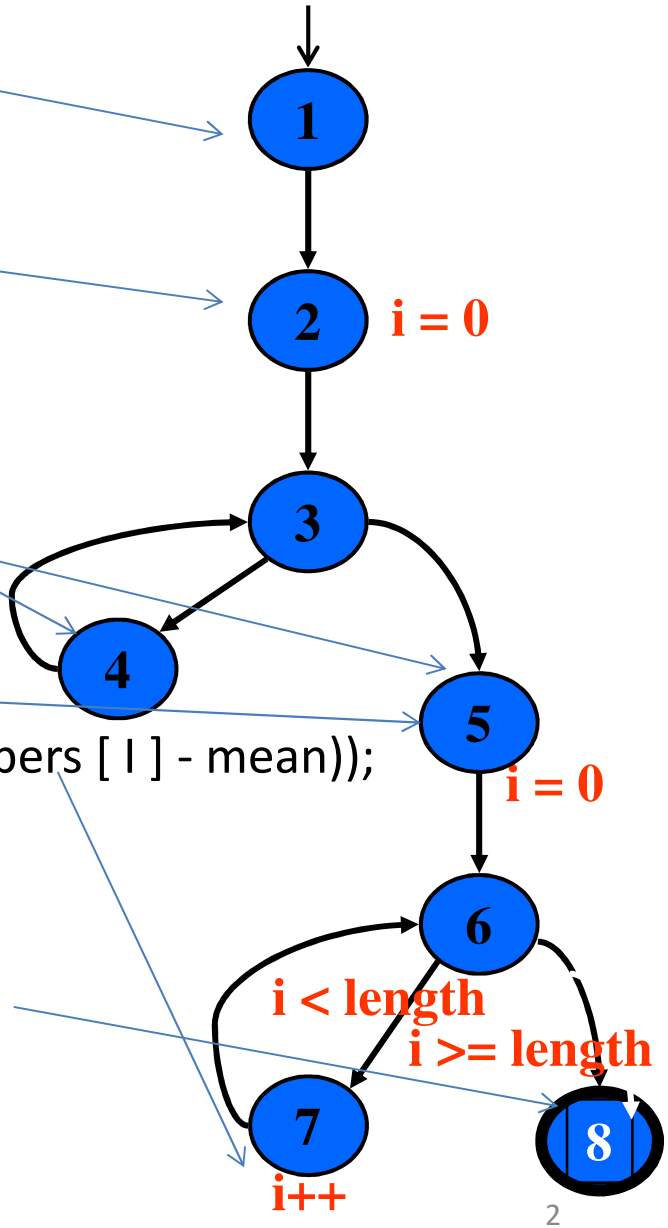
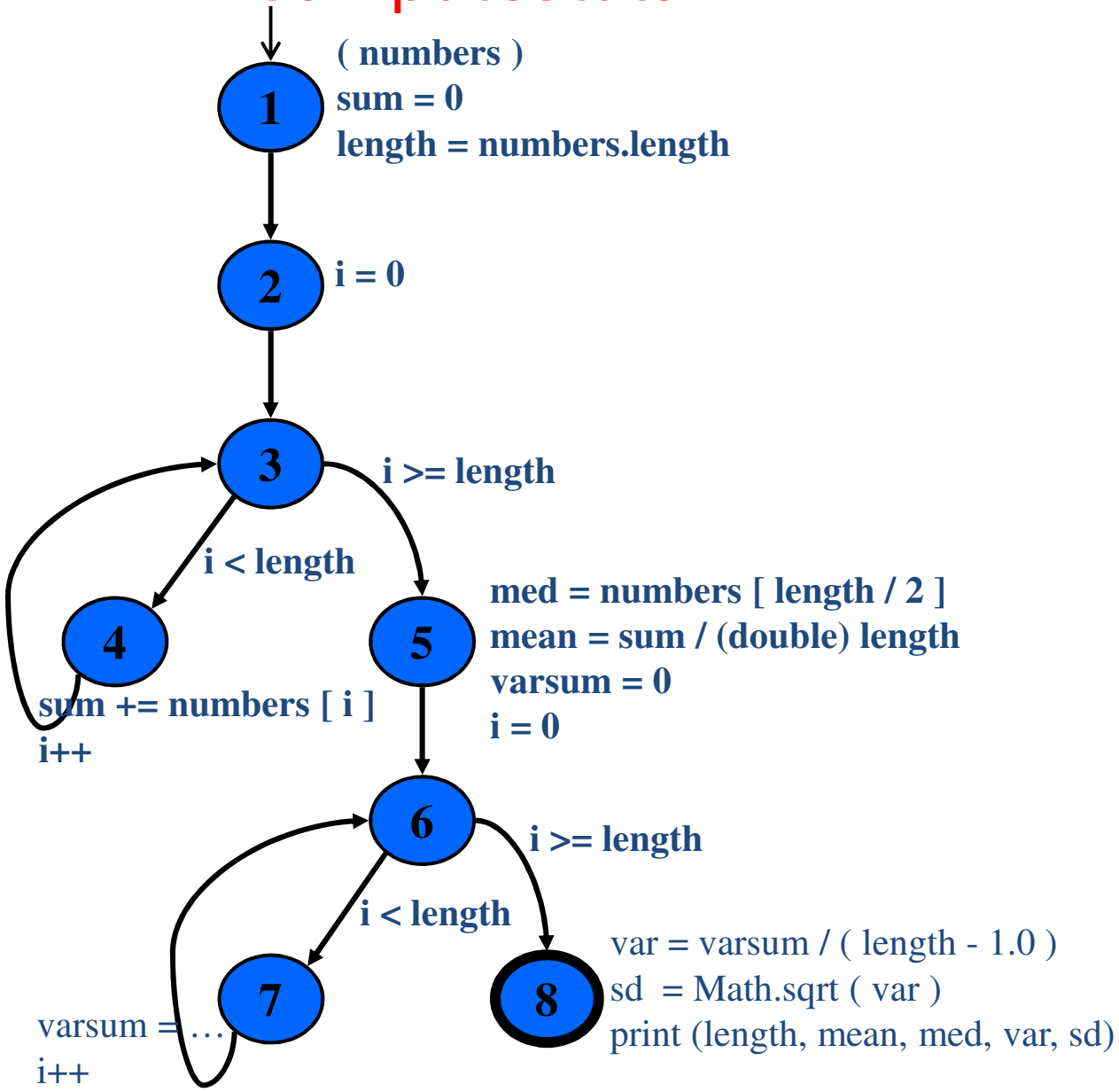
Graph Coverage for Source Code

Data Flow Graph Coverage for Source Code

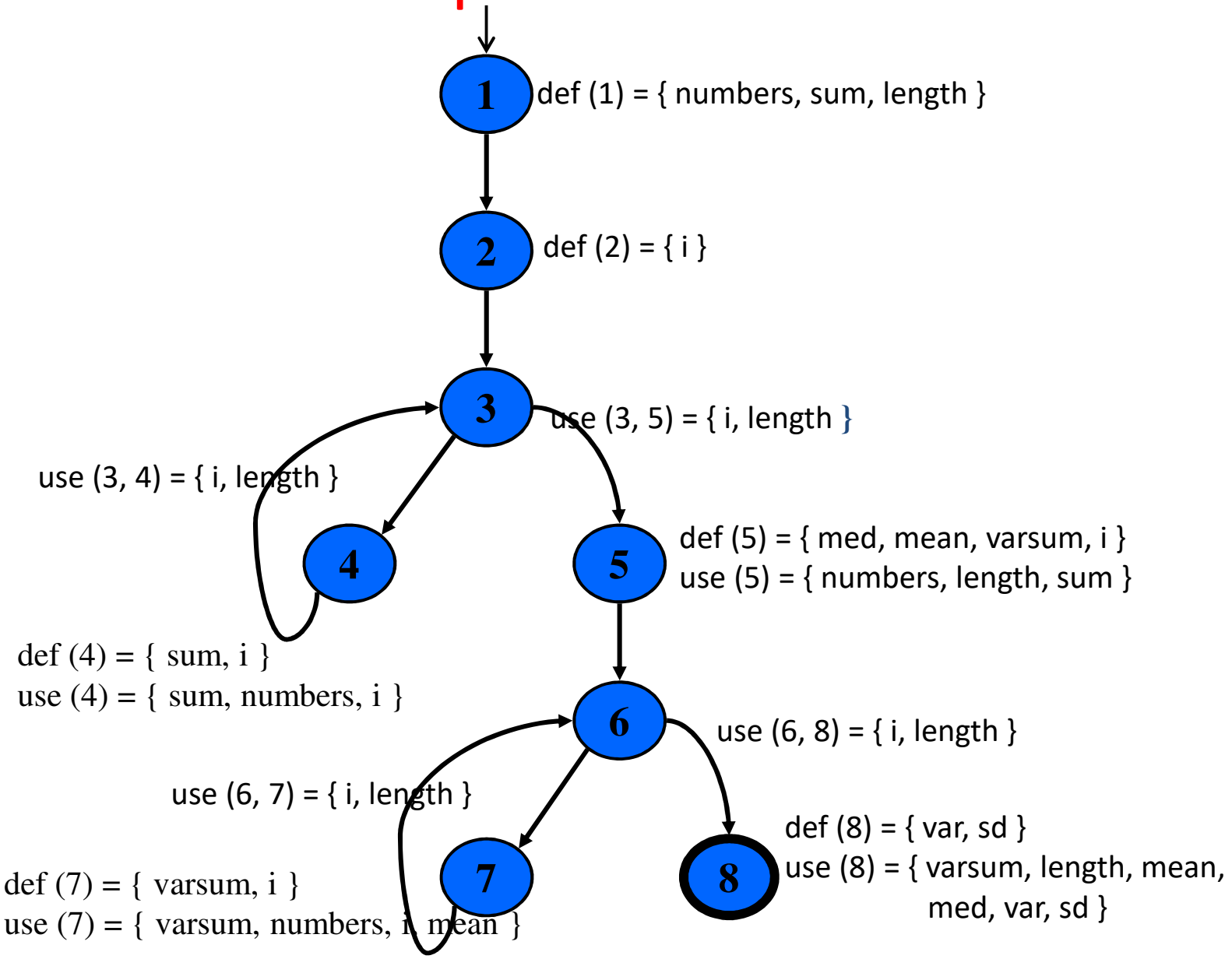
```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;
    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
   varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );
     System.out.println ("length:                " + length);
     System.out.println ("mean:                  " + mean);
     System.out.println ("median:                " + med);
     System.out.println ("variance:              " + var);
     System.out.println ("standard deviation: " + sd);     }
```



Control Flow Graph for the function ComputeStats



CFG for ComputeStats – With Defs & Uses



Defs and Uses Tables for ComputeStat

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

Graph Coverage for Design Elements

- Use of data abstraction and object oriented software has increased importance on modularity and reuse.
- Therefore testing of software based design (design elements) is more important that past
  - They are usually associated with integration testing.
- Since modularity, the software components can be tested independently.

Structural Graph Coverage for Desgin Elements

- Graph coverage for design elements usually starts

by creating graphs that are based on couplings between software components.

- *Coupling* measures the dependency relations

between two units by reflecting their interconnections

Data-Bound Relationships Between Design Elements

- **Caller:** unit that invokes the **callee**
- An **actual parameter** *(*value passed to the callee by the caller;) is in the caller
  - Its value is assigned to a **formal parameter** in the callee.

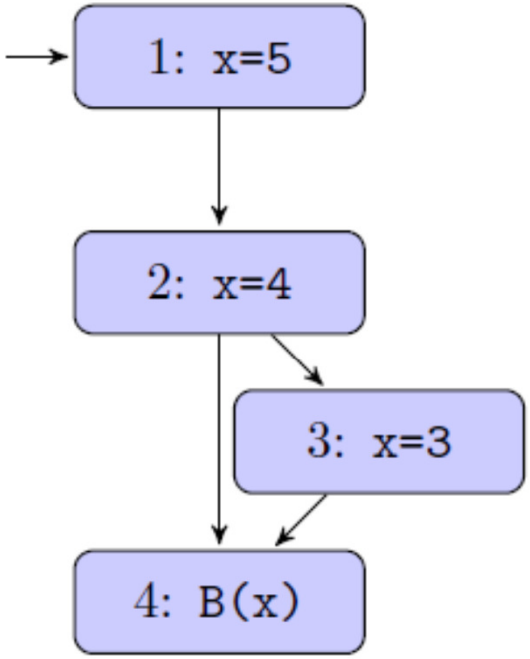
caller:

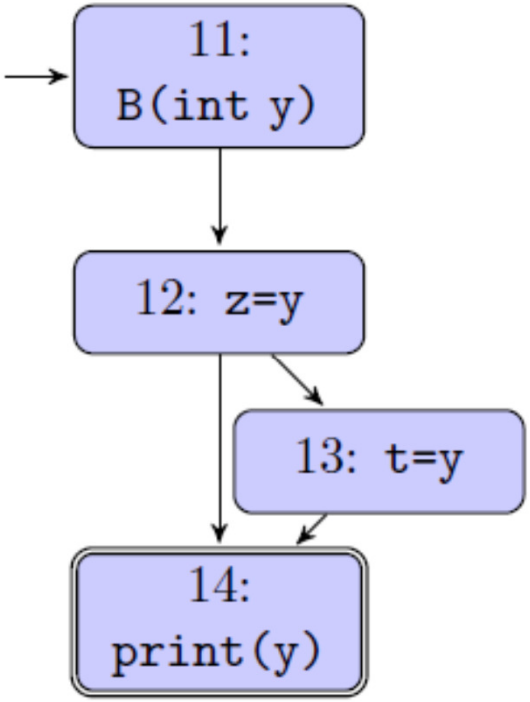
```
...
foo(actual1, actual2);
...
```

callee:

```
void foo(int formal1, int formal2) {
  ...
}
```

Define du-pairs between Callers and Callees





The last-defs are 2, 3

The first-uses are 12, 13

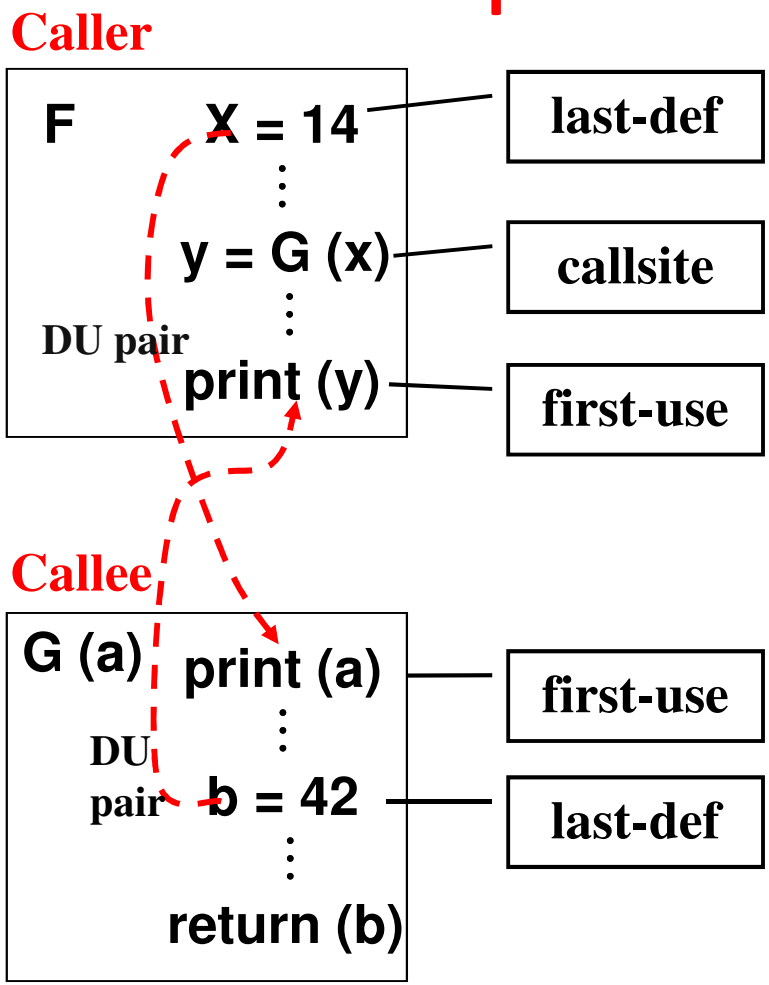
Inter-procedural DU pairs
last-def & first-use

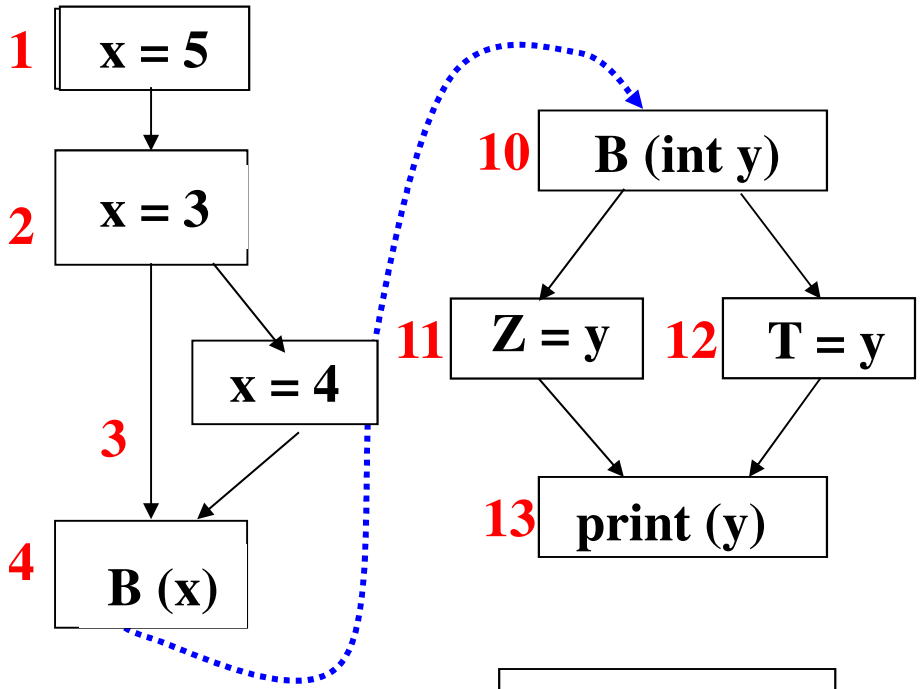
- The last-def is the definition that goes through the call or return
  - Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value
- The first-use picks up that definition.

```
x = 14;    // last-def
y = g(x);
print(y); // first-use
```

```
int g(a) {
  print(a); // first-use
  b = 24;   // last-def
  return b;
}
```

Inter-procedural DU Pairs





Last Defs
2, 3

First Uses
11, 12

Coupling

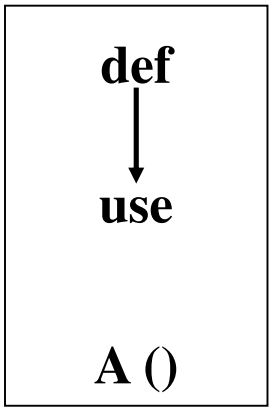
Three different ways that different program parts might interact:

Parameter Coupling: relationships between caller and callees; passing data as parameters;

Shared Data Coupling: one unit writes to some in-memory data, another unit reads this data;

External Data Coupling: one unit writes data e.g. to disk, another read reads the data.

def-use pairs
as Normal Coupling Data Flow Analysis I



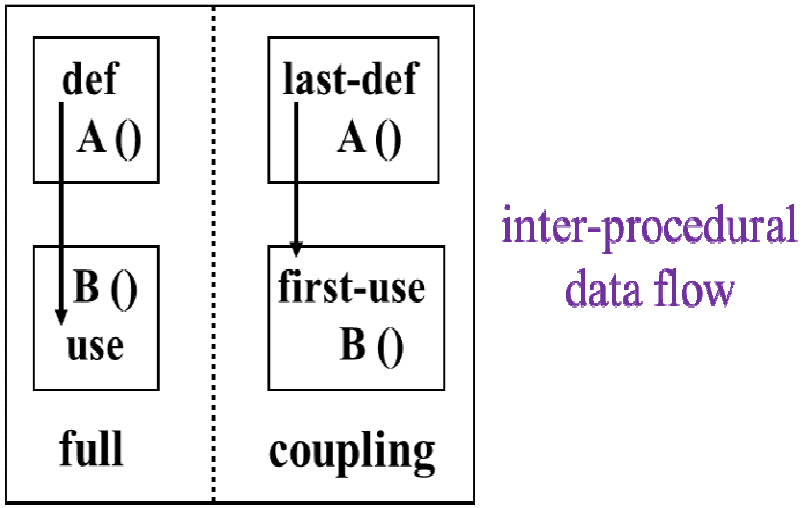
intra-procedural data flow
(within the same unit)

- The method A() contains def and use.
- Here the variable is omitted
- It is assumed that all du pairs refer to the same variable.

This is classic Data Flow Coverage for design elements

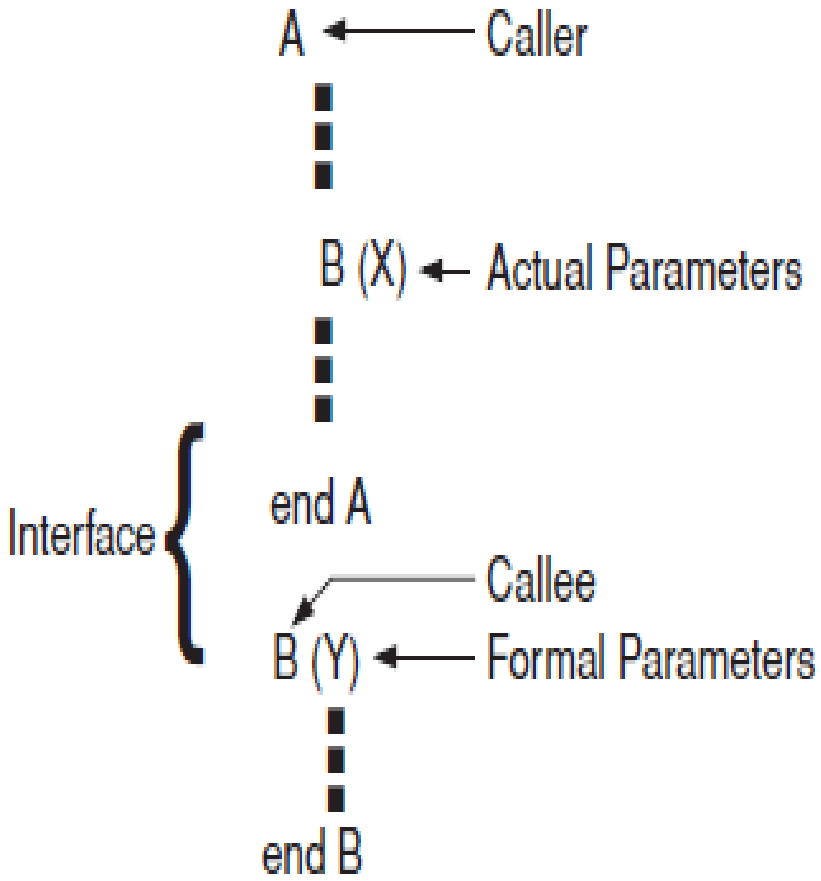
def-use pairs
as Normal Coupling Data Flow Analysis II

All du pairs between a caller A() and a callee B()

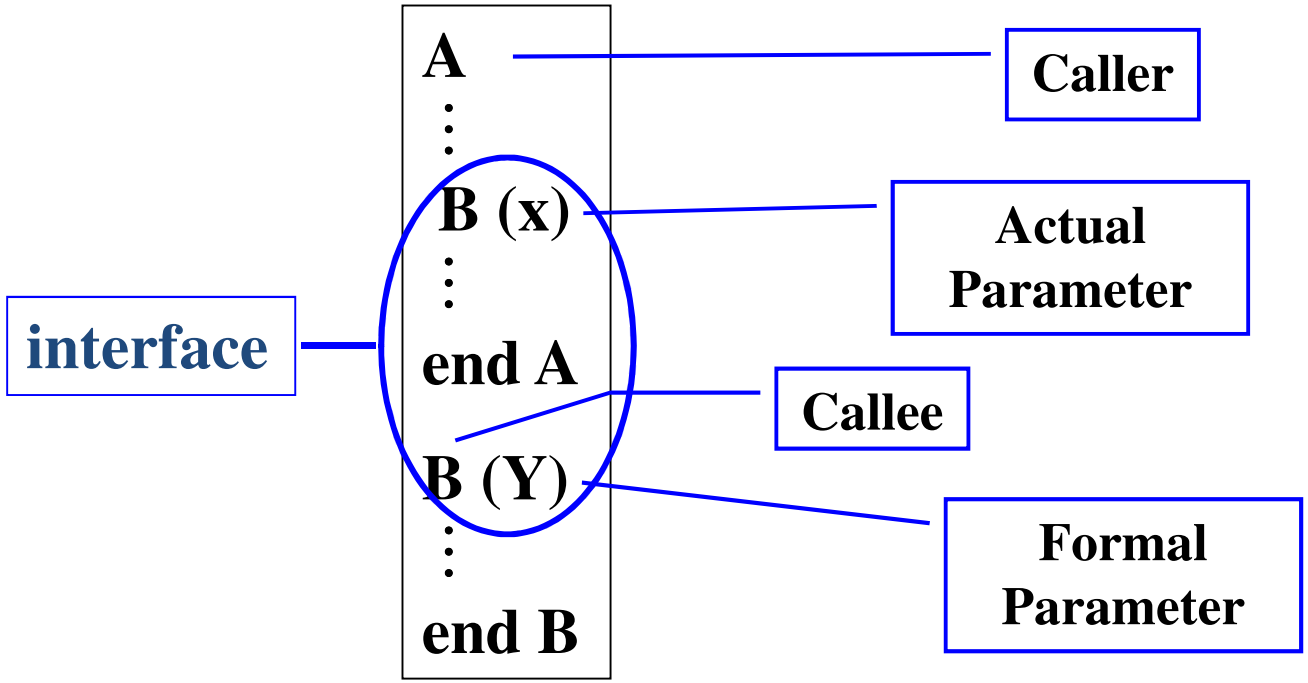


This example is another classic Data Flow Coverage for design elements

Parameter Coupling



Example Call Site



- Applying data flow criteria to def-use pairs between units is too expensive
- Too many possibilities
- But this is integration testing, and we really only care about the interface …

Coupling Data-flow

- We are only testing variables that have a definition on one side and a use on the otherside
- we do not impose test requirements in the following case:

```
foo(20, 17);

foo(int x, int y) {
  return x * 5;  // no use of y, so no test requirement
}
```

Java Example.

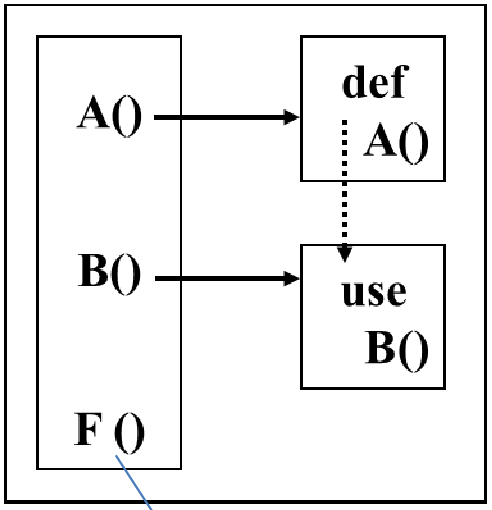
```
class M {
  int x;
  void foo() { x = 5; }
  int bar() { return x; }
}
```

```
void notCouplingMethod() {
  M m = new M(), n = new M();
  m.foo(); print (n.bar());
}
```

- We will generally treat an entire array as one variable.
- Because *m and n are different objects*, **there is no du-pair here.**
- We have a last- def of m.x and a first-use of n.x.

def-use pairs in Object-Oriented Software

In object oriented software du- pairs are usually based on class and state variables defined for class.



- A coupling method F() calls two methods A() and B().
  - A() defines a variable and B() uses it.
- For the variable references to the same, both A() and B() must be called through the same instance context, or object reference.
  - For example if the callers are o.A() and o.B(), they are called through the instance context of o.
- If the calls are not through the same instance context, the definition and use will be different instances of the variable.

def-use pairs in Object-Oriented Software

- In object oriented data flow testing

A() and B() could be in the same class , or they could be in different classes and accessing the same global variables

- These are the advanced topics for inheritance and polymorphisim.