Topics in Software Dynamic White-box Testing
Part 2: Data-flow Testing

[Reading assignment: Chapter 7, pp. 105-122 plus many things in slides that are not in the book …]

Data-Flow Testing

- **Data-flow testing** uses the control flowgraph to explore the unreasonable things that can happen to data (*i.e.*, anomalies).
- Consideration of data-flow anomalies leads to test path selection strategies that fill the gaps between complete path testing and branch or statement testing.

Data-Flow Testing (Cont'd)

- **Data-flow testing** is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- *E.g.,* Pick enough paths to assure that:
  - Every data object has been initialized prior to its use.
  - All defined objects have been used at least once.

Data Object Categories

- (d) Defined, Created, Initialized
- (k) Killed, Undefined, Released
- (u) Used:
  - (c) Used in a calculation
  - (p) Used in a predicate

(d) Defined Objects

- An object (*e.g.*, variable) is **defined** when it:
  - appears in a data declaration
  - is assigned a new value
  - is a file that has been opened
  - is dynamically allocated
  - ...

(u) Used Objects

- An object is **used** when it is part of a computation or a predicate.
- A variable is used for a computation **(c)** when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement.
- A variable is used in a predicate **(p)** when it appears directly in that predicate.

Example: Definition and Uses

What are the *definitions* and *uses* for the program below?

```
1.   read (x, y);
2.    z = x + 2;
3.    if (z < y)
4          w = x + 1;
      else
5.         y = y + 1;
6.    print (x, y, w,
   z);
```

Example: Definition and Uses

```
1.   read (x, y);
2.   z = x + 2;
3.   if (z < y)
4        w = x + 1;
     else
5.       y = y + 1;
6.   print (x, y, w, z);
```

Def	*C-use*	*P-use*
x, y		
z	x	
		z, y
w	x	
y	y	
	x, y, w, z	

Static vs Dynamic Anomaly Detection

- **Static Analysis** is analysis done on source code without actually executing it.
  - *E.g.,* Syntax errors are caught by static analysis.

Static vs Dynamic Anomaly Detection (Cont'd)

- **Dynamic Analysis** is analysis done as a program is executing and is based on intermediate values that result from the program's execution.
  - *E.g.,* A division by 0 error is caught by dynamic analysis.
- If a data-flow anomaly can be detected by static analysis then the anomaly does not concern testing. (Should be handled by the compiler.)

Anomaly Detection Using Compilers

- Compilers are able to detect several data-flow anomalies using static analysis.
- *E.g.,* By forcing declaration before use, a compiler can detect anomalies such as:
  - **-u**
  - **ku**
- Optimizing compilers are able to detect some dead variables.

Is Static Analysis Sufficient?

- **Questions:**
- Why isn't static analysis enough?
- Why is testing required?
- Could a good compiler detect all data-flow anomalies?
- **Answer:** No. Detecting all data-flow anomalies is provably unsolvable.

Static Analysis Deficiencies

- Current static analysis methods are inadequate for:
  - **Dead Variables:** Detecting unreachable variables is unsolvable in the general case.
  - **Arrays:** Dynamically allocated arrays contain garbage unless they are initialized explicitly. (**-u** anomalies are possible)

Static Analysis Deficiencies (Cont'd)

- **Pointers:** Impossible to verify pointer values at compile time.
- **False Anomalies:** Even an obvious bug (*e.g.,* **ku**) may not be a bug if the path along which the anomaly exists is unachievable. (Determining whether a path is or is not achievable is unsolvable.)

Data-Flow Modeling

- Data-flow modeling is based on the control flowgraph.
- Each link is annotated with:
  - symbols (*e.g.,* **d**, **k**, **u**, **c**, **p**)
  - sequences of symbols (*e.g.,* **dd**, **du**, **ddd**)
- that denote the sequence of data operations on that link with respect to the variable of interest.

Simple Path Segments

- A **Simple Path Segment** is a path segment in which at most one node is visited twice.
  - *E.g.,* (7,4,5,6,7) is simple.
- Therefore, a simple path may or may not be loop-free.

Loop-free Path Segments

- A **Loop-free Path Segment** is a path segment for which every node is visited at most once.
  - *E.g.,* (4,5,6,7,8,10) is loop-free.
  - path (10,11,4,5,6,7,8,10,11,12) is not loop-free because nodes 10 and 11 are visited twice.

du Path Segments

- A **du Path** is a path segment such that if the last link has a use of **X**, then the path is simple and definition clear.

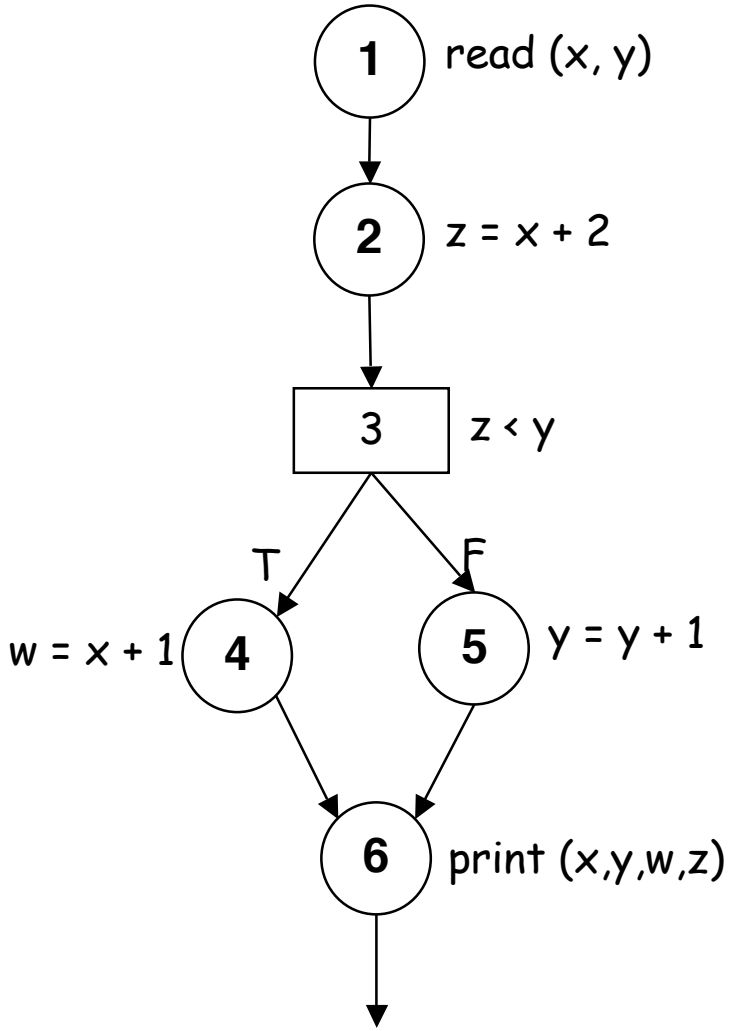
def-use Associations

- A def-use association is a triple (x, d, u,), where:

  x is a variable,
  d is a node containing a definition of x,
  u is either a statement or predicate node containing a use of x,

  and there is a sub-path in the flow graph from d to u with no other definition of x between d and u.

Example: Def-Use Associations



Some Def-Use Associations:

(x, 1, 2), (x, 1, 4), …

(y, 1, (3,t)), (y, 1, (3,f)), (y, 1, 5), …

(z, 2, (3,t)),...

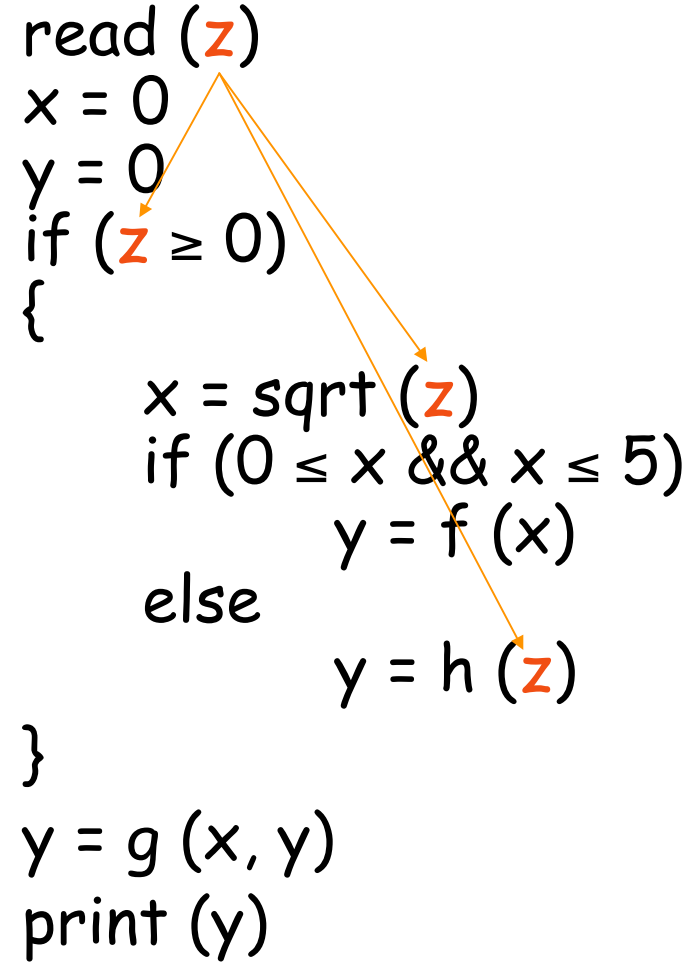
Example: Def-Use Associations

What are all the *def-use associations* for the program below?

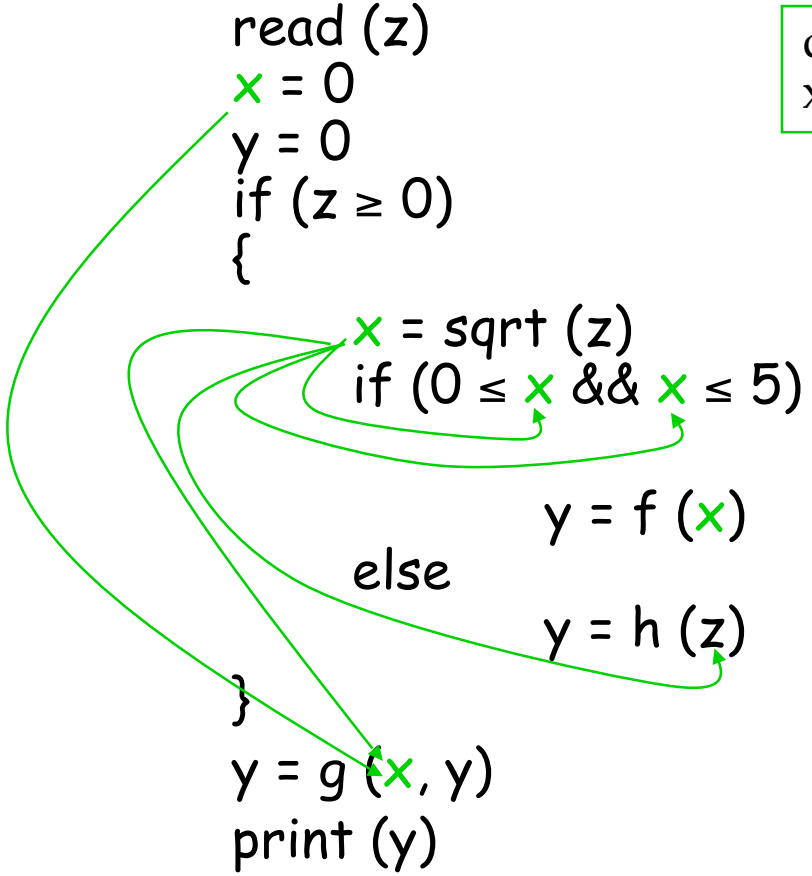
```
read (z)
x = 0
y = 0
if (z ≥ 0)
{
        x = sqrt (z)
        if (0 ≤ x && x ≤ 5)
                y = f (x)
        else
                y = h (z)
}
y = g (x, y)
print (y)
```

Example: Def-Use Associations

def-use associations for variable z.

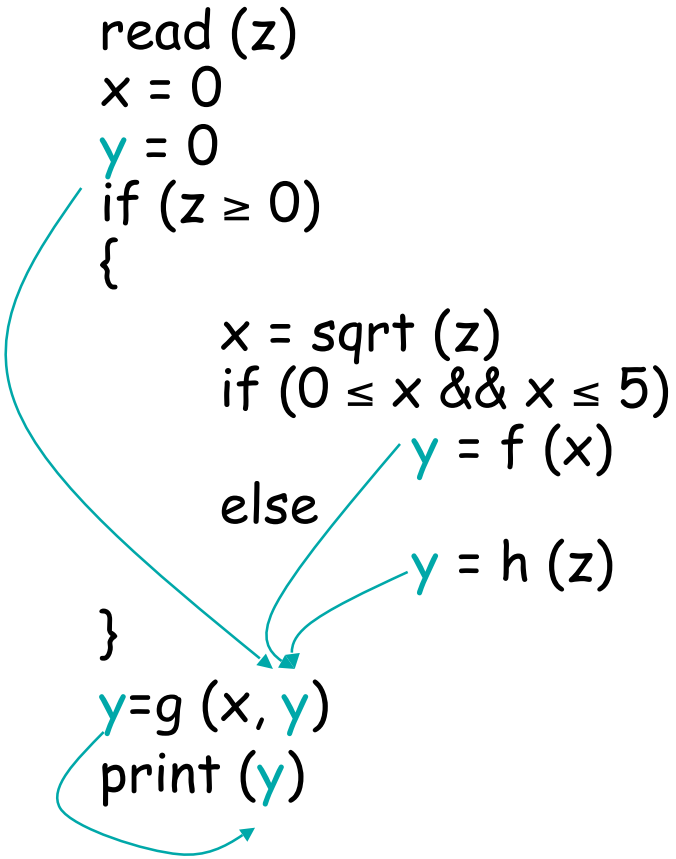


Example: Def-Use Associations



def-use associations for variable x.

Example: Def-Use Associations



def-use associations for variable y.

Definition-Clear Paths

- A path $(i, n_1, ..., n_m, j)$ is called a *definition-clear path* with respect to x from node i to node j if it contains no definitions of variable x in nodes $(n_1, ..., n_m, j)$.

- The family of data flow criteria requires that the test data execute definition-clear paths from each node containing a definition of a variable to specified nodes containing *c-use* and edges containing *p-use* of that variable.

Data-Flow Testing Strategies

- All **du** Paths (ADUP)
- All **Uses** (AU)
- Others not covered in this course …

All du Paths Strategy (ADUP)

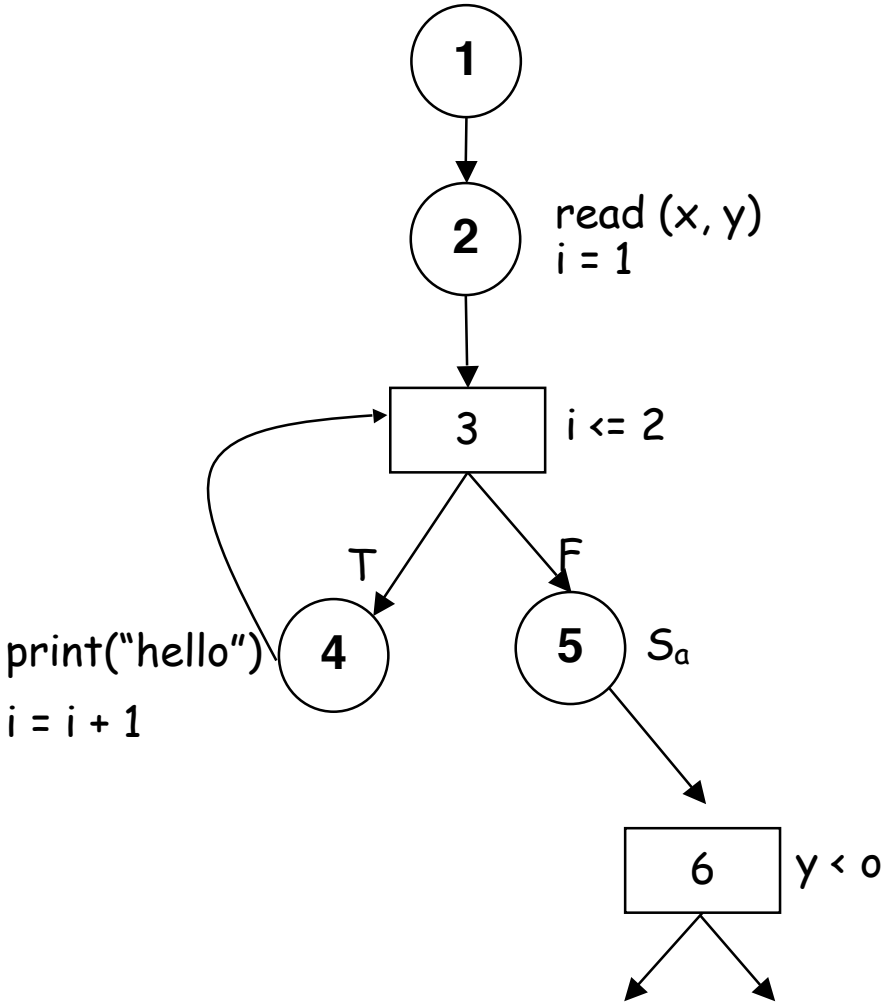
- **ADUP** is one of the strongest data-flow testing strategies.
- **ADUP** requires that every **du** path from every definition of every variable to every use of that definition be exercised under some test All **du** Paths Strategy (ADUP).

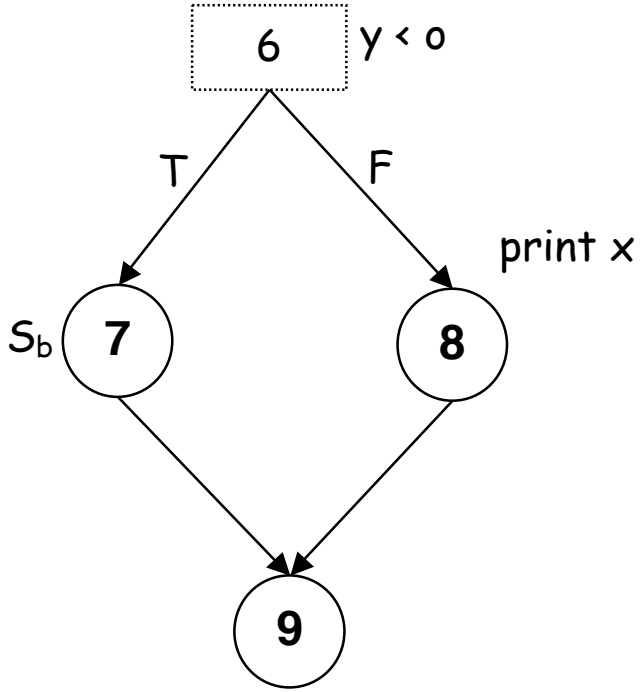
An example: All-du-paths

What are all the du-paths in the following program ?

```
read (x,y);
for (i = 1; i <= 2; i++)
        print ("hello");
Sa;
if (y < 0)
        Sb;
else
        print (x);
```

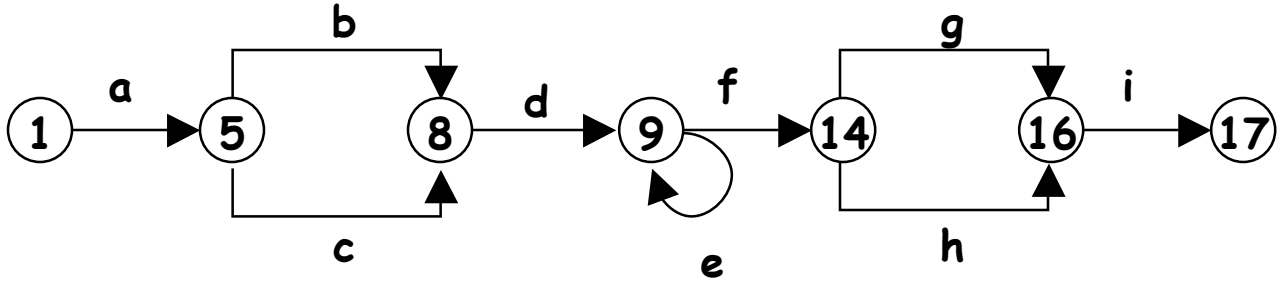
An example: All-du-paths





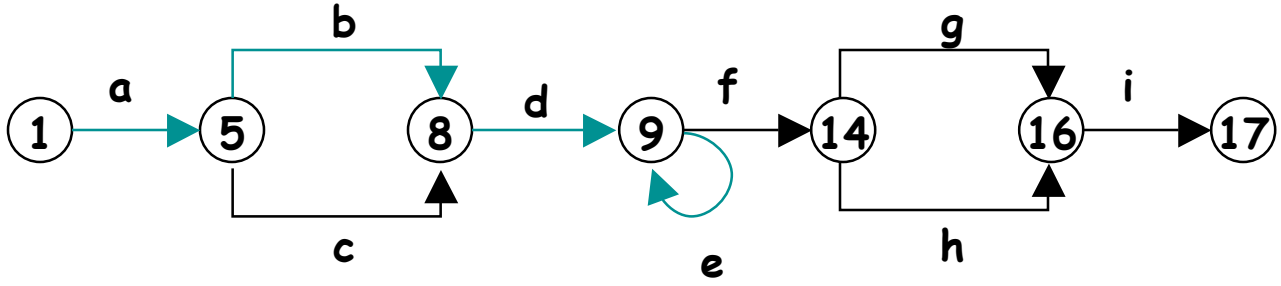
Example: pow(x,y)

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
 */
 1        void pow (int x, y)
 2        {
 3        float z;
 4        int p;
 5        if (y < 0)
 6           p = 0 – y;
 7        else p = y;
 8        z = 1.0;
 9        while (p != 0)
 10          {
 11          z = z * x;
 12          p = p – 1;
 13          }
 14       if (y < 0)
 15          z = 1.0 / z;
 16       printf(z);
 17       }
```



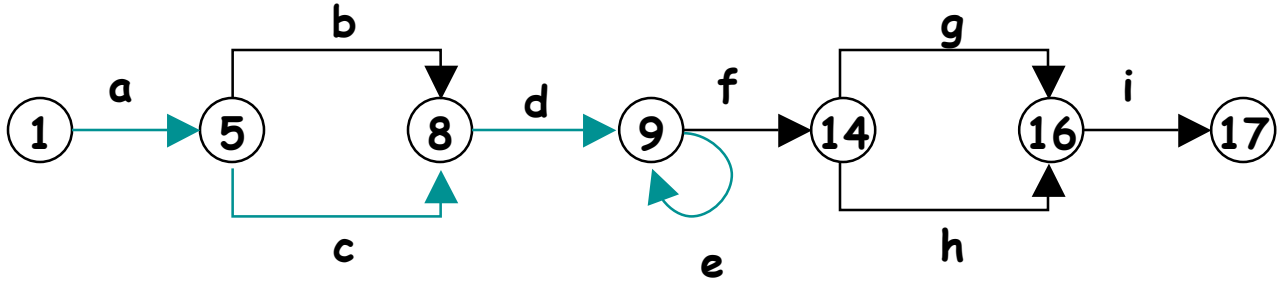
Example: pow(x,y)
du-Path for Variable x

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
 */
1        void pow (int x, y)
2        {
3        float z;
4        int p;
5        if (y < 0)
6           p = 0 – y;
7        else p = y;
8        z = 1.0;
9        while (p != 0)
10          {
11          z = z * x;
12          p = p – 1;
13          }
14       if (y < 0)
15          z = 1.0 / z;
16       printf(z);
17       }
```



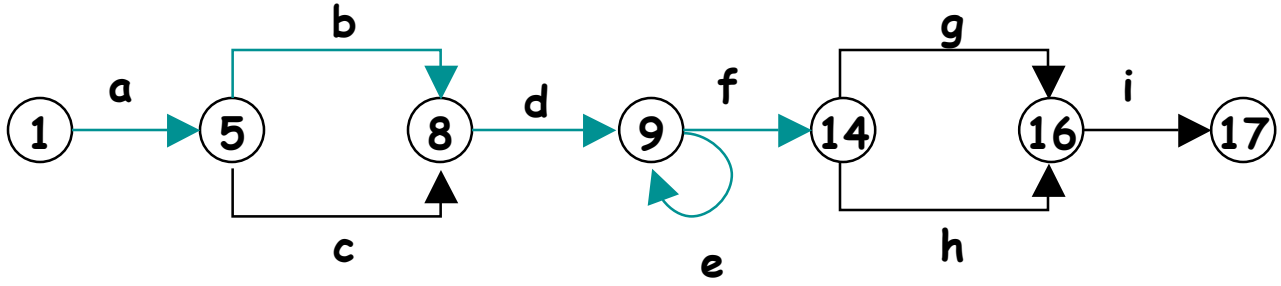
Example: pow(x,y)
du-Path for Variable x

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
 */
1        void pow (int x, y)
2        {
3        float z;
4        int p;
5        if (y < 0)
6           p = 0 – y;
7        else p = y;
8        z = 1.0;
9        while (p != 0)
10          {
11          z = z * x;
12          p = p – 1;
13          }
14       if (y < 0)
15          z = 1.0 / z;
16       printf(z);
17       }
```



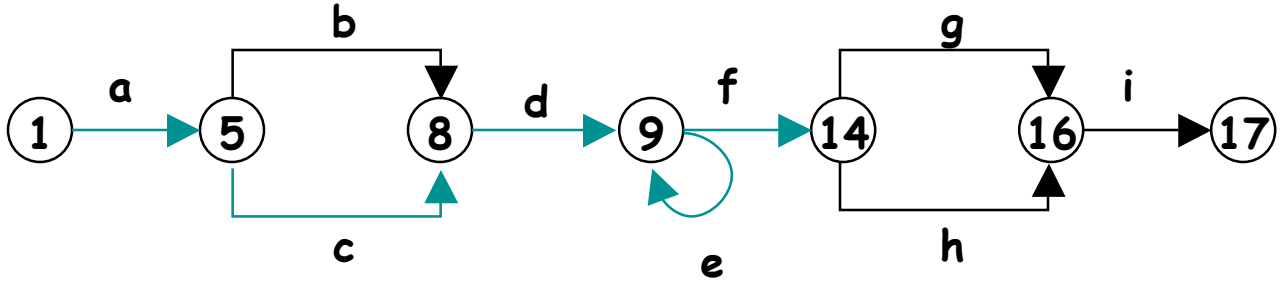
Example: pow(x,y)
du-Path for Variable y

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
 */
1        void pow (int x, y)
2        {
3        float z;
4        int p;
5        if (y < 0)
6           p = 0 – y;
7        else p = y;
8        z = 1.0;
9        while (p != 0)
10          {
11          z = z * x;
12          p = p – 1;
13          }
14       if (y < 0)
15          z = 1.0 / z;
16       printf(z);
17       }
```



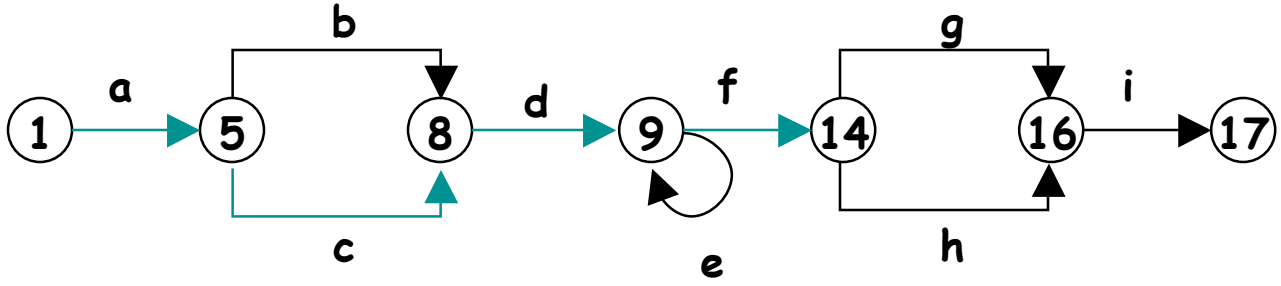
Example: pow(x,y)
du-Path for Variable y

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
 */
1        void pow (int x, y)
2        {
3        float z;
4        int p;
5        if (y < 0)
6           p = 0 – y;
7        else p = y;
8        z = 1.0;
9        while (p != 0)
10          {
11          z = z * x;
12          p = p – 1;
13          }
14       if (y < 0)
15          z = 1.0 / z;
16       printf(z);
17       }
```



Example: pow(x,y)
du-Path for Variable y

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
 */
1        void pow (int x, y)
2        {
3        float z;
4        int p;
5        if (y < 0)
6           p = 0 – y;
7        else p = y;
8        z = 1.0;
9        while (p != 0)
10          {
11          z = z * x;
12          p = p – 1;
13          }
14       if (y < 0)
15          z = 1.0 / z;
16       printf(z);
17       }
```



All Uses Strategy (AU)

- **AU** requires that at least one path from every definition of every variable to every use of that definition be exercised under some test.
- Hence, at least one definition-clear path from every definition of every variable to every use of that definition be exercised under some test.
- Clearly, **AU** < **ADUP**.

Effectiveness of Strategies

- Ntafos compared **Random**, **Branch**, and **All uses** testing strategies on 14 Kernighan and Plauger programs.
- Kernighan and Plauger programs are a set of mathematical programs with known bugs that are often used to evaluate test strategies.
- Ntafos conducted two experiments:

Results of 2 of the 14 Ntafos Experiments

Strategy	Mean Number of Test Cases	Percentage of Bugs Found
Random	35	93.7
Branch	3.8	91.6
All Uses	11.3	96.3

Strategy	Mean Number of Test Cases	Percentage of Bugs Found
Random	100	79.5
Branch	34	85.5
All Uses	84	90.0

Data-Flow Testing Tips

- Resolve all data-flow anomalies.
- Try to do all data-flow operations on a variable within the same routine (*i.e.,* avoid integration problems).
- Use strong typing and user defined types when possible.

Data-Flow Testing Tips (Cont'd)

- Use explicit (rather than implicit) declarations of data when possible.
- Put data declarations at the top of the routine and return data objects at the bottom of the routine.

Summary

- Data are as important as code.
- Define what you consider to be a data-flow anomaly.
- Data-flow testing strategies span the gap between **all paths** and **branch** testing.

Summary

- **AU** has the best payoff for the money. It seems to be no worse than twice the number of required test cases for **branch** testing, but the results are much better.
- Path testing with **Branch Coverage** and Data-flow testing with **AU** is a very good combination.

You now know …

- … data flow testing
- … du coverage
- … AU coverage