depends on two predicates?

(3) PDG – Control Dependence is Tricky!

A node (basic block) Y is control-dependent on another X iff X directly determines whether Y executes

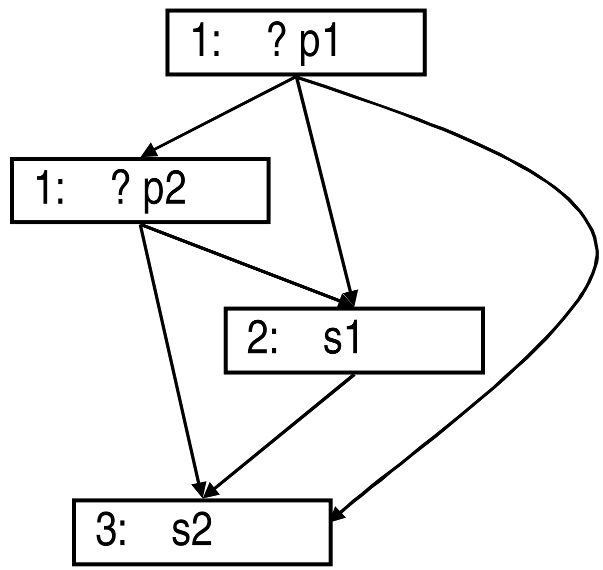
- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y
- X is not strictly post-dominated by Y

- Can one statement control depends on two predicates?

```
1:   if ( p1 || p2 )
2:          s1;
3:   s2;
```

What if ?

```
1:   if ( p1 && p2 )
2:          s1;
3:   s2;
```



Interprocedural CD, CD in case of exception,…

(3) PDG

- A program dependence graph consists of control dependence graph and data dependence graph
- Why it is so important to software reliability?
  - In debugging, what could possibly induce the failure?
  - In security

```
p=getpassword( );
…
if (p==“zhang”) {
    send (m);
}
```

(4) Points-to Graph

- Aliases: two expressions that denote the same memory location.
- Aliases are introduced by:
  - pointers
  - call-by-reference
  - array indexing
  - C unions

(4) Points-to Graph

- Aliases: two expressions that denote the same memory location.
- Aliases are introduced by:
  - pointers
  - call-by-reference
  - array indexing
  - C unions

(4) Why Do We Need Points-to Graphs

- Debugging

```
x.lock();
...
y.unlock(); // same object as x?
```

- Security

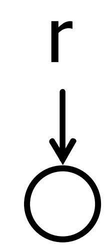
```
F(x,y)
{
   x.f=password;
   …
   print (y.f);
}

F(a,a); disaster!
```

(4) Points-to Graph

- Points-to Graph
  - at a program point, compute a set of pairs of the form p -> x, where p MAY/MUST points to x.

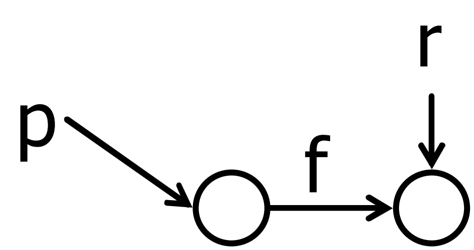
```
m(p) {
→ r = new C();
  p->f = r;
  t = new C();
  if (...)
     q=p;
  r->f = t;
}
```



(4) Points-to Graph

- Points-to Graph
  - at a program point, compute a set of pairs of the form p->x, where p MAY/MUST points to x.

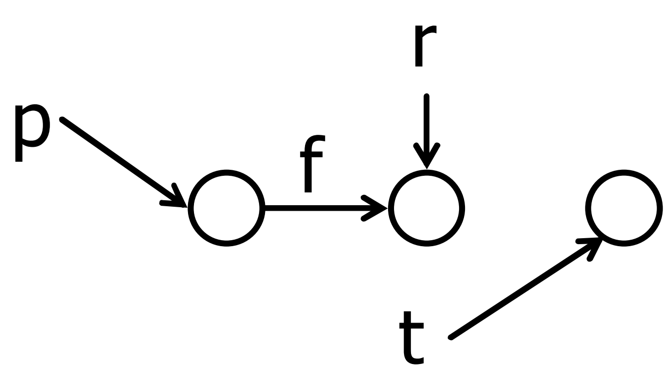
```
m(p) {
    r = new C();
→   p->f = r;
    t = new C();
    if (...)
        q=p;
    r->f = t;
}
```



(4) Points-to Graph

- Points-to Graph
  - at a program point, compute a set of pairs of the form p->x, where p MAY/MUST points to x.

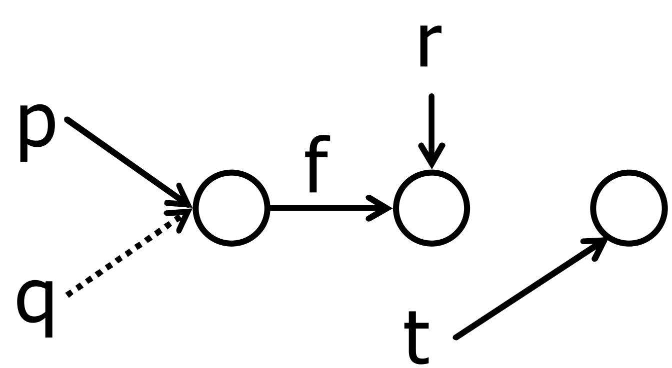
```
m(p) {
    r = new C();
    p->f = r;
→   t = new C();
    if (...)
        q=p;
    r->f = t;
}
```



(4) Points-to Graph

- Points-to Graph
  - at a program point, compute a set of pairs of the form p->x, where p MAY/MUST points to x.

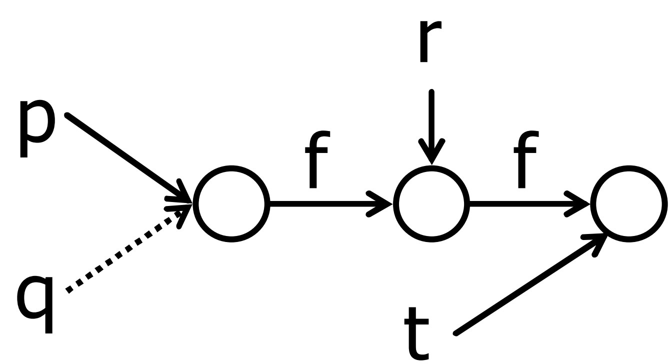
```
m(p) {
    r = new C();
    p->f = r;
    t = new C();
    if (...)
➡       q=p;
    r->f = t;
}
```



(4) Points-to Graph

- Points-to Graph
  - at a program point, compute a set of pairs of the form p->x, where p MAY/MUST points to x.

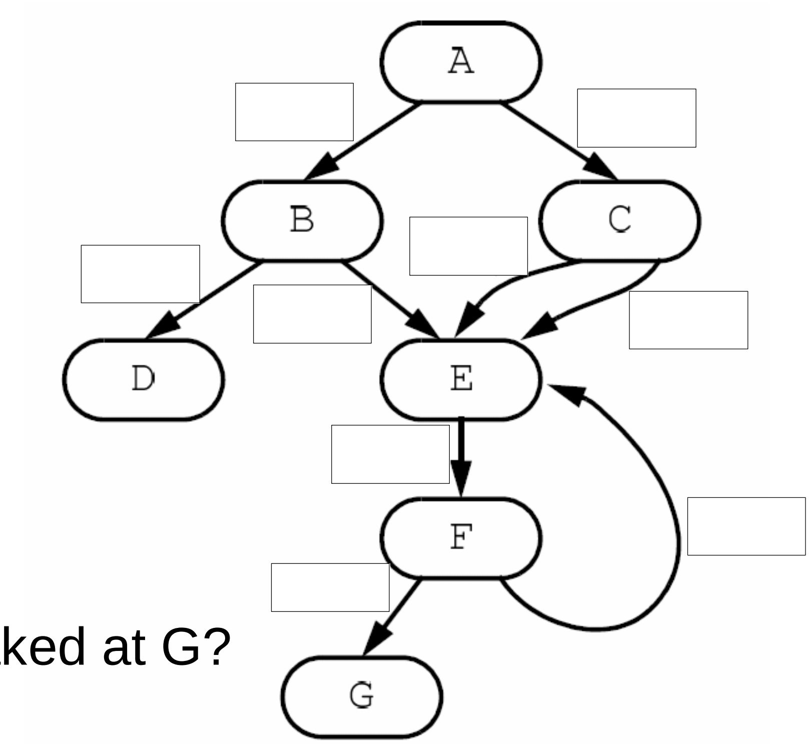
```
m(p) {
    r = new C();
    p->f = r;
    t = new C();
    if (...)
        q=p;
➡   r->f = t;
}
```



p->f->f and t are aliases

(5) Call Graph

- Call graph
  - nodes are procedures
  - edges are calls
- Hard cases for building call graph
  - calls through function pointers



Can the password acquired at A be leaked at G?

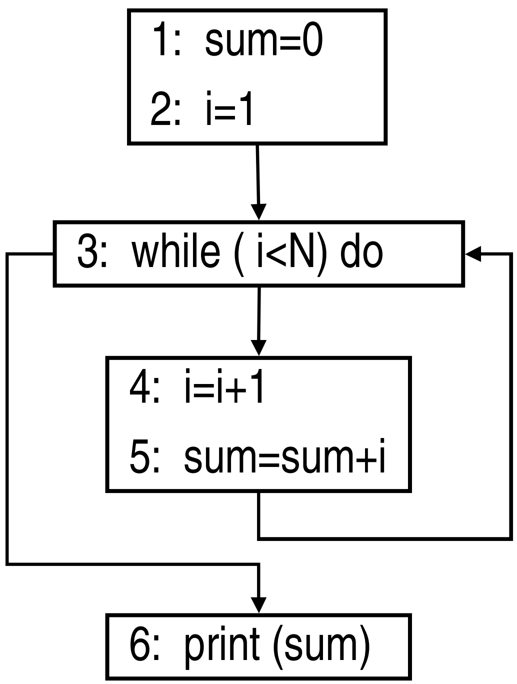
How to acquire and use these representations?

- Will be covered by later lectures.

Program Representations

- Static program representations
  - ✓ Abstract syntax tree;
  - ✓ Control flow graph;
  - ✓ Program dependence graph;
  - ✓ Call graph;
  - ✓ Points-to relations.
- Dynamic program representations
  - Control flow trace;
  - Address trace, Value trace;
  - Dynamic dependence graph;
  - Whole execution trace;

(1) Control Flow Trace



N=2:

1_1: sum=0

2_1: i=1

3_1: while (i<N) do

4_1: i=i+1

5_1: sum=sum+i

3_2: while (i<N) do

4_2: i=i+1

5_2: sum=sum+i

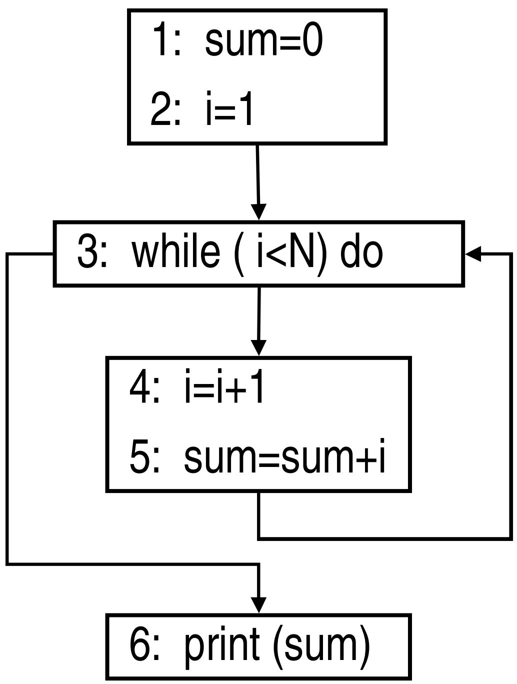
3_3: while (i<N) do

6_1: print (sum)

<… x_i, …> x is a program point, x_i is an execution point

<… 8048057_{37}, $804805a_{29}$, …>

(1) Control Flow Trace



N=2:

1_1: sum=0
i=1

3_1: while (i<N) do

4_1: i=i+1
sum=sum+i

3_2: while (i<N) do

4_2: i=i+1
sum=sum+i

3_3: while (i<N) do

6_1: print (sum)

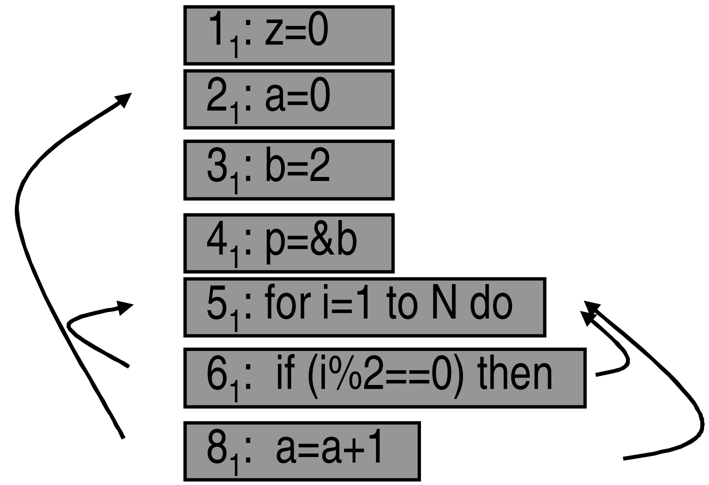
$1_1\ 2_1\ 3_1\ 4_1\ 5_1\ 3_2\ 4_2\ 5_2\ 3_3\ 6_1 \longrightarrow 1_1\ 3_1\ 4_1\ 3_2\ 4_2\ 3_3\ 6_1$

A More Compact CFT: < T, T, F >

(2) Dynamic Dependence Graph (DDG)

Input: N=2

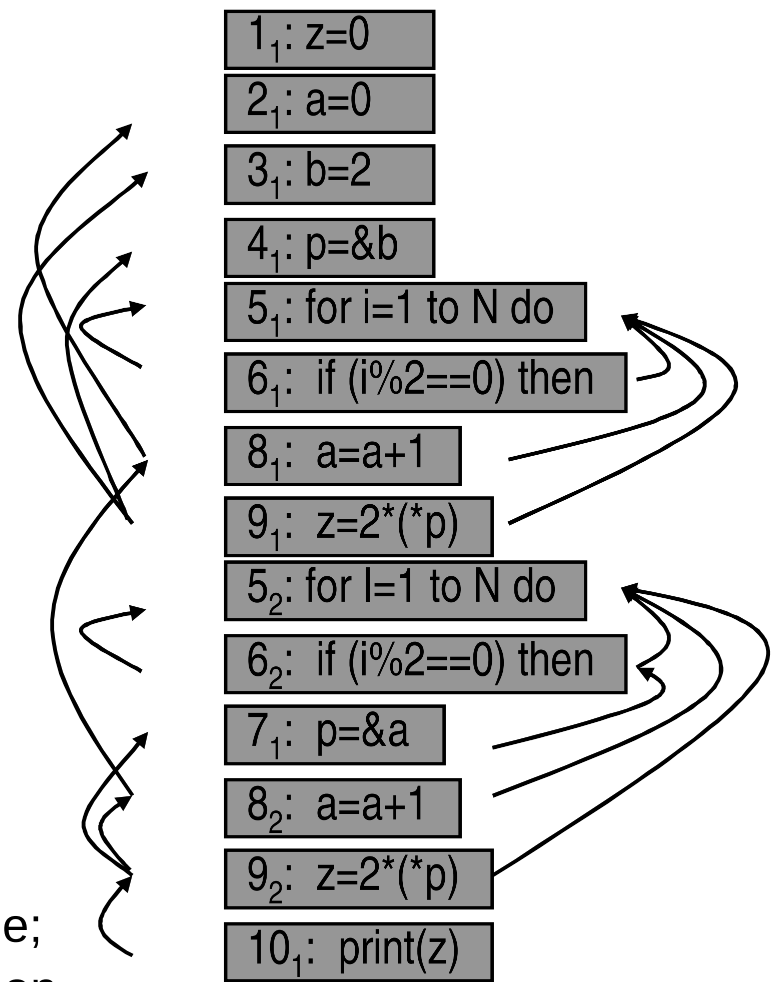
```
1:    z=0
2:    a=0
3:    b=2
4:    p=&b
5:    for i = 1 to N do
6:       if ( i %2 == 0) then
7:             p=&a
         endif
      endfor
8:    a=a+1
9:    z=2*(*p)
10:   print(z)
```



(2) Dynamic Dependence Graph (DDG)

Input: N=2

```
1:      z=0
2:      a=0
3:      b=2
4:      p=&b
5:      for i = 1 to N do
6:        if ( i %2 == 0) then
7:                p=&a
          endif
        endfor
8:     a=a+1
9:     z=2*(*p)
10:    print(z)
```

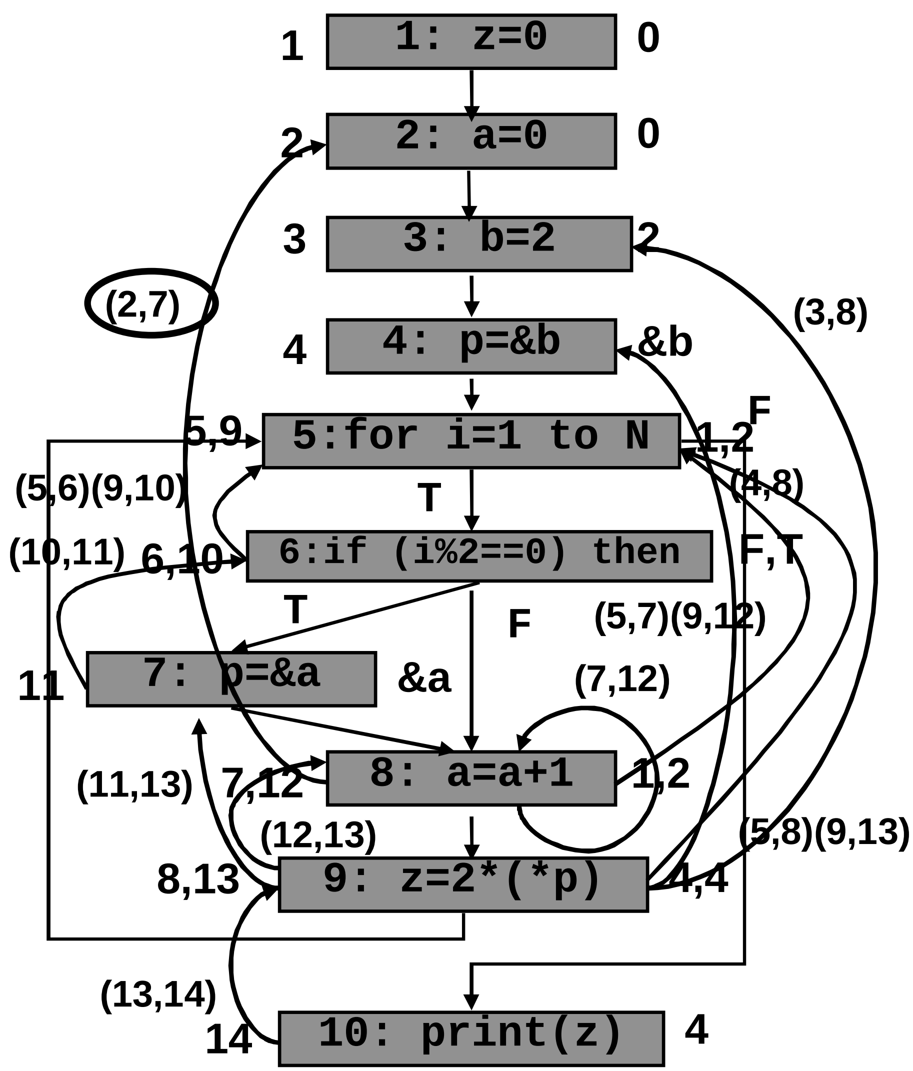


One use has only one definition at runtime;

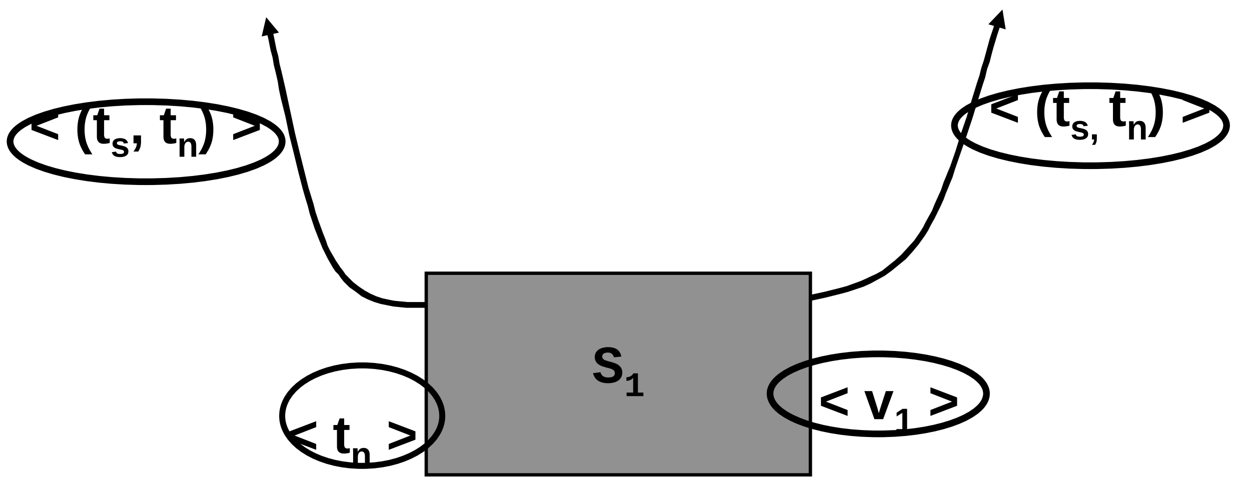
One statement instance control depends on only one predicate instance.

(3) Whole Execution Trace

T	Input: N=2
1	1_1: z=0
2	2_1: a=0
3	3_1: b=2
4	4_1: p=&b
5	5_1: for i = 1 to N do
6	6_1: if (i %2 == 0) then
7	8_1: a=a+1
8	9_1: z=2*(*p)
9	5_2: for i = 1 to N do
10	6_2: if (i %2 == 0) then
11	7_1: p=&a
12	8_2: a=a+1
13	9_2: z=2*(*p)
14	10_1: print(z)



(3) Whole Execution Trace



Multiple streams of numbers.

Program Representations

- Static program representations
  - Abstract syntax tree;
  - Control flow graph;
  - Program dependence graph;
  - Call graph;
  - Points-to relations.
- Dynamic program representations
  - Control flow trace, address trace and value trace;
  - Dynamic dependence graph;
  - Whole execution trace;

Next Lecture – Program Analysis

- Static analysis
- Dynamic analysis