Topics in Data-Flow Testing

Material drawn from [Beizer]

Courtesy Spiros Mancoridis

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
  - ...

(k) Killed Objects

- An object is **killed** when it is:
  - released (*e.g.*, free) or otherwise made unavailable (*e.g.*, out of scope)
  - a loop control variable when the loop exits
  - a file that has been closed
  - ...

(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.

Data-Flow Anomalies

- A **data-flow anomaly** is denoted by a two character sequence of actions. *E.g.*,
  - **ku**: Means that an object is killed and then used.
  - **dd**: Means that an object is defined twice without an intervening usage.

Example

- *E.g.*, of a valid (not anomalous) scenario where variable *A* is a **dpd**:

```
A = C + D;
if(A > 0)
    X = 1;
else
    X = -1;
A = B + C;
```

Two Letter Combinations for ***d k u***

- **dd**: Probably harmless, but suspicious.
- **dk**: Probably a bug.
- **du**: Normal situation.
- **kd**: Normal situation.
- **kk**: Harmless, but probably a bug.
- **ku**: Definitely a bug.
- **ud**: Normal situation (reassignment).
- **uk**: Normal situation.
- **uu**: Normal situation.

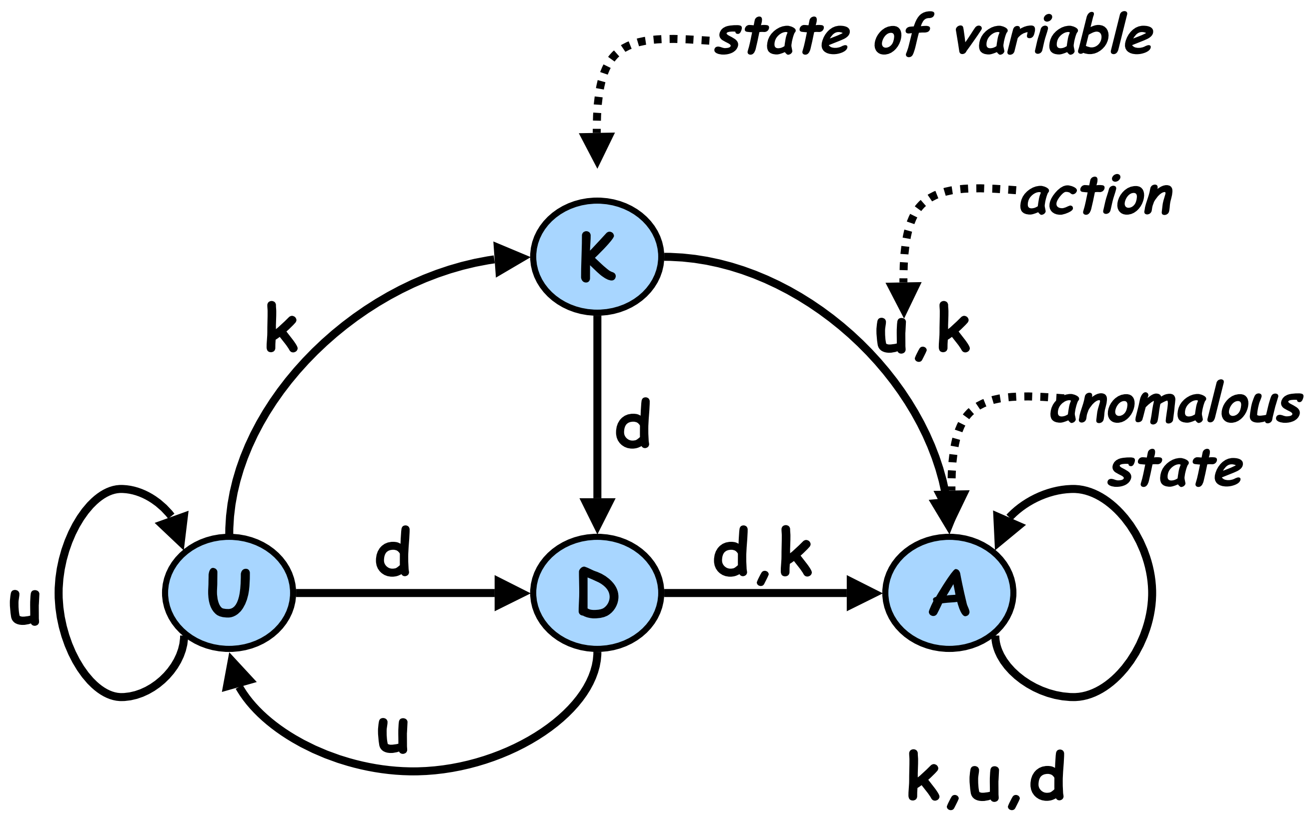
Single Letter Situations

- A leading dash means that nothing of interest (**d**, **k**, **u**) occurs prior to the action noted along the entry-exit path of interest.
- A trailing dash means that nothing of interest happens after the point of action until the exit.

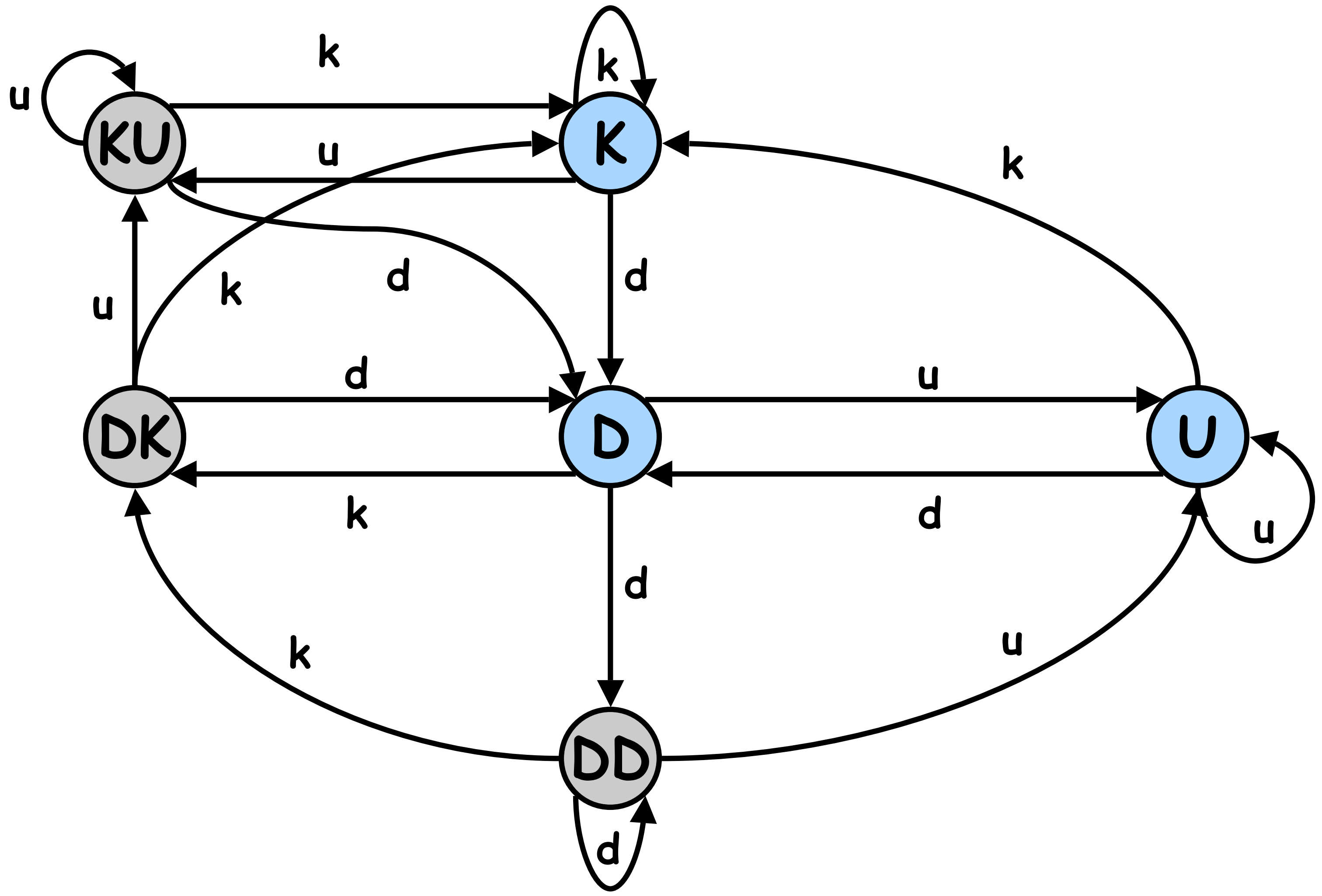
Single Letter Situations

- **-k**: Possibly anomalous:
  - Killing a variable that does not exist.
  - Killing a variable that is global.
- **-d**: Normal situation.
- **-u**: Possibly anomalous, unless variable is global.
- **k-**: Normal situation.
- **d-**: Possibly anomalous, unless variable is global.
- **u-**: Normal situation.

Data-Flow Anomaly State Graph



Data-Flow Anomaly State Graph with Variable Redemption



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 <u>does not</u> 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?

- **<u>Questions:</u>**
- Why isn’t static analysis enough?
- Why is testing required?
- Could a good compiler detect all data-flow anomalies?
- **<u>Answer:</u>** 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)

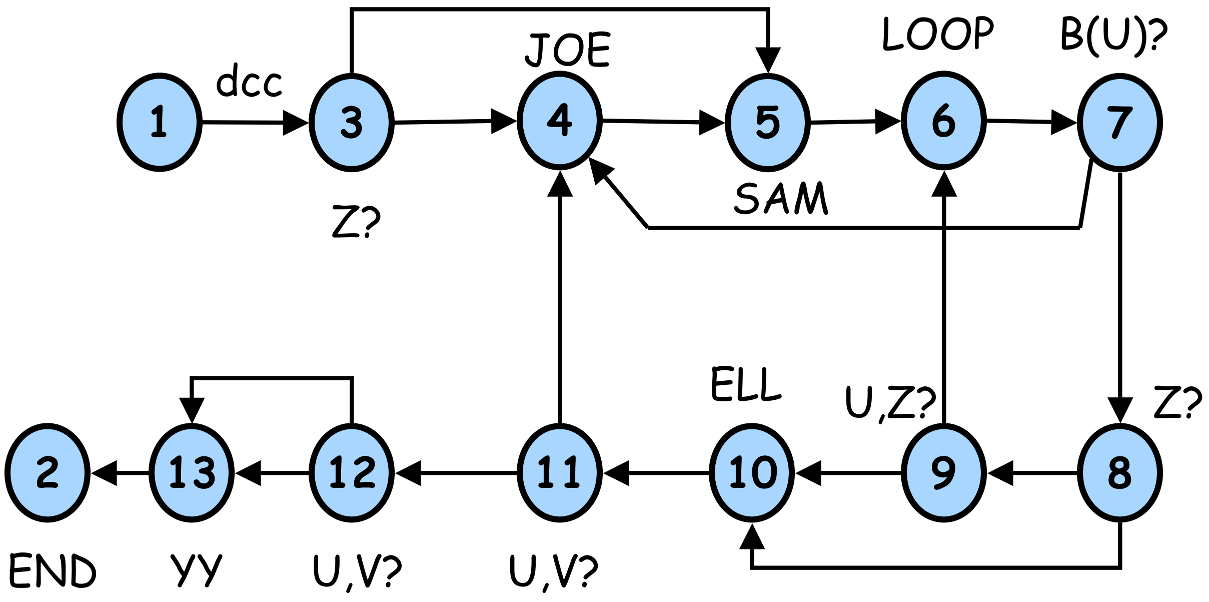
- **Pointers:** Impossible to verify pointer values at compile time.
- **False Anomalies:** Even an obvious bug (*e.g.*, **ku**) may not be a bug if the path along which the anomaly exists is unachievable. (Determining whether a path is or is not achievable is unsolvable.)

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.

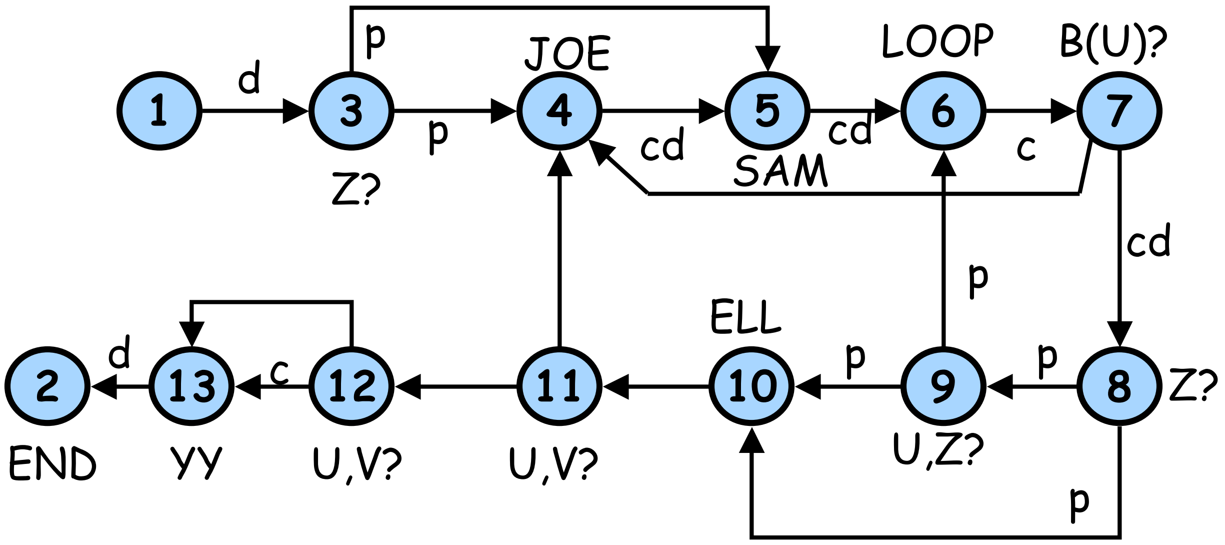
Control Flowgraph Annotated for X and Y Data Flows

```
1           INPUT X,Y
            Z:= X+Y
            Y:= X-Y
3           IF Z>=0 GOTO SAM
4           JOE: Z:=Z-1
5           SAM: Z:=Z+V
            U:=0
6           LOOP
            B(U),Q(V):=(Z+V)*U
7           IF B(U)=0 GOTO JOE
            Z:=Z-1
8           IF Z=0 GOTO ELL
            U:=U+1
9           UNTIL U=z
            B(U-1):=B(U+1)+Q(V-1)
10      ELL: B(U+Q(V)):=U+V
11          IF U=V GOTO JOE
12          IF U>V THEN U:=Z
13      YY:Z:=U
2           END
```



Control Flowgraph Annotated for Z Data Flows

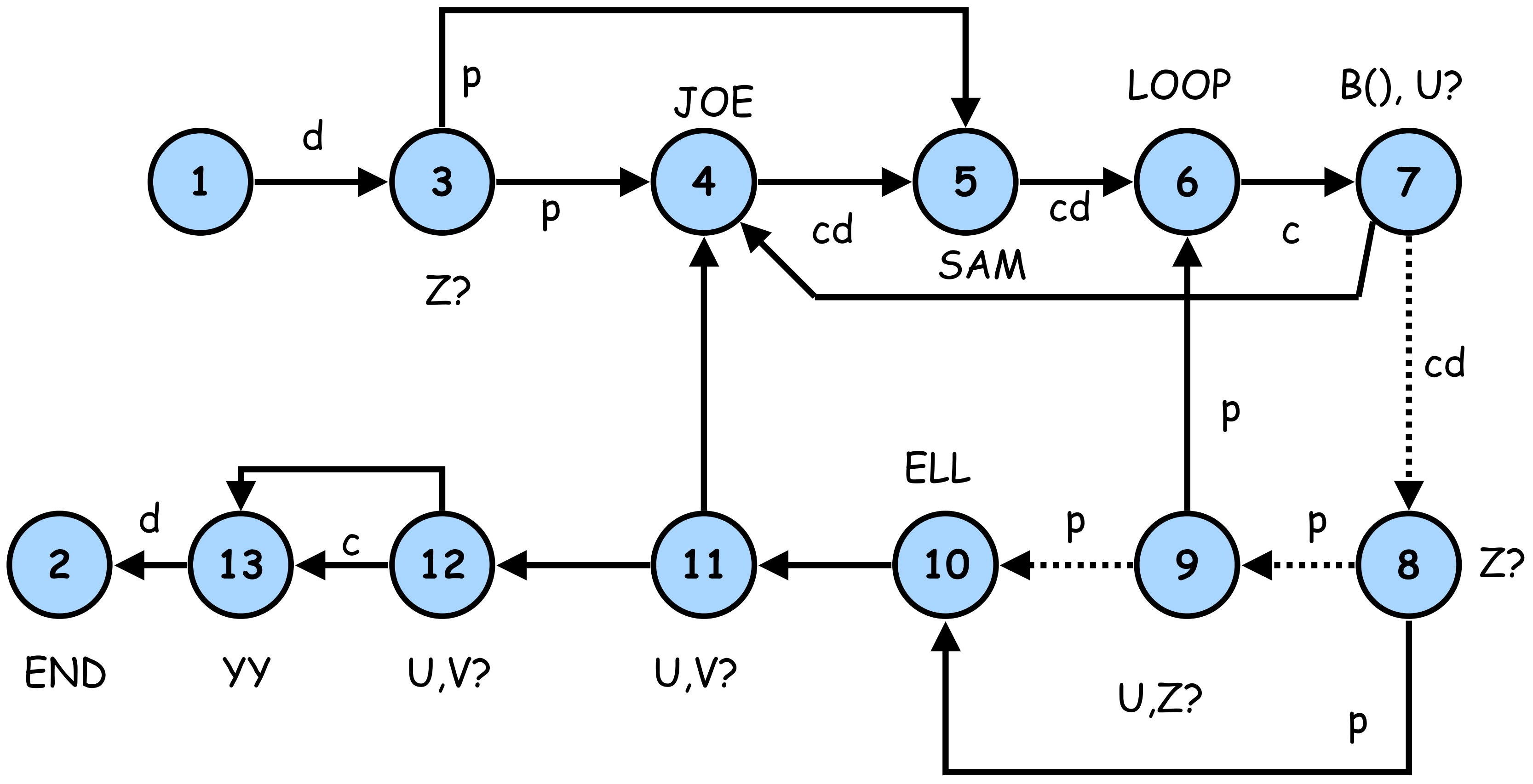
```
1          INPUT X,Y
           Z:= X+Y
           Y:= X-Y
3          IF Z>=0 GOTO SAM
4      JOE: Z:=Z-1
5      SAM: Z:=Z+V
           U:=0
6          LOOP
           B(U),Q(V):=(Z+V)*U
7          IF B(U)=0 GOTO JOE
           Z:=Z-1
8          IF Z=0 GOTO ELL
           U:=U+1
9          UNTIL U=z
           B(U-1):=B(U+1)+Q(V-1)
10      ELL: B(U+Q(V)):=U+V
11         IF U=V GOTO JOE
12         IF U>V THEN U:=Z
13      YY:Z:=U
2          END
```



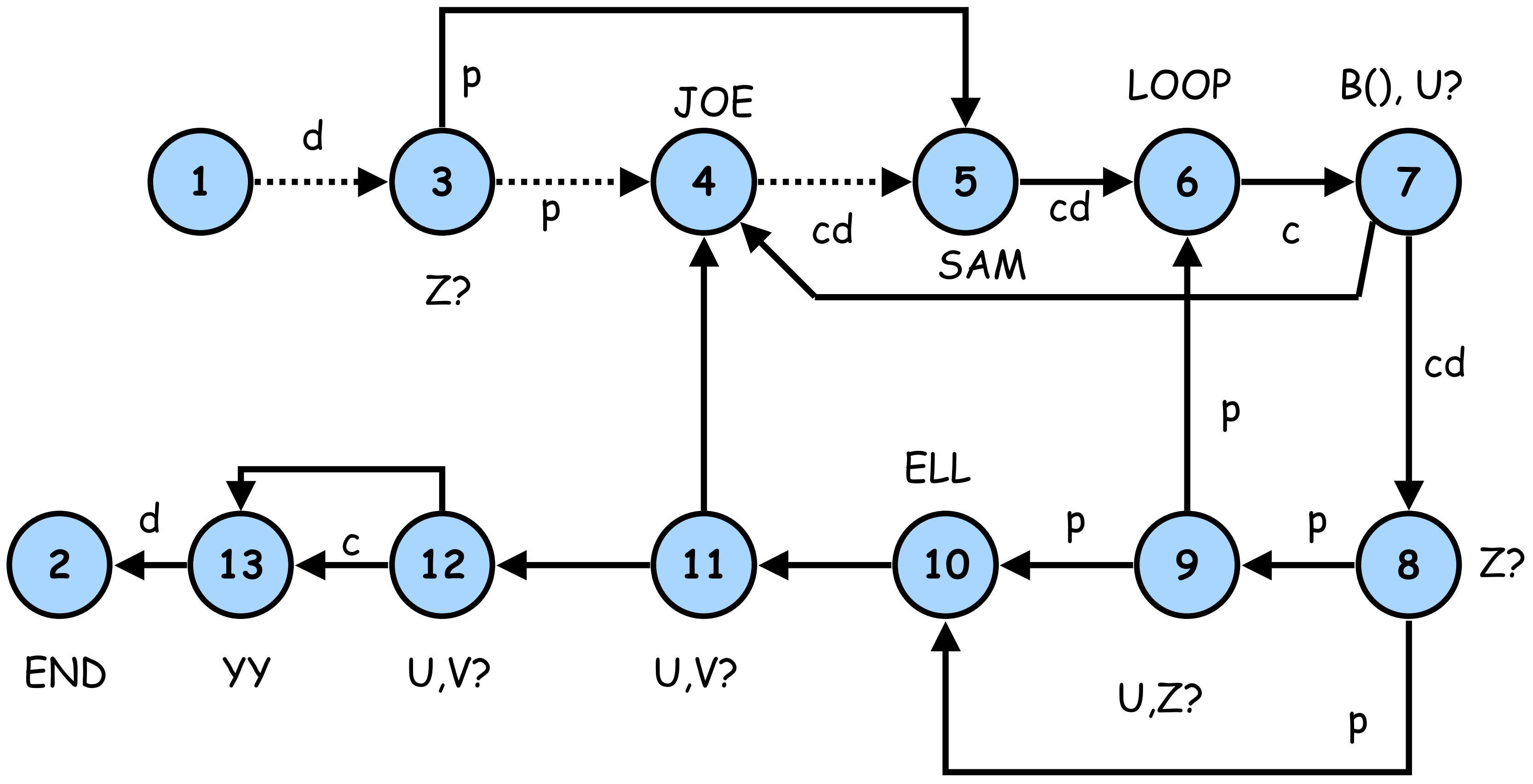
Definition-Clear Path Segments

- A **Definition-clear Path Segment** (w.r.t. variable **X**) is a connected sequence of links such that X is defined on the first link and not redefined or killed on any subsequent link of that path segment.

Definition-Clear Path Segments for Variable Z (Cont'd)



Non Definition-Clear Path Segments for Variable Z (Cont'd)



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.

Data-Flow Testing Strategies

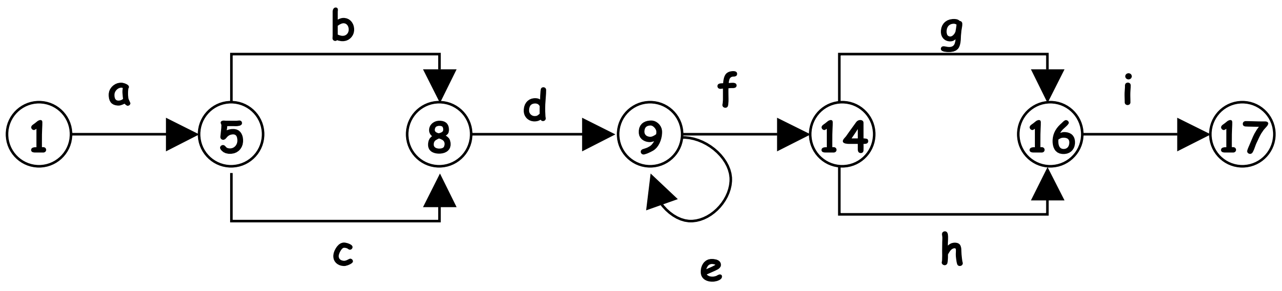
- All **du** Paths (ADUP)
- All **Uses** (AU)
- All **p-uses**/some **c-uses** (APU+C)
- All **c-uses**/some **p-uses** (ACU+P)
- All **Definitions** (AD)
- All **p-uses** (APU)
- All **c-uses** (ACU)

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).

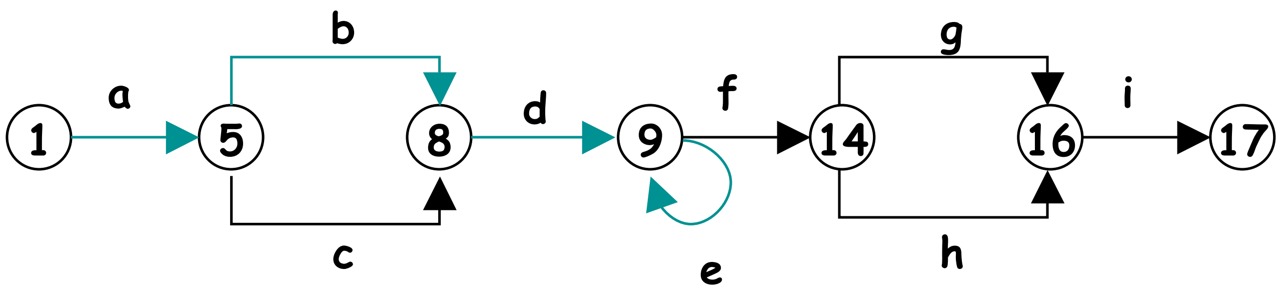
Example: pow(x,y)

```
/* 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        }
```



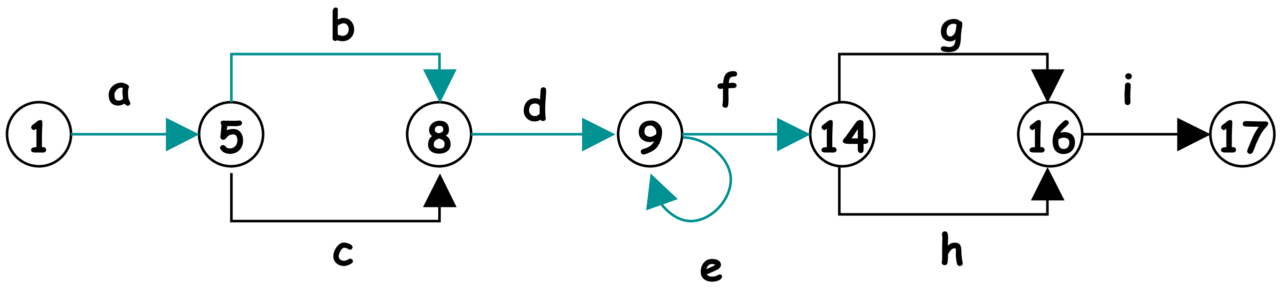
Example: pow(x,y) du-Path for Variable x

```
/* 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         }
```

$$\begin{tikzcd}
1 \arrow[r, "a"] & 5 \arrow[r, bend left, "b"] \arrow[r, bend right, "c"'] & 8 \arrow[r, "d"] & 9 \arrow[loop below, "e"] \arrow[r, "f"] & 14 \arrow[r, bend left, "g"] \arrow[r, bend right, "h"'] & 16 \arrow[r, "i"] & 17
\end{tikzcd}$$

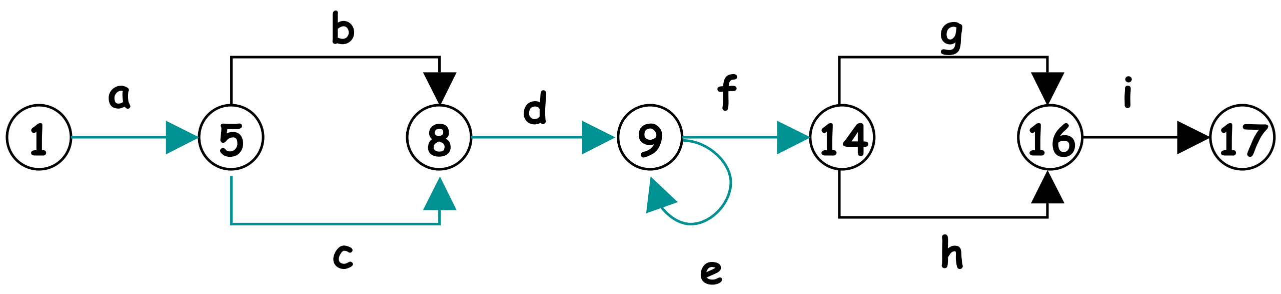
Example: pow(x,y) du-Path for Variable y

```
/* 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        }
```



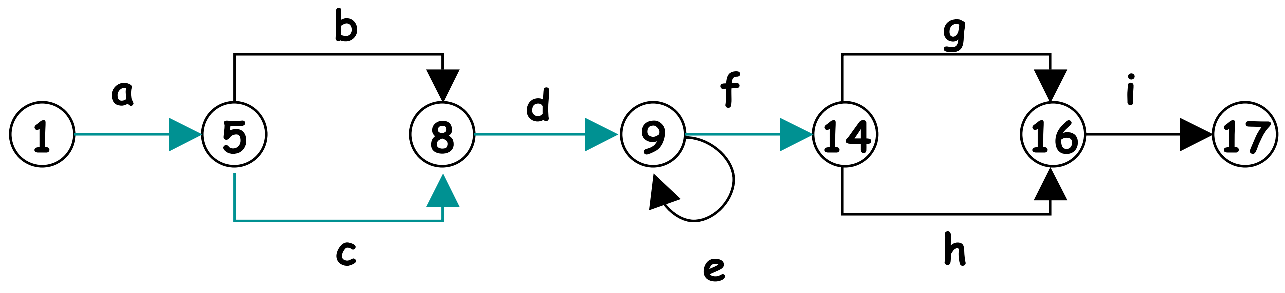
Example: pow(x,y)
du-Path for Variable y

```
/* 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      }
```



Example: Using du-Path Testing to Test Program COUNT

- Consider the following program:

```
/* COUNT
   This program counts the number of characters and lines in a text file
    INPUT: Text File
    OUTPUT: Number of characters and number of lines.
*/
1           main(int argc, char *argv[])
2           {
3           int numChars = 0;
4           int numLines = 0;
5           char chr;
6           FILE *fp = NULL;
7
```

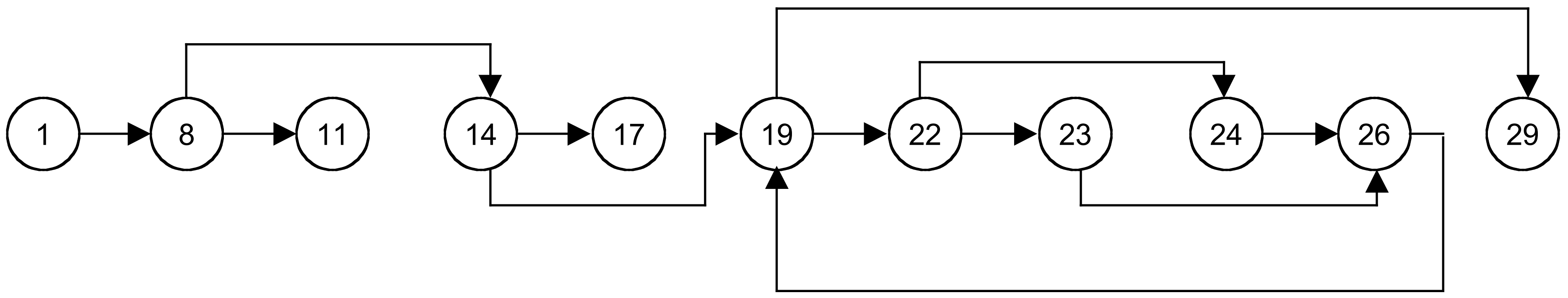
Program COUNT (Cont’d)

```
     if (argc < 2)
        {
        printf(“\nUsage: %s <filename>”, argv[0]);
        return (-1);
        }
    fp = fopen(argv[1], “r”);
    if (fp == NULL)
       {
        perror(argv[1]);      /* display error message */
        return (-2);
        }
```

Program COUNT (Cont'd)

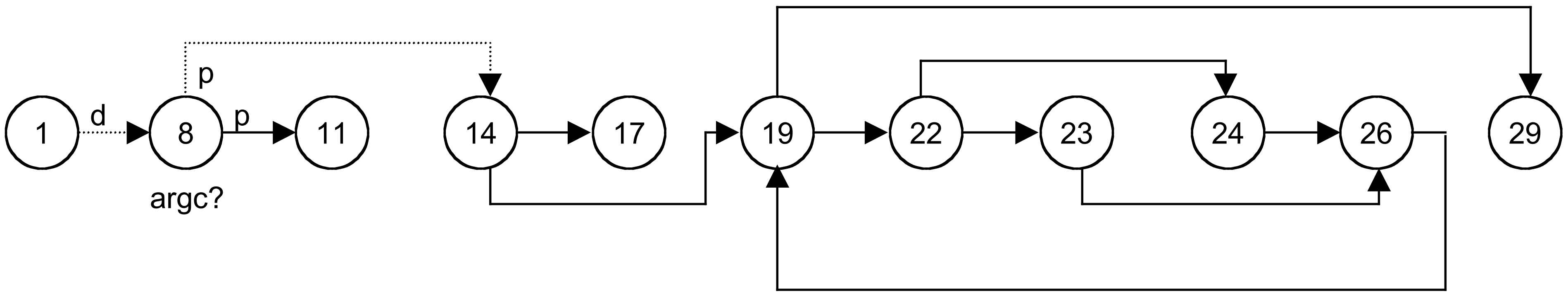
```
  while (!feof(fp))
   {
   chr = getc(fp);         /* read character */
   if (chr == '\n')       /* if carriage return */
       ++numLines;
   else
       ++numChars;
   }
printf("\nNumber of characters = %d", numChars);
printf("\nNumber of lines = %d", numLines);
}
```

The Flowgraph for COUNT

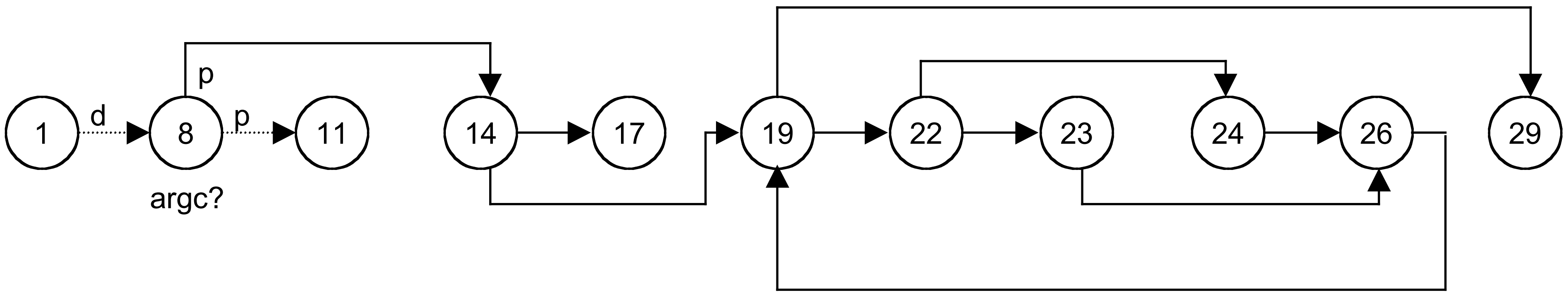


- The junction at line 12 and line 18 are not needed because if you are at these lines then you must also be at line 14 and 19 respectively.

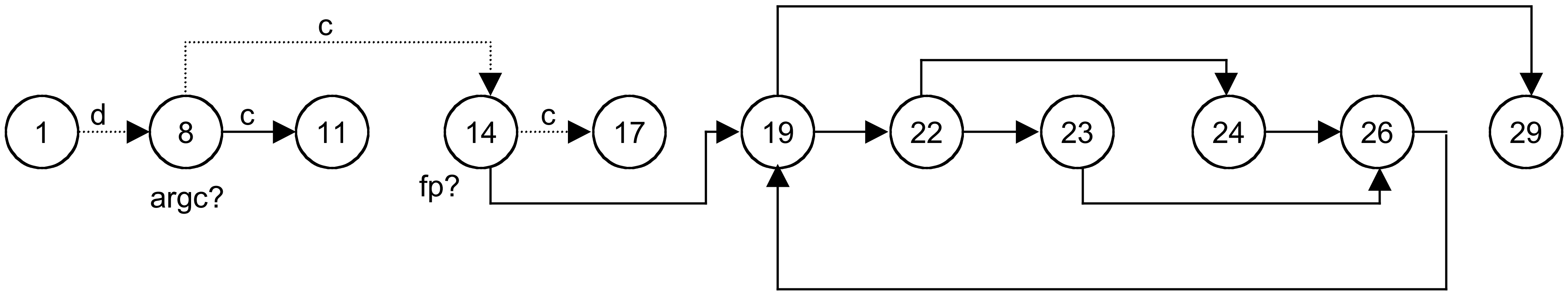
du-Path for argc



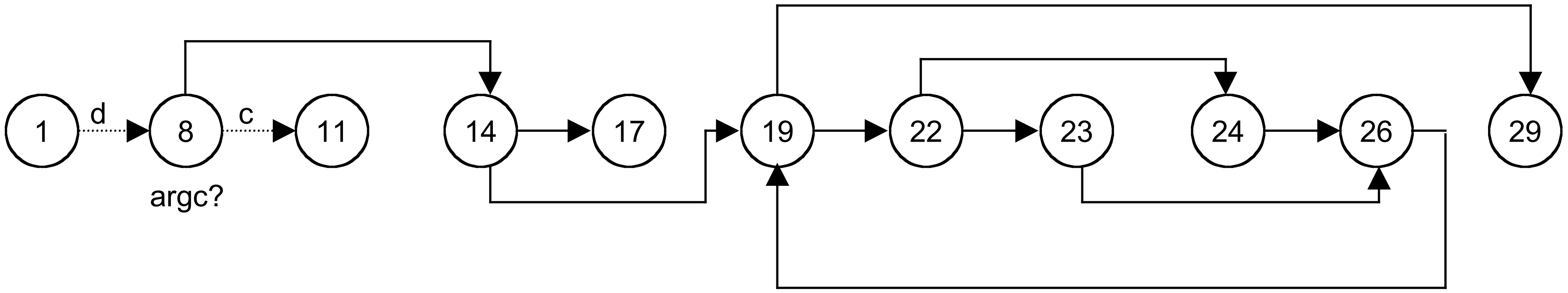
du-Path for argc



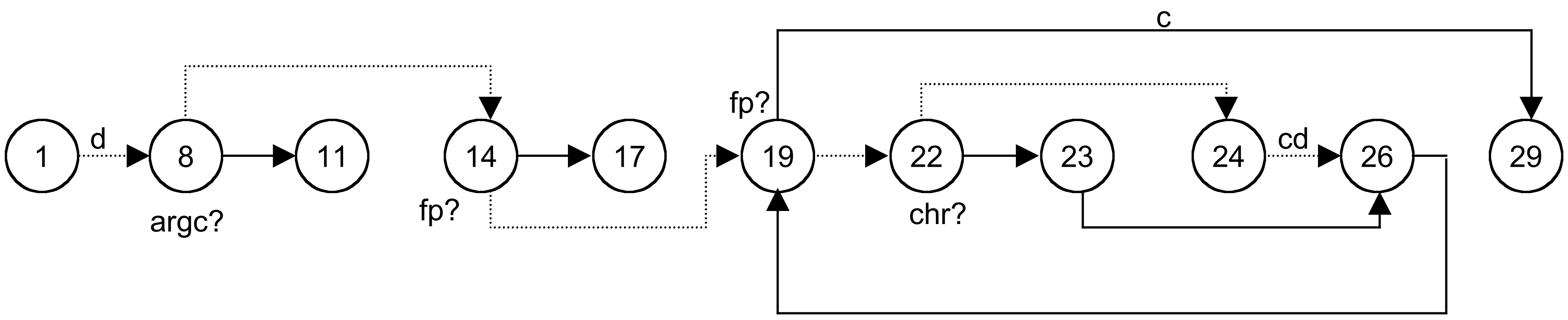
du-Path for argv[]



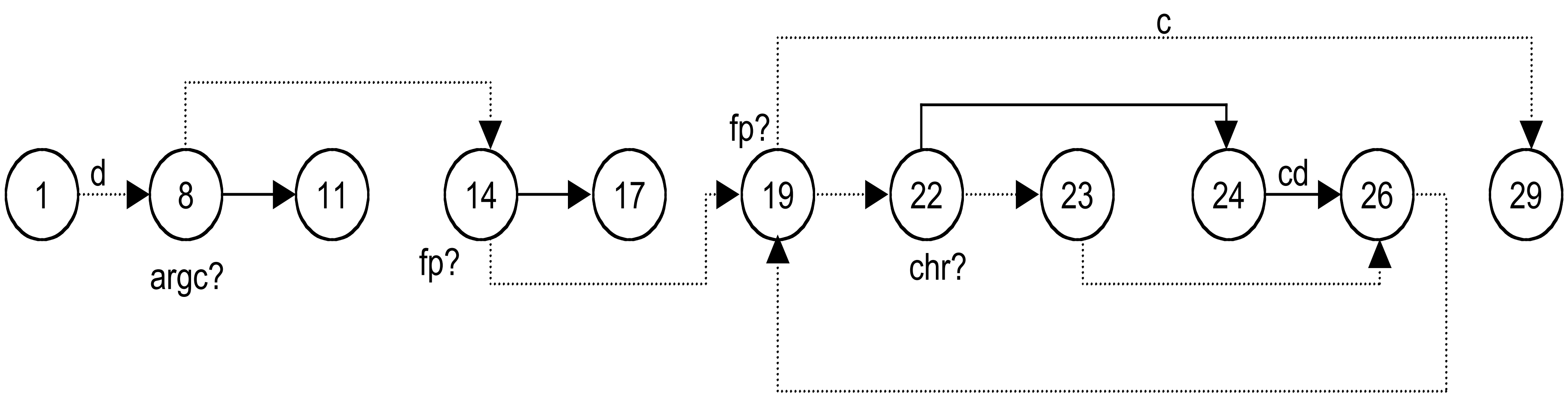
du-Path for argv[]



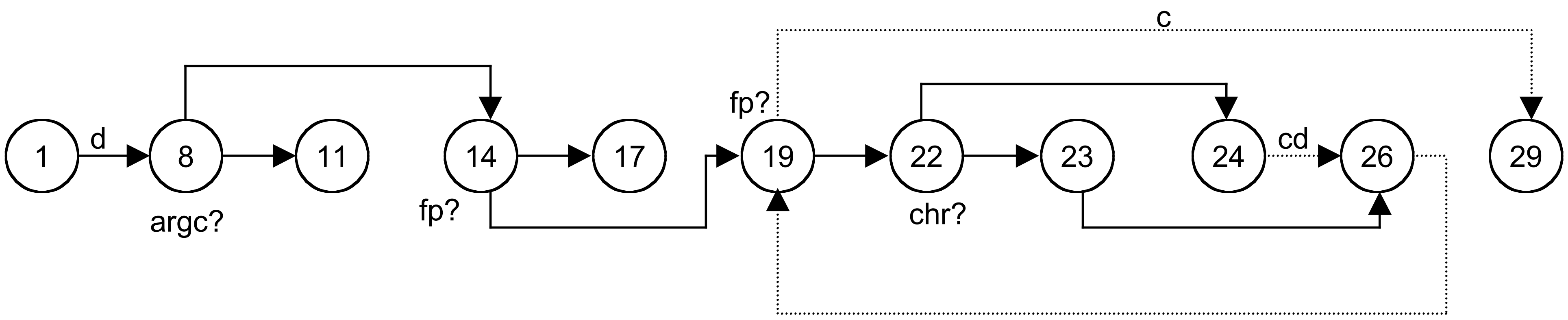
du-Path for numChars



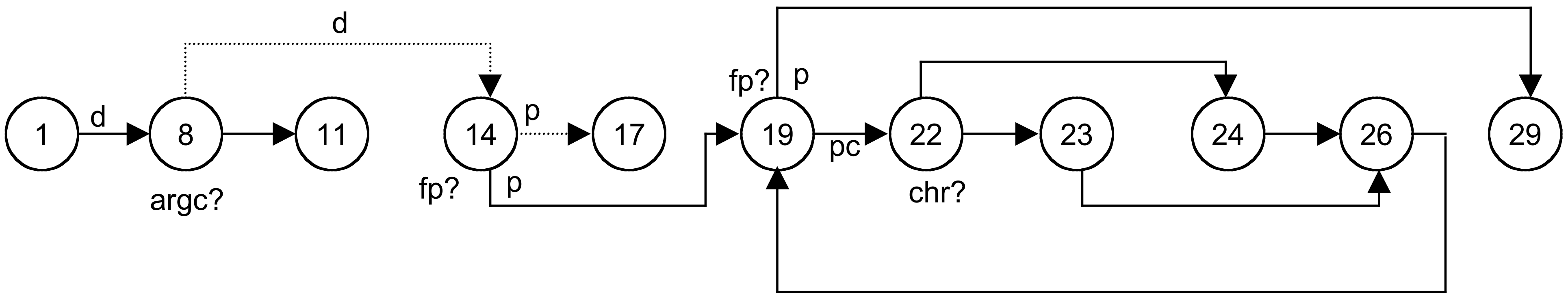
du-Path for numChars



du-Path for numChars



du-Path for fp



All Uses Strategy (AU)

- **AU** requires that <u>at least one path</u> from <u>every definition</u> of <u>every variable</u> to <u>every use</u> 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**.

All p-uses/Some c-uses Strategy (APU+C)

- **APU+C** requires that for every variable and every definition of that variable include at least one definition-free path from the definition to every predicate use.
- If there are definitions of the variable that are not covered by the above prescription, then add computational-use test cases to cover every definition.

All c-uses/Some p-uses Strategy (ACU+P)

- **ACU+P** requires that for every variable and every definition of that variable include at least one definition-free path from the definition to every computational use.
- If there are definitions of the variable that are not covered by the above prescription, then add predicate-use test cases to cover every definition.

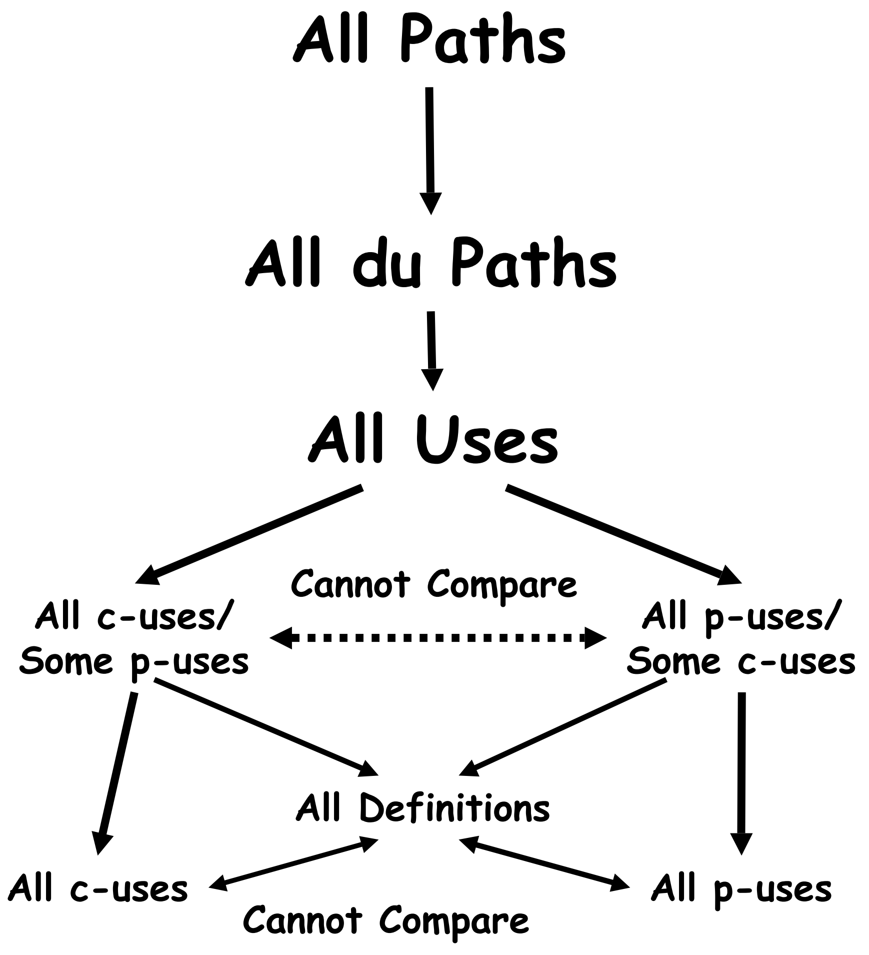
All Definitions Strategy (AD)

- **AD** requires that for every variable and every definition of that variable include at least one definition-free path from the definition to a computational or predicate use.
- **AD** < **ACU+P** and **AD** < **APU+C**.

All p-uses (APU)
All c-uses (ACU)

- **APU** is the same as **APU+C** without the **C** requirement.
- **APU** < **APU+C**.
- **ACU** is the same as **ACU+P** without the **P** requirement.
- **ACU** < **ACU+P**.

Relative Strength of Data-Flow Testing Strategies



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.