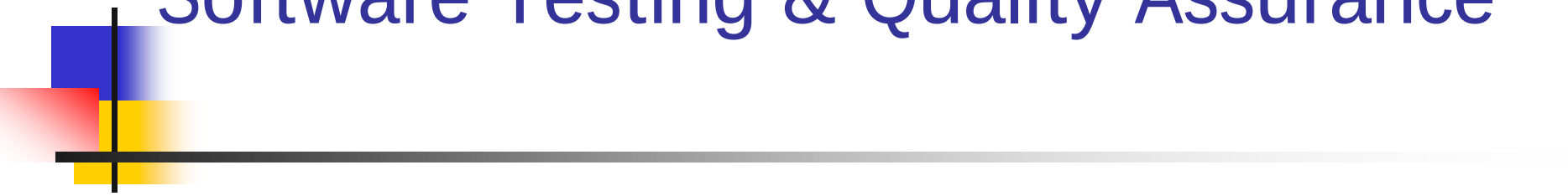


Software Testing & Quality Assurance

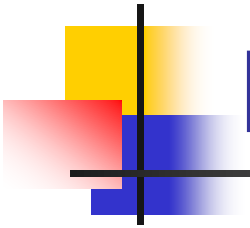
Data Flow Testing

- Define/Use Testing
- Slice-Based Testing
- Guidelines and Observations

Credits & Readings

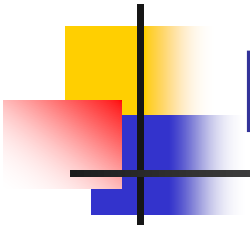
The material included in these slides are mostly adopted from the following books:

- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, *"Testing Computer Software"* Wiley (see also http://www.testingeducation.org/)
- Paul Ammann and Jeff Offutt, *"Introduction to Software Testing"*, Cambridge University Press
- Beizer, Boris, "*Software Testing and Quality Assurance"*, Van Nostrand Reinhold
- Glen Myers, *"The Art of Software Testing"*
- Stephen R. Schach, "*Software Engineering",* Richard D. Irwin, Inc. and Aksen Associates, Inc.



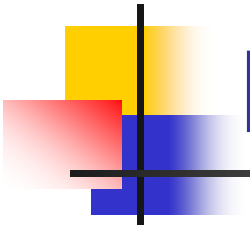
Data Flow Testing

- Shortcomings of Path Testing
  - Testing All-Nodes and All-Edges in a control flow graph may miss significant test cases
  - Testing All-Paths in a control flow graph is often too time-consuming
- Can we select a subset of these paths that will reveal the most faults?
- Data Flow Testing focuses on the points at which variables receive values and the points at which these values are used/referenced



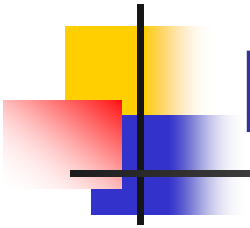
Data Flow Analysis

- Can reveal interesting bugs such as
  - A variable that is defined but never used
  - A variable that is used but never defined
  - A variable that is defined twice before it is used
- Paths from the *definition* of a variable to its *use* are more likely to contain bugs



Definitions

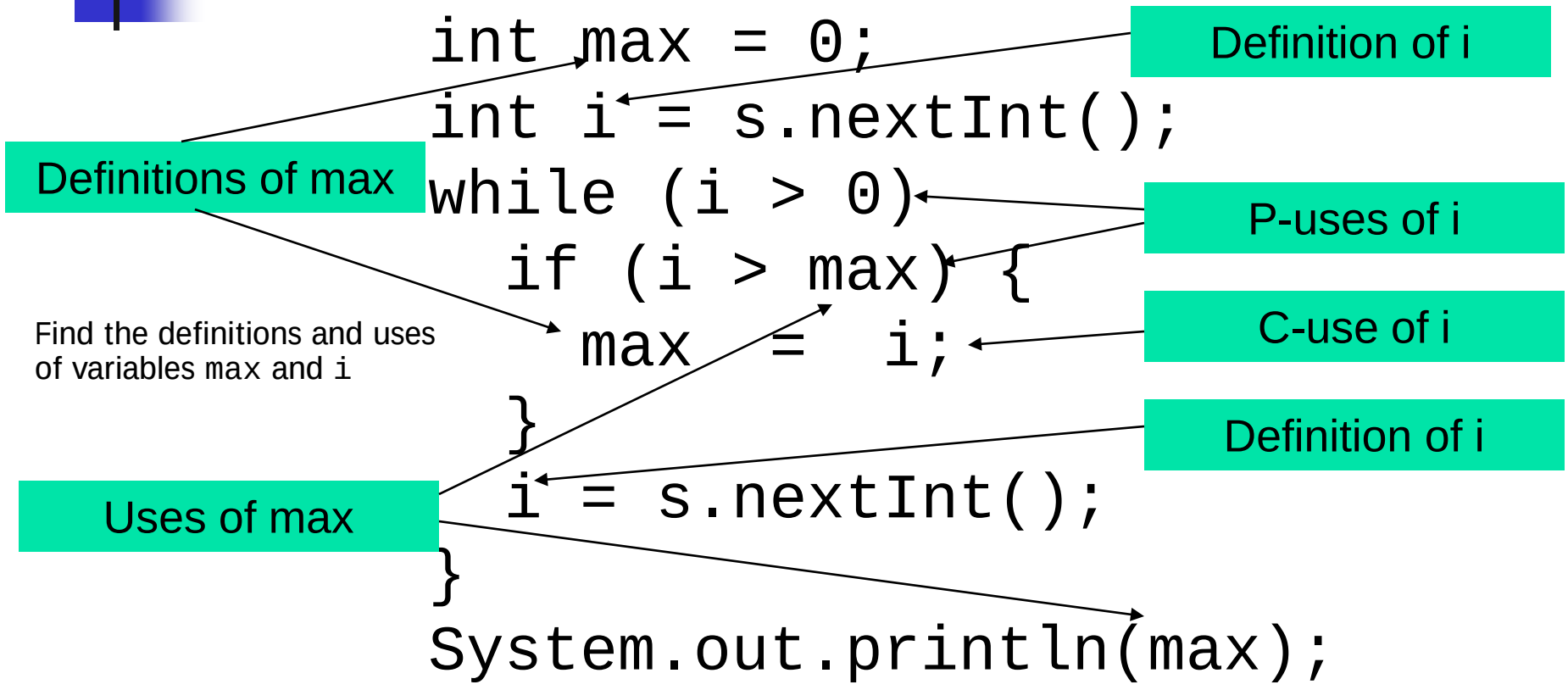
- A node in the program graph is a *defining* node for variable v if the value of v is defined at the statement fragment in that node
  - Input, assignment (left side), procedure calls
  - Contents of memory location are changed
- A node in the program graph is a *usage* node for variable v if the value of v is used at the statement fragment in that node
  - Output, assignment (right side), conditionals
  - Contents of memory location remain unchanged

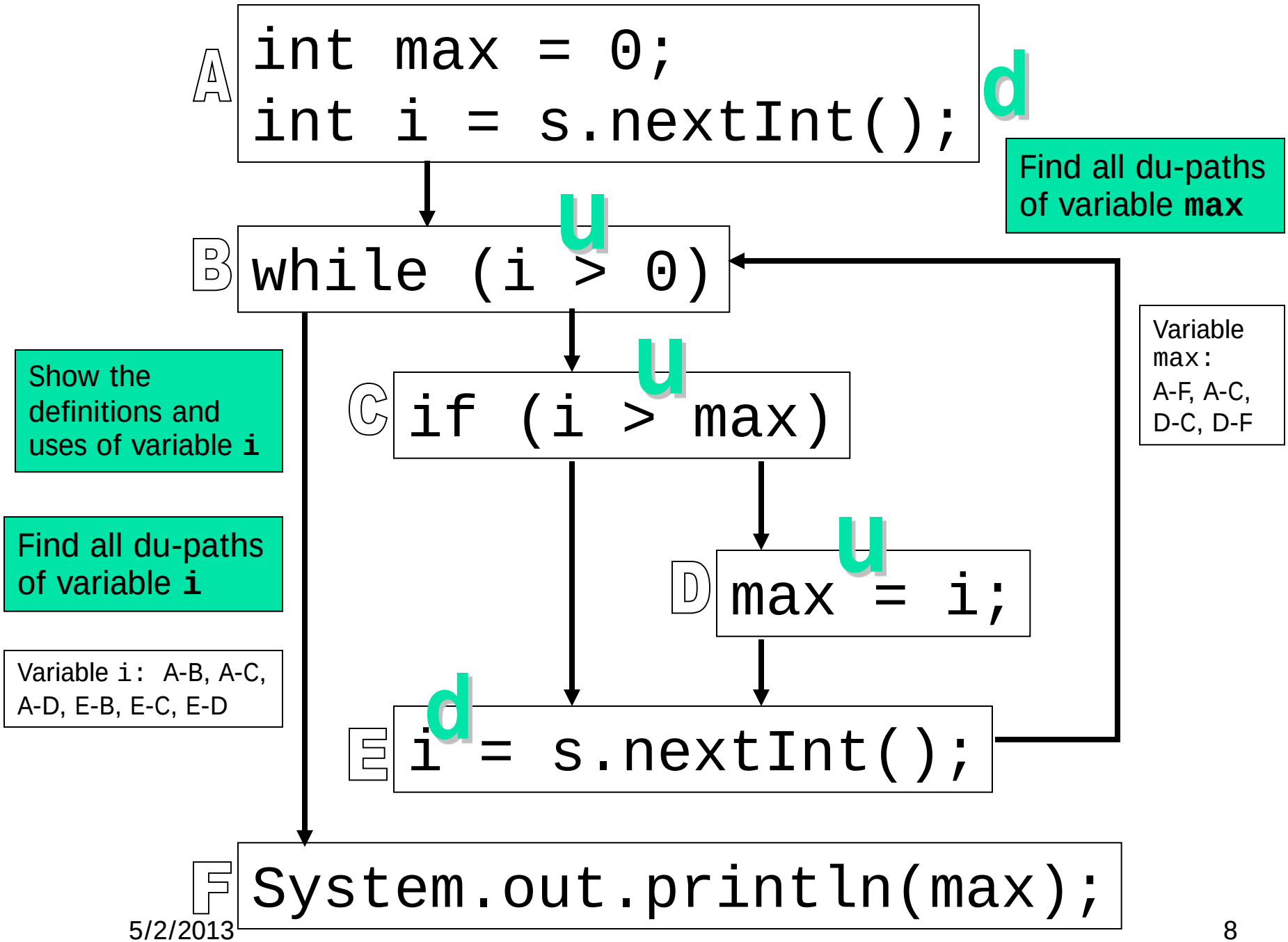


More Definitions

- A usage node is a predicate use (*P-Use*) if variable v appears in a predicate expression
- A usage node is a computation use (*C-Use*) if variable v appears in a computation
- A definition-use path (*du-path*) with respect to a variable v is a path whose first node is a defining node for v, and its last node is a usage node for v
- A du-path with no other defining node for v is a definition-clear path (*dc-path*)

An Example


int max = 0;
int i = s.nextInt();
while (i > 0)
if (i > max) {
max = i;
}
i = s.nextInt();
}
System.out.println(max);
Definition of i
Definitions of max
P-uses of i
C-use of i
Definition of i
Uses of max
Find the definitions and uses of variables max and i


A
int max = 0;
int i = s.nextInt();
d
Find all du-paths of variable max
B
while (i > 0)
u
Variable max: A-F, A-C, D-C, D-F
Show the definitions and uses of variable i
C
if (i > max)
u
Find all du-paths of variable i
D
max = i;
u
Variable i: A-B, A-C, A-D, E-B, E-C, E-D
E
i = s.nextInt();
d
F
System.out.println(max);

Commission Pseudo-code

```
1.  Program Commission (INPUT,OUTPUT)
2.  Dim locks, stocks, barrels As Integer
3.  Dim lockPrice, stockPrice, barrelPrice As Real
4.  Dim totalLocks, totalStocks, totalBarrels As Integer
5.  Dim lockSales, stockSales, barrelSales As Real
6.  Dim sales, commission As Real
7.  lockPrice = 45.0
8.  stockPrice = 30.0
9.  barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15.     Input(stocks, barrels)
16.     totalLocks = totalLocks + locks
17.     totalStocks = totalStocks + stocks
18.     totalBarrels = totalBarrels + barrels
19.     Input(locks)
20.  EndWhile
21.  Output("Locks sold: ", totalLocks)
22.  Output("Stocks sold: ", totalStocks)
23.  Output("Barrels sold: ", totalBarrels)
```

```
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30.   Then
31.     commission = 0.10 * 1000.0
32.     commission = commission + 0.15 * 800.0
33.     commission = commission + 0.20 *(sales-1800.0)
34.   Else If (sales > 1000.0)
35.           Then
36.             commission = 0.10 * 1000.0
37.             commission = commission + 0.15 *(sales-1000.0)
38.           Else
39.             commission = 0.10 * sales
40.         EndIf
41.  EndIf
42. Output("Commission is $", commission)
43. End Commission
```

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

Defining & Usage Nodes for the Commission Problem

```
1.  Program Commission (INPUT,OUTPUT)
2.  Dim locks, stocks, barrels As Integer
3.  Dim lockPrice, stockPrice, barrelPrice As Real
4.  Dim totalLocks, totalStocks, totalBarrels As Integer
5.  Dim lockSales, stockSales, barrelSales As Real
6.  Dim sales, commission As Real
7.  lockPrice = 45.0
8.  stockPrice = 30.0
9.  barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15.     Input(stocks, barrels)
16.     totalLocks = totalLocks + locks
17.     totalStocks = totalStocks + stocks
18.     totalBarrels = totalBarrels + barrels
19.     Input(locks)
20. EndWhile
21. Output("Locks sold: ", totalLocks)
22. Output("Stocks sold: ", totalStocks)
23. Output("Barrels sold: ", totalBarrels)
```

Variable	*Defined at Node*	*Used at Node*
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

Defining & Usage Nodes for the Commission Problem (continued)

```
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30.   Then
31.     commission = 0.10 * 1000.0
32.     commission = commission + 0.15 * 800.0
33.     commission = commission + 0.20 *(sales-1800.0)
34.   Else If (sales > 1000.0)
35.           Then
36.             commission = 0.10 * 1000.0
37.             commission = commission + 0.15 *(sales-1000.0)
38.           Else
39.             commission = 0.10 * sales
40.         EndIf
41.  EndIf
42. Output("Commission is $", commission)
43. End Commission
```

Variable	*Defined at Node*	*Used at Node*
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26

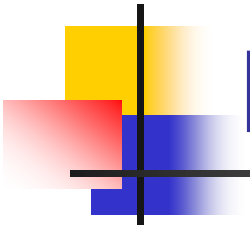
Du-Paths for the “locks” Variable in the Commission Problem

```
13. Input(locks)
14. While NOT(locks = -1)
15.   Input(stocks, barrels)
16.   totalLocks = totalLocks+locks
17.   totalStocks = totalStocks+stocks
18.   totalBarrels = totalBarrels+barrels
19.   Input(locks)
20. EndWhile
```

DEF (locks, 13), DEF (locks, 19)
USE (locks, 14), a predicate use
USE (locks, 16), a computation use

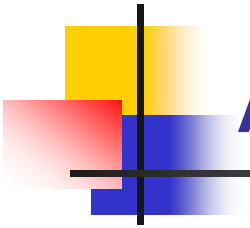
du-paths for locks are the node sequences
<13, 14> (a dc-path),
<13, 14, 15, 16>,
<19, 20, 14 >,
< 19, 20, 14 , 15, 16>

Is <13, 14, 15, 16> definition clear?
Is < 19, 20, 14, 15, 16> definition clear?



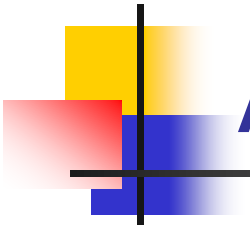
Data Flow Coverage Metrics

- Based on these definitions we can define a set of coverage metrics for a set of test cases
- We have already seen
  - All-Nodes
  - All-Edges
  - All-Paths



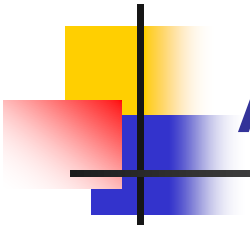
All-Defs Criterion

- For every variable v
  - For every defining node d of v
    - There exists at least one test case that follows a path from d to a usage node of v



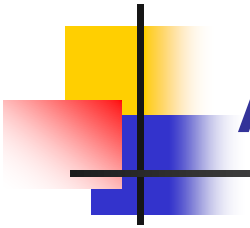
All-Uses Criterion

- For every variable v
  - For every defining node d of v
    - For every usage node u of v
      - There exists at least one test case that follows a path from d to u



All-P-Uses / Some-C-Uses

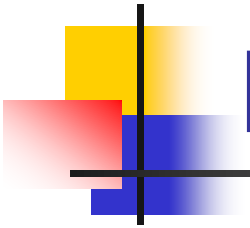
- For every variable v
  - For every defining node d of v
    - For every P-Use u of v
      - There exists at least one test case that follows a path from d to u
      - If there is no P-Use of v, there must exist at least one test case that follows a path from d to a C-Use of v



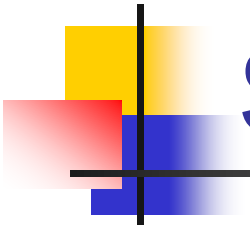
All-C-Uses / Some-P-Uses

- For every variable v
  - For every defining node d of v
    - For every C-Use u of v
      - There exists at least one test case that follows a path from d to u
      - If there is no C-Use of v, there must exist at least one test case that follows a path from d to a P-Use of v

Strategy for Generating Define/Use Test Cases

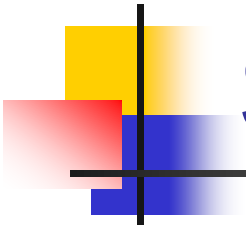


- For a particular variable:
  - find all its definition and usage nodes
  - find the du-paths and dc-paths among these
  - for each path, devise a "suitable" (functional) set of test cases
- Note: du-paths and dc-paths have both static and dynamic interpretations
  - Static: just as seen in the source code
  - Dynamic: must consider execution-time flow (particularly for loops)
- Definition clear paths are easier to test
  - No need to check each definition node, as is necessary for du-paths



Some Observations

- Data flow testing is indicated in
  - Computation-intensive applications
  - “Long” programs
  - Programs with many variables
- A definition-clear du-path represents a small function that can be tested by itself
- If a du-path is not definition-clear, it should be tested for each defining node
- Aliasing of variables causes serious problems
- Working things out by hand for anything but small methods is hopeless
- Compiler-based tools help in determining coverage values



Slice-Based Testing[1]

- Informally, a program slice is a set of program statements that contribute to or affect a value for a variable at some point in the program
- The idea of a slice is to separate a program into components that have some useful (functional) meaning

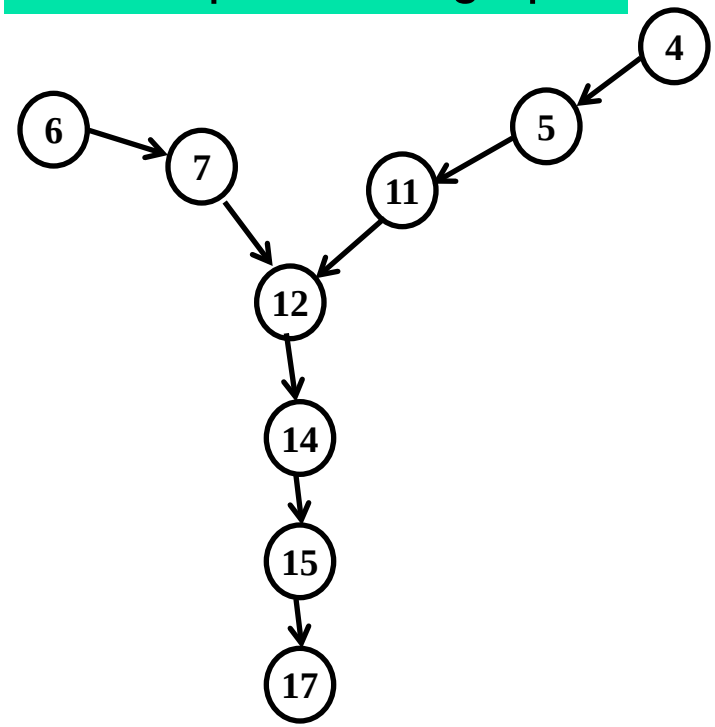
Slice-Based Testing[2]

- More formally, given a program P and a set V of variables in P, a slice on the variable set V at statement n, written S(V, n), is the set of all statements in P that contribute to the values of variables in V
- With slices we consider 5 forms of usage:
  - P-use used in predicate (decision)
  - C-use used in computation
  - O-use used for output
  - L-use used for location (pointers, subscripts)
  - I-use iteration (internal counters, loop indices)
- Also, we consider 2 forms of definition nodes
  - I-def defined by input
  - A-def defined by assignment

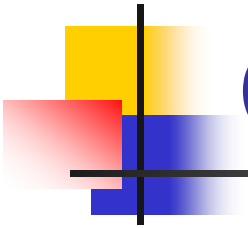
An Example

Find the (backward) slice on `out1` at line 17 & the data dependence graph

Data dependence graph




```
1.  int n1, n2, n3;
2.  int a, b, c, x, y, z;
3.  int out1, out2, out3;
4.  read(n1);
5.  a = n1;
6.  read(n2);
7.  b = n2;
8.  read(n3);
9.  c = n3;
10. x = b + c;
11. y = a + 1;
12. z = y + b;
13. out2 = z + 1;
14. z = z + 2;
15. out1 = z;
16. out3 = x;
17. print(out1, out2, out3);
```



Observations and Guidelines

- Working “backwards” from points in a program (where a fault is suspected) is the way good programmers think when they debug code (according to some researchers)
- Make slices on a single variable
- Make slices for all A-def nodes
- Make slices for P-use nodes
- Slices on non-P-use usage nodes are not very interesting
- Consider making slices compilable
- Slices do not map nicely into test cases
- Slices provide a handy way to eliminate interaction among variables
- Use the slice composition approach to redevelop difficult sections of code, and test these slices before you *splice* (compose) them with other slices