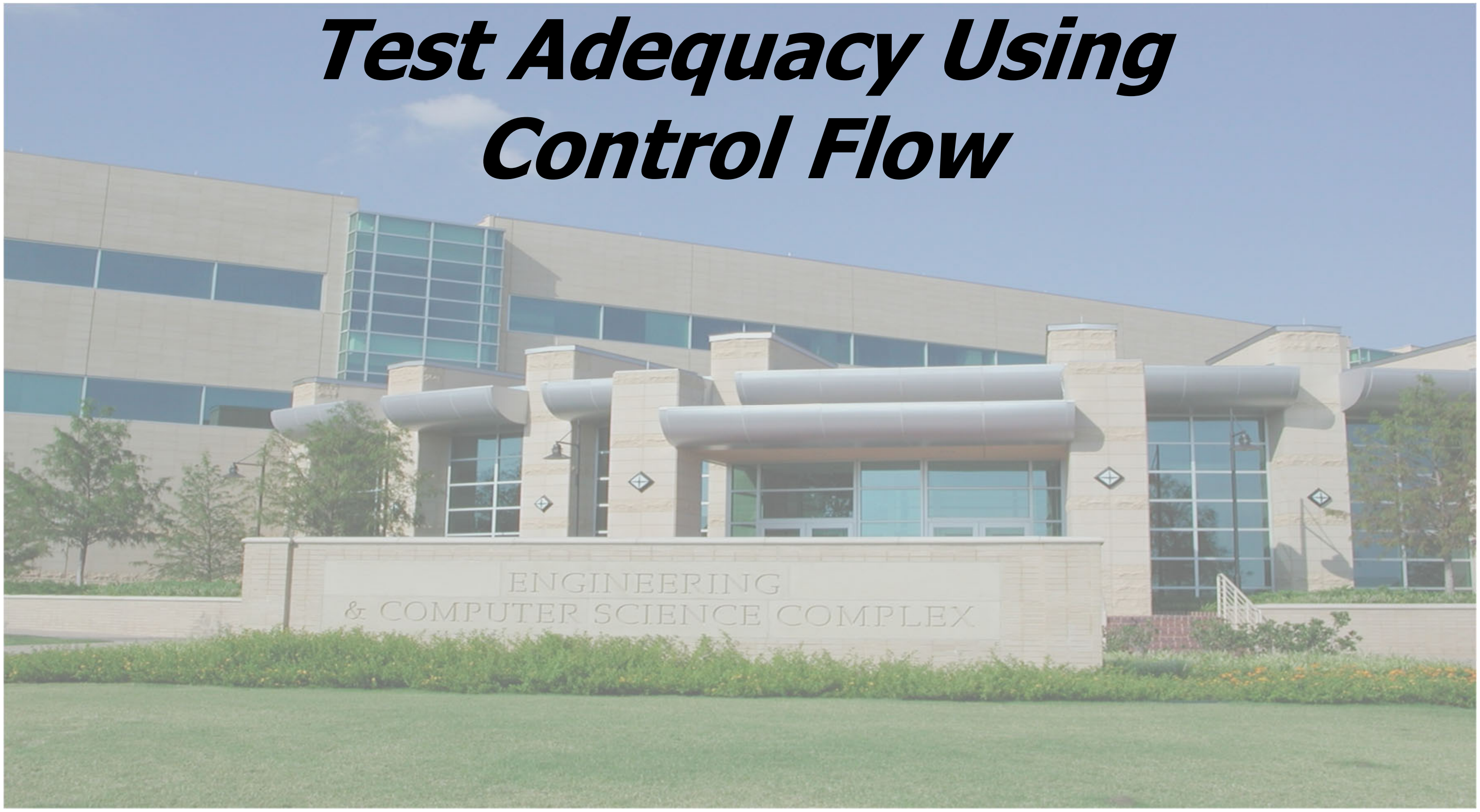
Test Adequacy Using Control Flow



Dr. Mark C. Paulk

SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

Test Adequacy Assessment Topics

<u>Part III. Test Adequacy Assessment and Enhancement</u>

7. Test Adequacy Assessment Using Control Flow and Data Flow

- **Basic**
- **Adequacy criteria based on control flow**
- **Concepts from data flow**
- **Adequacy criteria based on data flow**
- **Control flow versus data flow**
- **The “subsumes” relation**
- **Structural and functional testing**
- **Scalability of coverage measurement**
- **Tools**

8. Test Adequacy Assessment Using Program Mutation

Statement Coverage

The statement coverage of T with respect to (P,R) is computed as $|S_c| / (|S_e| - |S_i|)$

- **$|S_c|$ is the number of statements covered**
- **$|S_e|$ is the total number of statements in the program**
- **$|S_i|$ is the number of unreachable statements**

T is considered adequate with respect to the statement coverage criterion if the statement coverage of T with respect to (P,R) is 1.

Mathur, Example 7.10

Program P7.4

```
~~1 begin~~
2   int x, y;
3   int z;
4   input (x,y); z=0;
5   if (x<0 and y<0) {
6      z=x*x;
7      if (y≥0) z=z+1;
~~8    }~~
~~9   else~~
10     z=x*x*x;
11 output (z);
1~~2 end~~
```

Coverage domain

S_e = {2, 3, 4, 4b, 5, 6, 7, 7b, ~~9~~, 10, 11}

Let T_1 = {t_1: <x=-1,y=-1>, t_2:<x=1,y=1>}

Statements covered

- **t_1: 2, 3, 4, 4b, 5, 6, 7, 11**
- **t_2: 2, 3, 4, 4b, 5, ~~9~~, 10, 11**

$|S_c|$ = 9, $|S_i|$ = 1, $|S_e|$ = 10

- **7b is unreachable**

Statement coverage for T_1 is

$|S_c| / (|S_e| – |S_i|)$ = 9 / (10 - 1) = 1.00

T_1 is adequate for (P,R) with respect to the statement coverage criterion.

Syntactical Markers in Statement Coverage

Note that in Mathur’s Example 7.10, there are some physical lines that are not counted: those that consist of a syntactical marker.

- **begin**
- **}**
- **end**

Note that else is a syntactical marker that is covered and should not be!

4b is split out of line 4 because of z=0 (note 7b)

Counting Statements in SE 4367

Logical statements, such as as 4b and 7b, should always be separately counted.

- **I will not give you any problems with multiple logical lines of code on a physical line or the "if (y≥0) {" treatment of {.**

Counting syntactical markers, or not, is an operational definition choice.

- **If syntactical markers are counted, they should be treated as statements and all counted.**
  - **This means that begin, {, }, else, end, … should be counted.**
  - **Lines 1, 8, 9, and 12 would be counted in Program 7.4.**
- **If syntactical markers are not counted, none should be counted… including else.**
- **Do not count syntactical markers in SE 4367.**

An IF Statement Example

```
 Program P1
1) integer A, B;
2) input (A);
3) if (A == 0)
~~4) {~~
5)       B = A + 1;
~~6) }~~
~~7) else~~
~~8) {~~
9)       B = A - 2;
~~10) }~~
11)output (A,B);
~~12)end;~~
```

T = {t_1: 0}

t_1: 1, 2, 3 (t), 5, 11

D_S = {~~1, 2, 3, 5~~, 9, ~~11~~}

C_S = 5/6 = 83%

To achieve 100% statement coverage, need a test case with A≠0, t_2: 1

A WHILE Statement Example

```
 Program P2
1) integer A, B;
2) input (A);
3) B = 1;
4) while (int i=1;
   i<=A; i++)
~~5) {~~
6)     B = B * i;
~~7) }~~
8) output (A,B);
~~9) end;~~
```

$T = \{t_1: 2\}$

t_1: 1, 2, 3, 4 (t), 6, 4 (t), 6, 4 (f), 8

D_S = {~~1, 2, 3, 4, 6, 8~~}

$C_S = 6 / 6 = 100\%$

Another Statement Example

Program P3

```
1)    integer A, B;
2)    input (A);
3)    if (A > 7)
4)          B = 1;
~~5)    else~~
~~6)    {~~
7)          B = 2;
8)          if (A < 2)
9)                B = 3;
~~10)       } // end else A>7~~
11)   while (int i=1; i<=A; i++)
~~12)   {~~
13)         if (B<0)
14)               B = B + 4;
~~15)         else~~
16)               B = B + 5;
~~17)   } // end for loop~~
18)   output (A,B);
~~19)   end;~~
```

$T = \{t_1$: 1, t_2: 0}

t_1: 1, 2, 3 (f), 7, 8 (t), 9, 11 (t), 13 (f), 16, 11 (f), 18
t_2: 1, 2, 3 (f), 7, 8 (t), 9, 11 (f), 18

D_S = {~~1, 2, 3~~, 4, ~~7,~~ ~~8, 9, 11, 13~~, 14, ~~16, 18~~}

Note that B<0 can never be true…

C_S = 10 / (12-1) = 91%

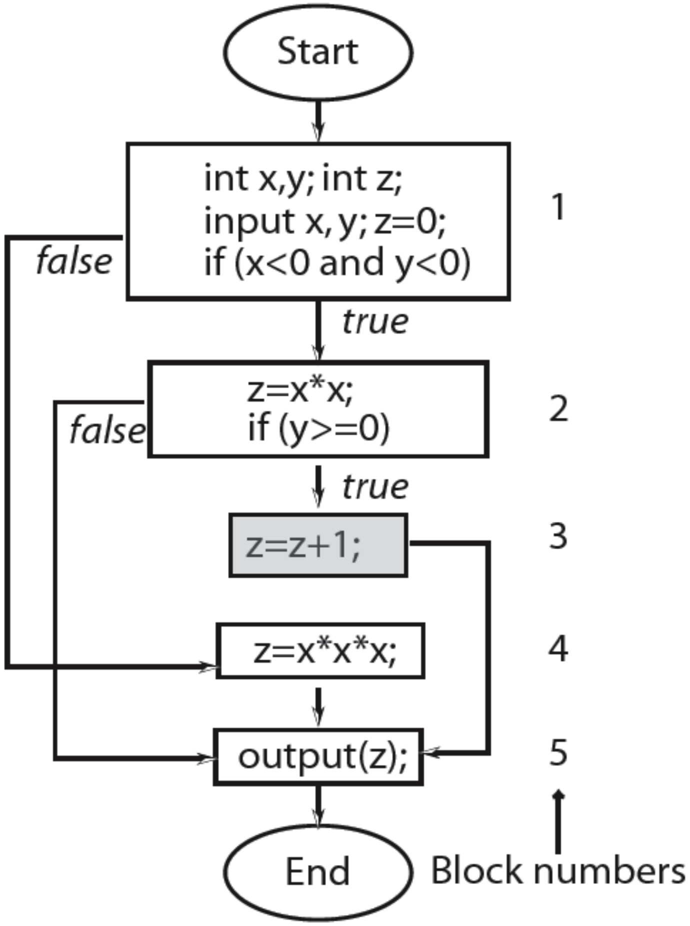
Block Coverage

The block coverage of T with respect to (P,R) is computed as $|B_c| / (|B_e| - |B_i|)$

- **$|B_c|$ is the number of blocks covered**
- **$|B_e|$ is the total number of blocks in the program**
- **$|B_i|$ is the number of unreachable blocks**

T is considered adequate with respect to the block coverage criterion if the statement coverage of T with respect to (P,R) is 1.

Mathur, Example 7.11



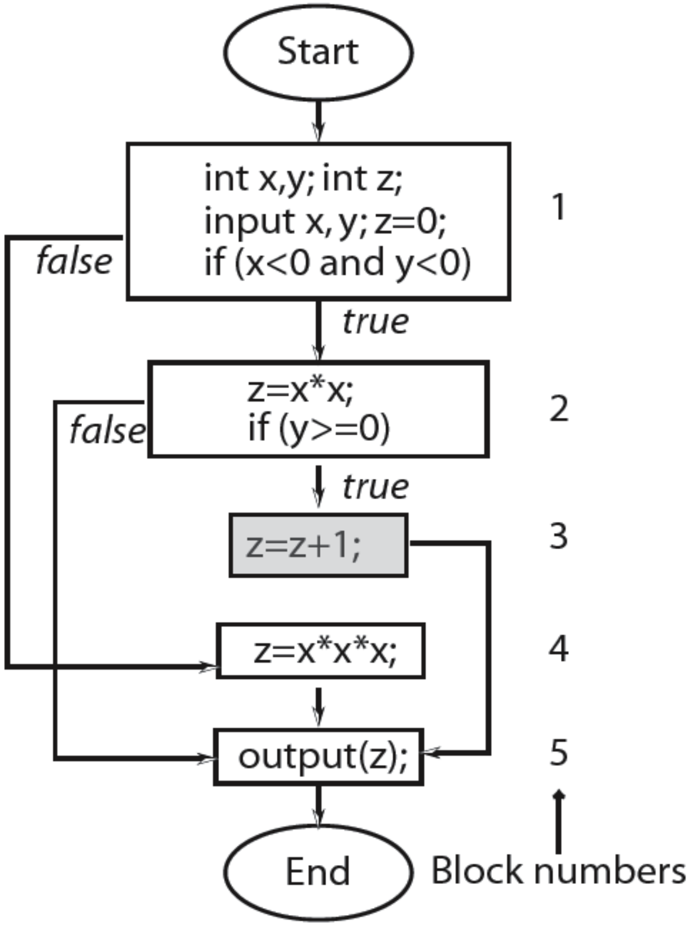
Coverage domain

- **B_e = {1, 2, 3, 4, 5}**

T_2 = {t_1: <x=-1,y=-1>,
t_2: <x=-3,y=-1>,
t_3: <x=-1,y=-3>}

Blocks covered:

- **t_1, t_2, t_3: blocks 1, 2, 5**



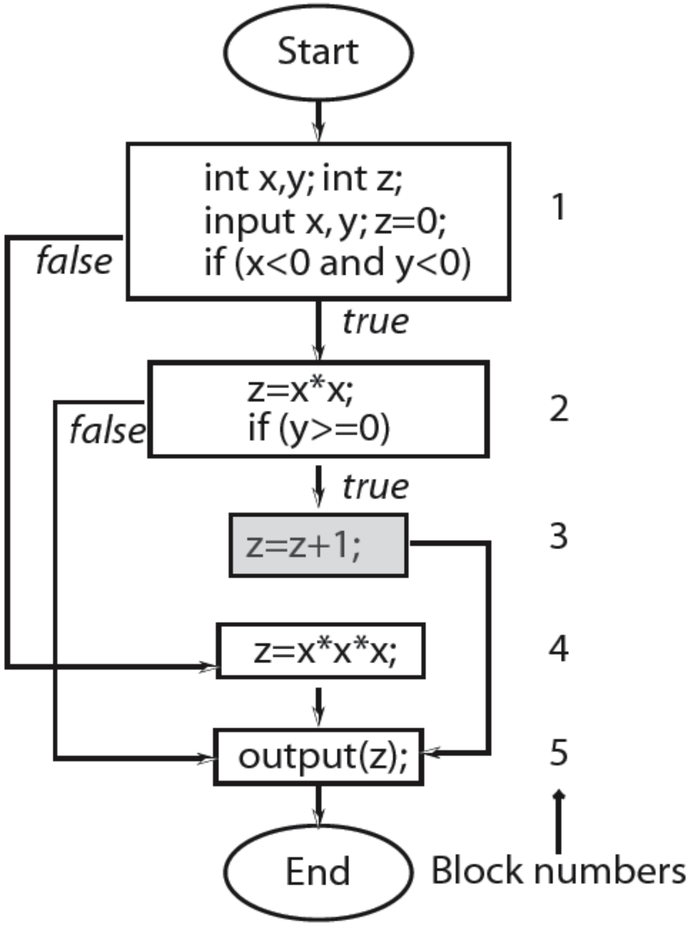
$|B_e| = 5, |B_c| = 3, |B_i| = 1$

- **the condition in block 2 will never be true**
  - **y<0 and y>=0**
- **block 3 is unreachable**

Block coverage for T_2

$|B_c| / (|B_e| - |B_i|) = 3 / (5 - 1) = 0.75$

T_2 is not adequate for (P,R) with respect to the block coverage criterion.



T_1 is adequate with respect to block coverage criterion.

Verify this statement.

If test t_2 in T_1 is added to T_2, we obtain a test set adequate with respect to the block coverage criterion for this program.

Verify this statement.

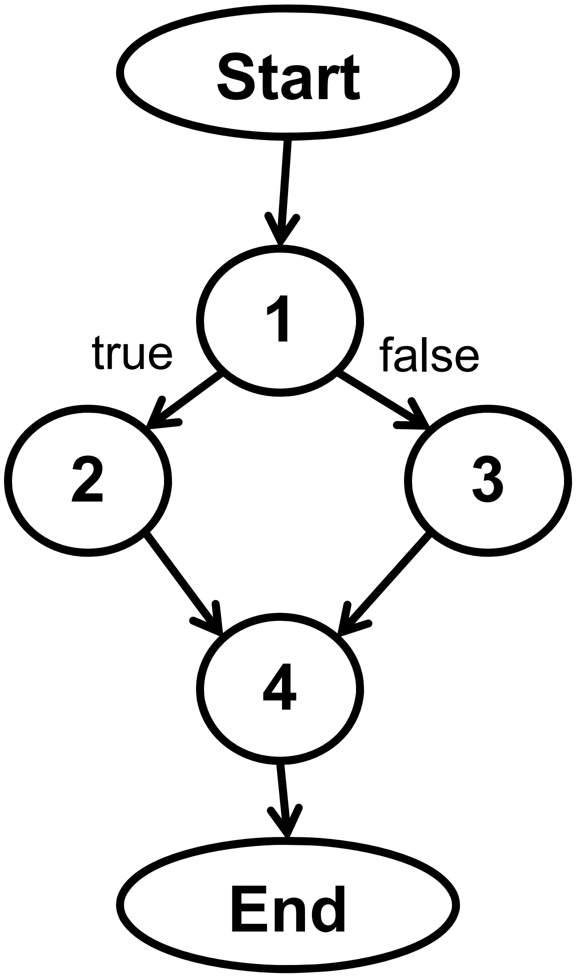
An IF Block Example

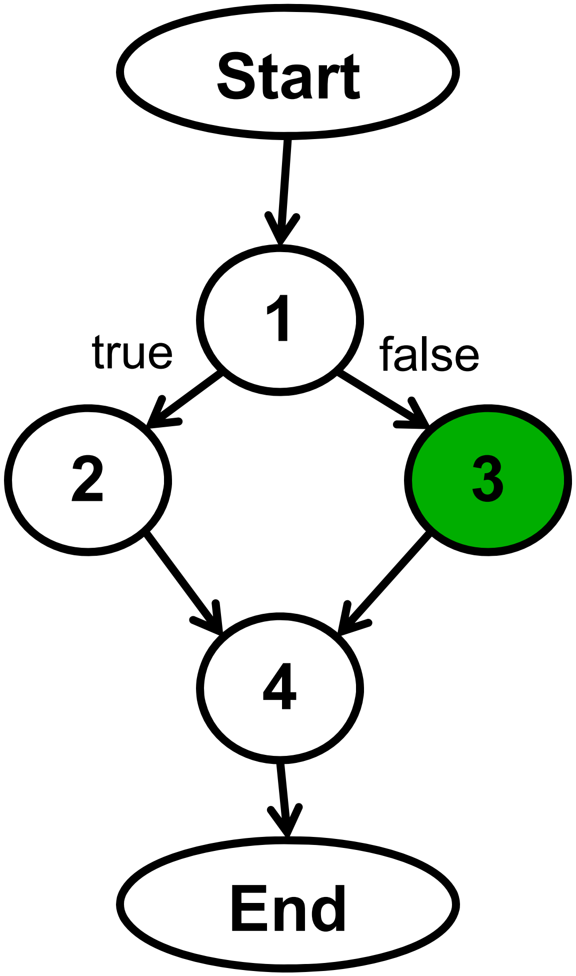
Program P1

```
1) integer A, B;
2) input (A);
3) if (A == 0)
~~4) {~~
5)        B = A + 1;
~~6) }~~
~~7) else~~
~~8) {~~
9)        B = A - 2;
~~10) }~~
11) output (A,B);
~~12) end;~~
```

Basic blocks

1 – 1, 2, 3 (4)
2 – 5 (6, 7, 8)
3 – 9 (10)
4 – 11 (12)





$T = \{t_1: 0\}$

t_1: 1 (t), 2, 4

D_B = {~~1, 2~~, 3, ~~4~~}

$C_B = 3 / 4 = 75\%$

To achieve 100% statement coverage, need a test case with $A \neq 0$, t_2: 1

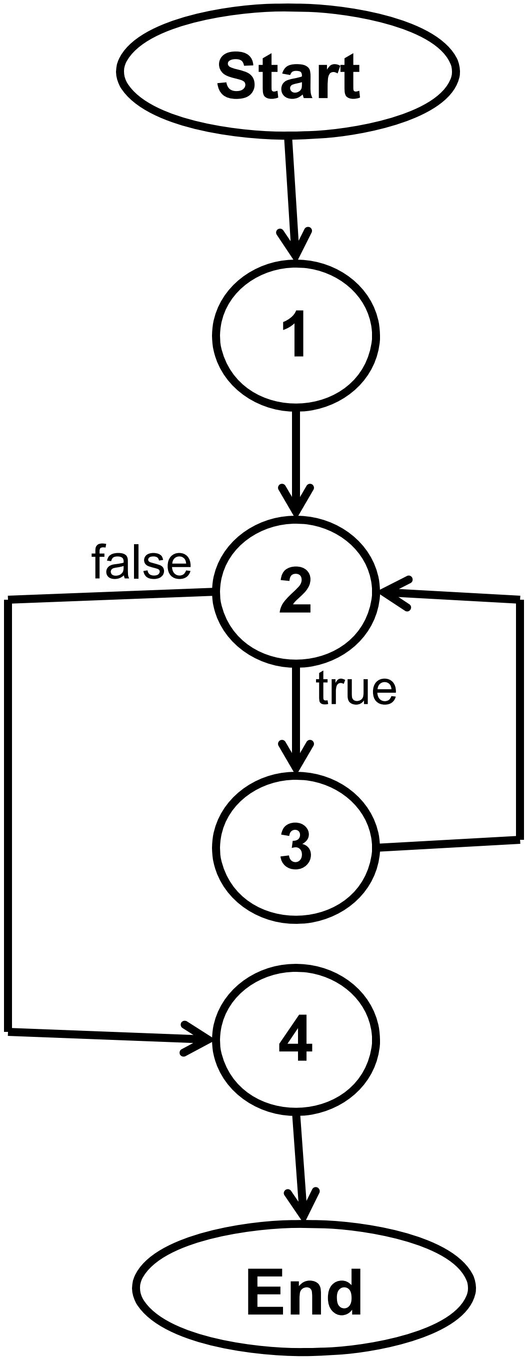
A WHILE Block Example

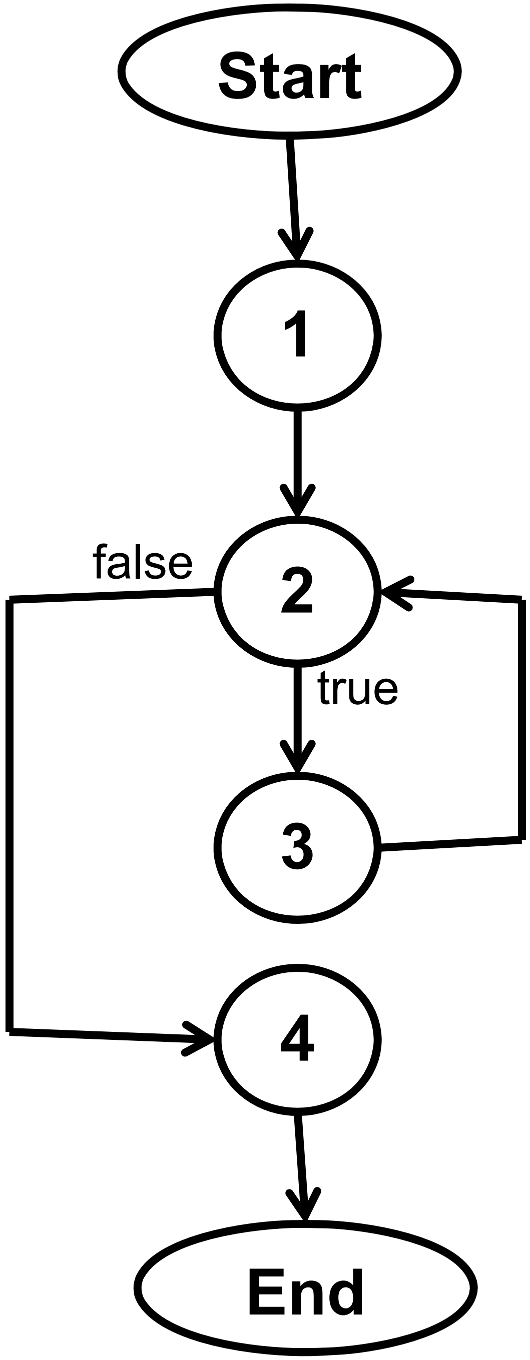
Program P2

```
1)  integer A, B;
2)  input (A);
3)  B = 1;
4)  while (int i=1;
    i<=A; i++)
~~5) {~~
6)       B = B * i;
~~7) }~~
8)  output (A,B);
~~9) end;~~
```

Basic blocks

1 – 1, 2, 3
2 – 4 (5)
3 – 6 (7)
4 – 8 (9)





T = {t_1: 2}

For statements

- t_1: 1, 2, 3, 4 (t), 6, 4 (t), 6, 4 (f), 8

  Basic blocks

  1 – 1, 2, 3

  2 – 4 (5)

  3 – 6 (7)

  4 – 8 (9)

For blocks

t_1: 1, 2 (t), 3, 2 (t), 3, 2 (f), 4

D_B = {~~1, 2, 3, 4~~}

C_B = 4 / 4 = 100%

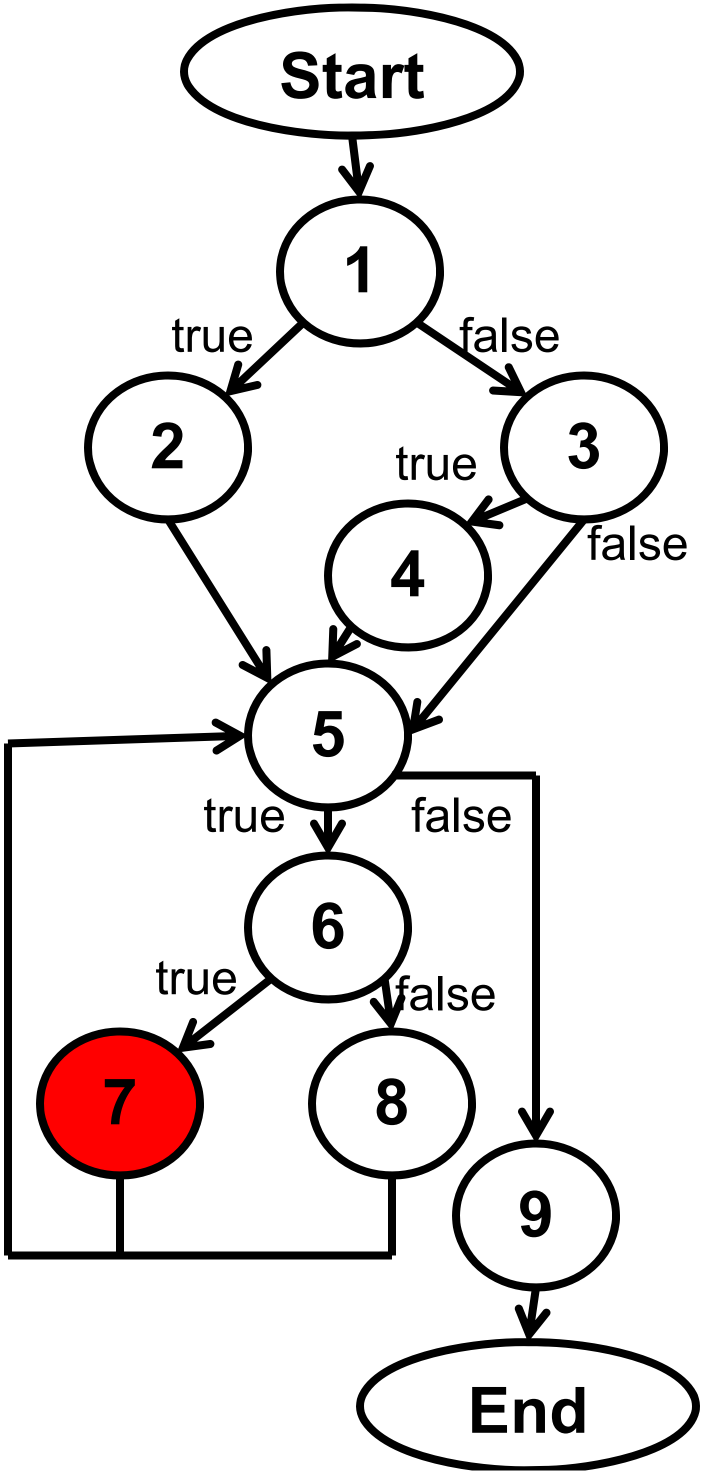
Another Block Example

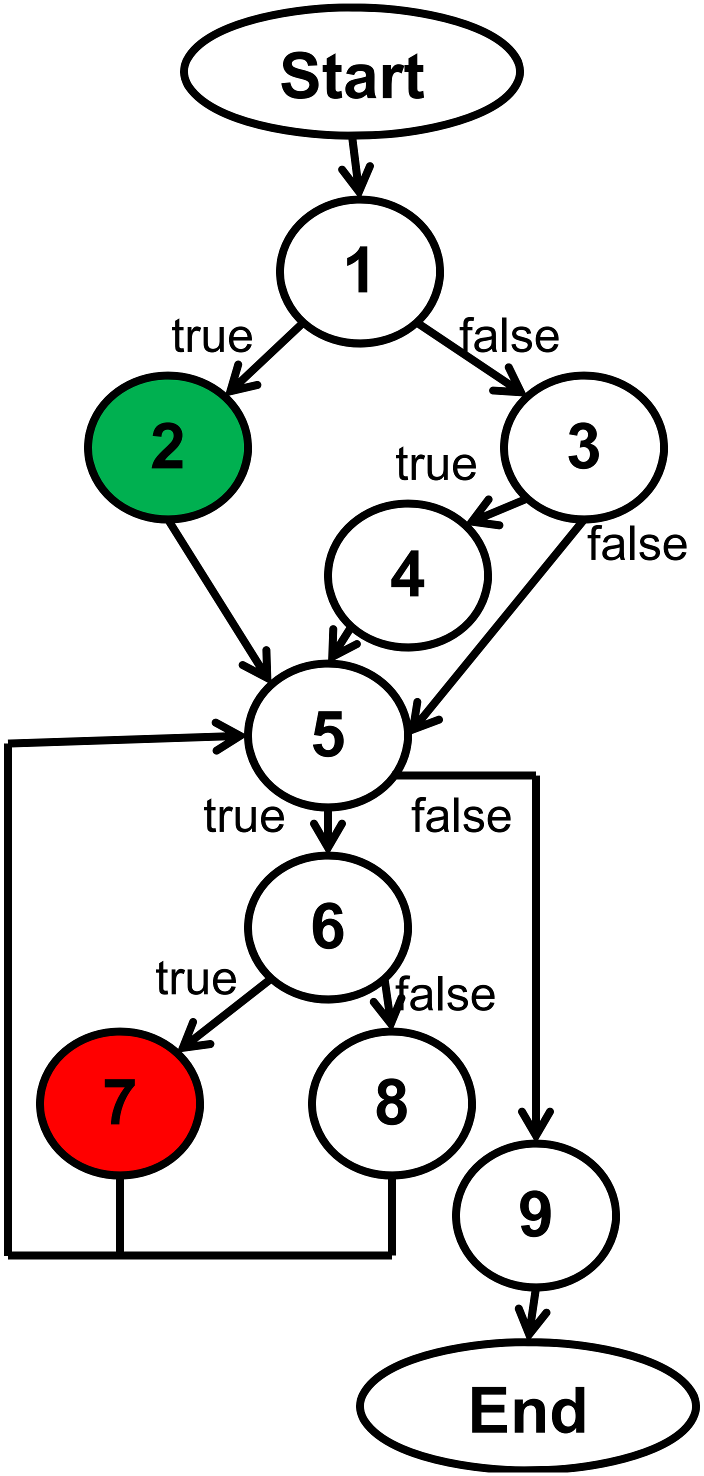
Program P3

```
1)    integer A, B;
2)    input (A);
3)    if (A > 7)
4)          B = 1;
~~5)    else~~
~~6)    {~~
7)          B = 2;
8)          if (A < 2)
9)                B = 3;
~~10)       } // end else A>7~~
11)   while (int i=1; i<=A;
      i++)
~~12)   {~~
13)         if (B<0)
14)               B = B + 4;
~~15)         else~~
16)               B = B + 5;
~~17)   } // end for loop~~
18)   output (A,B);
~~19)   end;~~
```

Basic blocks

1 – 1, 2, 3
2 – 4 (5, 6)
3 – 7, 8
4 – 9 (10)
5 – 11 (12)
6 – 13
7 – 14 (15)
8 – 16 (17)
9 – 18 (19)





$T = \{t_1\text{: } 1,\ t_2\text{: } 0\}$

t_1: 1, 2, 3 (f), 7, 8 (t), 9, 11 (t), 13 (f), 16, 11 (f), 18
t_2: 1, 2, 3 (f), 7, 8 (t), 9, 11 (f), 18

<u>Basic blocks</u>
1–1,2,3 2–4 3–7,8 4–9 5–11 6–13
7–14 8–16 9–18

D_B = {~~1,~~ 2, ~~3, 4, 5, 6,~~ 7, ~~8, 9~~}

Note that B<0 can never be true…

$C_S = 7 \,/\, (9\text{-}1) = 88\%$

Conditions

Any expression that evaluates to true or false constitutes a condition.

- **also known as a predicate**

Given that A and B are Boolean variables and x and y are integers

- **A**
- **x > y**
- **A OR B**
- **A AND (x < y)**
- **A AND B**

are examples of conditions

Conditions in Programming Languages

Note that in some programming languages…

x and x + y are valid conditions

The constants 1 and 0 may correspond to, respectively, true and false

The constant 0 may correspond to false and any non-zero constant corresponds to true

Odd constants may correspond to true and even constants may correspond to false

Simple and Compound Conditions

A simple condition does not use any Boolean operators except for the NOT operator.

- **made up of variables**
- **at most one relational operator from the set {<, ≤, >, ≥, ==, ≠}**

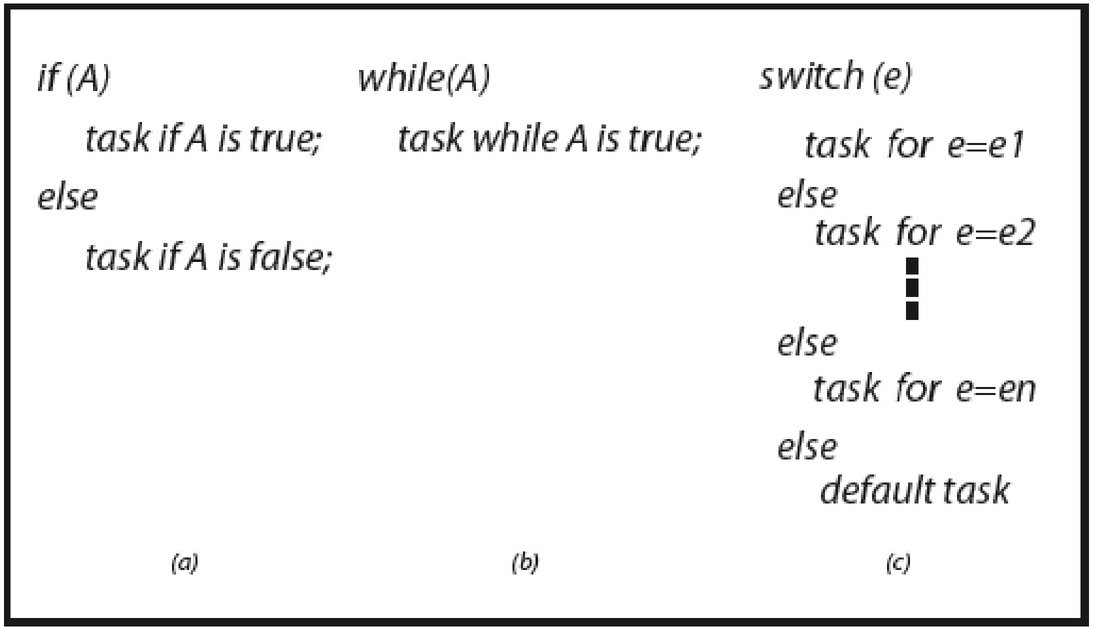
Simple conditions are also referred to as atomic or elementary conditions because they cannot be parsed any further into two or more conditions.

A compound condition is made up of two or more simple conditions joined by one or more Boolean operators.

Conditions as Decisions

Any condition can serve as a decision in an appropriate context within a program.

Most high level languages provide if, while, and switch statements to serve as contexts for decisions.



```
if (A)
    task if A is true;
else
    task if A is false;
```

(a)

```
while(A)
    task while A is true;
```

(b)

```
switch (e)
    task for e=e1
  else
    task for e=e2
          ⋮
  else
    task for e=en
  else
    default task
```

(c)

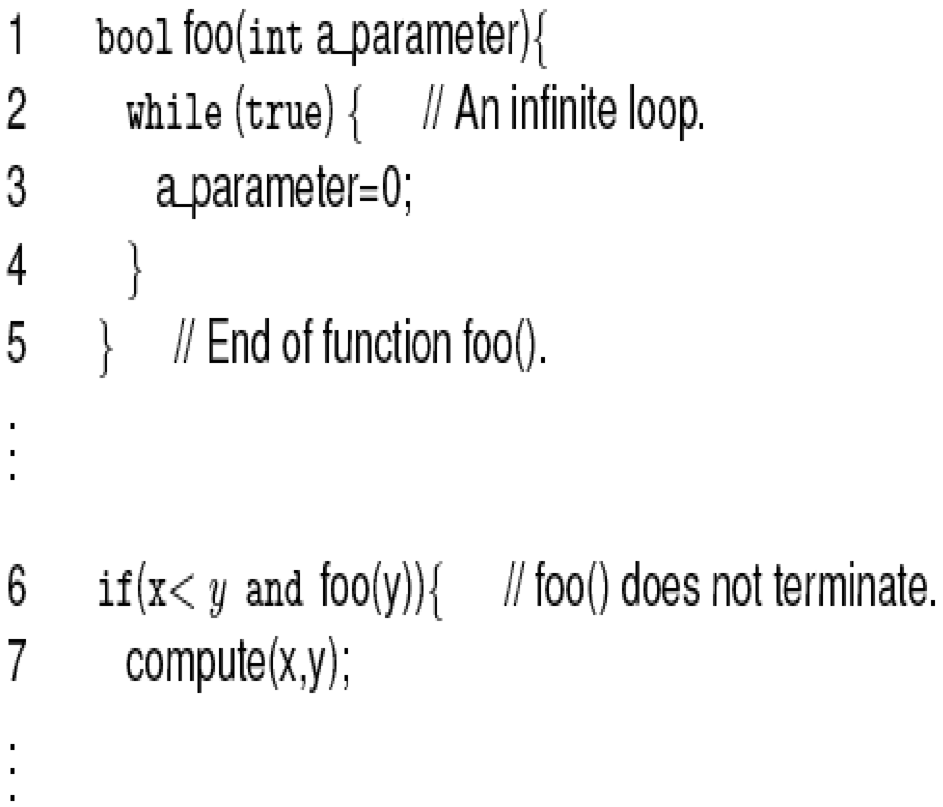
Outcomes of a Decision

A decision can have three possible outcomes, true, false, and undefined.

When the condition corresponding to a decision evaluates to true or false, the decision to take one or the other path is made.

In some cases the evaluation of a condition might fail, in which case the corresponding decision's outcome is undefined.

Undefined Condition



```
1   bool foo(int a_parameter){
2     while (true) {     // An infinite loop.
3       a_parameter=0;
4     }
5   }    // End of function foo().
⋮

6   if(x< y and foo(y)){     // foo() does not terminate.
7     compute(x,y);
⋮
```

The condition inside the if statement at line 6 will remain undefined because the loop at lines 2-4 will never terminate.

The decision at line 6 evaluates to undefined.

- **unreachable**
- **dead code**

Would it make more sense to characterize this as infeasible?

Coupled Conditions

How many simple conditions are there in the compound condition

- X = (A AND B) OR (C AND A)

The first occurrence of A is said to be coupled to its second occurrence.

Does X contain three or four simple conditions?

- Both answers are correct depending on one's point of view.
- There are three distinct conditions A, B, and C.
- The answer is four when one is interested in the number of occurrences of simple conditions in a compound condition.

Conditions Within Assignments

A = x < y;

- **a simple condition assigned to Boolean variable A**

X = P OR Q;

- **a compound condition assigned to Boolean variable X**

x = y + z * s; if (x) …

- **condition true if x = 1, false otherwise**

A = x < y; x = A * B

- **A is used in a subsequent expression for x but not as a decision**

Conditions vs Decisions

Strictly speaking, a condition becomes a decision only when it is used in the appropriate context, such as within an if statement.

a = x < y is not a decision

if (x<y) then … is a decision

x = a * b is not a decision

while (x) … is a decision

Decision Coverage

A decision is considered covered if the flow of control has been diverted to all possible destinations that correspond to this decision.

- **all outcomes of the decision have been taken**

This implies that, for example, the expression in an if or while statement has evaluated to true in some execution of the program under test and to false in the same or another execution.

Decision coverage is also known as branch decision coverage.

Switch Statement

A decision implied by the switch statement is considered covered if during one or more executions of the program under test, the flow of control has been diverted to all possible destinations.

Mathur, Example 7.13

Program P7.7

```
begin
    int x, y;
    input (x);
    if (x < 0)
        x = -x;
        z = foo1 (x);
    output (z);
end
```

This program inputs an integer x, and if necessary, transforms it into a positive value before invoking foo1 to compute the output z.

The program has an error.

The program is supposed to compute z using foo2 when $x \geq 0$

Program P7.7

```
begin
    int x, y;
    input (x);
    if (x < 0)
        x = -x;
        z = foo1 (x);
    output (z);
end
```

Consider the test set

- **T = {t_1: <x=-5>}**

It is adequate with respect to statement and block coverage criteria but does not reveal the error.

Another test set

- **T’ = {t_1: <x=-5>, t_2: <x=3>}**

does reveal the error.

It covers the decision where T does not. Confirm this…

Decision Coverage

The decision coverage of T with respect to (P,R) is computed as $|D_c| / (|D_e| - |D_i|)$

- **$|D_c|$ is the number of decisions covered**
- **$|D_e|$ is the total number of decisions in the program**
- **$|D_i|$ is the number of infeasible decisions**

T is considered adequate with respect to the decision coverage criterion if the decision coverage of T with respect to (P,R) is 1.

Decision Coverage Domain

The domain of decision coverage consists of all decisions in the program under test.

Each if and each while contributes to one decision.

A switch contributes to more than one decision.

Another Decision Example

```
 Program P3
1)    integer A, B;
2)    input (A);
3)    if (A > 7)
4)          B = 1;
5)    else
6)    {
7)          B = 2;
8)          if (A < 2)
9)                B = 3;
10)    } // end else A>7
11)   while (int i=1; i<=A; i++)
12)   {
13)         if (B<0)
14)               B = B + 4;
15)         else
16)               B = B + 5;
17)   } // end for loop
18)   output (A,B);
19)   end;
```

T = {t_1: 1, t_2: 0}

t_1: 1, 2, 3 (f), 7, 8 (t), 9, 11 (t), 13 (f), 16, 11 (f), 18
t_2: 1, 2, 3 (f), 7, 8 (t), 9, 11 (f), 18

D_D = {line 3, line 8, line 11, line 13}

Note that line 13) B<0 can never be true…

	t_1	t_2
line 3	**f**	**f**
line 8	**t**	**t**
line 11 (loop)	**t/f**	**f**
line 13	**f**	**--**

C_D = 1 / (4-1) = 33%

```
 Program P3
1)   integer A, B;
2)   input (A);
3)   if (A > 7)
4)        B = 1;
5)   else
6)   {
7)        B = 2;
8)        if (A < 2)
9)             B = 3;
10)   } // end else A>7
11)  while (int i=1; i<=A; i++)
12)  {
13)       if (B<0)
14)             B = B + 4;
15)       else
16)             B = B + 5;
17)  } // end for loop
18)  output (A,B);
19)  end;
```

To enhance T = {t_1: 1, t_2: 0}

To cover the decision at line 3 (A>7 true), A=8

To cover the decision at line 8, A≤7 and A<2 false, A=5

T’ = {1, 0, 8, 5}

C_D = 3 / (4-1) = 100%

Conditions and Decisions

A decision can be composed of a simple condition such as x < 0 or of a more complex condition, such as ((x < 0 AND y < 0) OR (p ≥ q)).

AND, OR, XOR are the logical operators that connect two or more simple conditions to form a compound condition.

A simple condition is considered covered if it evaluates to true and false in one or more executions of the program in which it occurs.

A compound condition is considered covered if each simple condition it is comprised of is also covered.

Condition Coverage

The condition coverage of T with respect to (P,R) is computed as $|C_c| / (|C_e| - |C_i|)$

- **$|C_c|$ is the number of simple conditions covered**
- **$|C_e|$ is the total number of simple conditions in the program**
- **$|C_i|$ is the number of infeasible simple conditions**

T is considered adequate with respect to the condition coverage criterion if the condition coverage of T with respect to (P,R) is 1.

An Alternate Formula for Condition Coverage

An alternate formula where each simple condition contributes 2, 1, or 0 to C_c depending on whether it is

- **covered**
- **partially covered**
- **not covered**

respectively, is $|C_c| / (2 * (|C_e| – |C_i|))$

We will not use this alternate formula in SE 4367.

Compound Conditions

Decision coverage is concerned with the coverage of decisions regardless of whether a decision corresponds to a simple or a compound condition.

In the statement

```
1  if (x < 0 AND y < 0)
2       z = foo (x, y);
```

there is only one decision that leads control to line 2

- **if the compound condition inside the if evaluates to true**

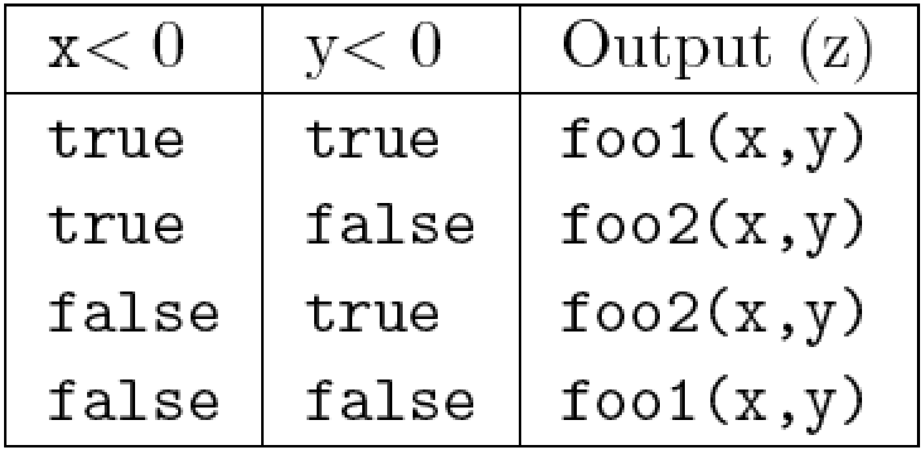
A compound condition might evaluate to true or false in one of several ways.

Mathur, Example 7.14

Program P7.8

```
begin
    int x, y, z;
    input (x, y);
    if (x < 0 and y < 0)
        z = foo1 (x, y);
    else
        z = foo2 (x, y);
    output (z);
    end
```

Partial specification for computing z



x< 0	y< 0	Output (z)
true	true	foo1(x,y)
true	false	foo2(x,y)
false	true	foo2(x,y)
false	false	foo1(x,y)

Note that for $x \geq 0$ AND $y \geq 0$, Program P7.8 incorrectly computes z as foo2 (x, y).

Program P7.8

```
begin
    int x, y, z;
    input (x, y);
    if (x < 0 and y < 0)
        z = foo1 (x, y);
    else
        z = foo2 (x, y);
    output (z);
    end
```

Consider the test set:

T = {t_1: <x=-3,y=-2>,
t_2: <x=-4,y=2>}

Check that T is adequate with respect to the statement, block, and decision coverage criteria.

Verify that P7.8 behaves correctly against t_1 and t_2.

Condition coverage for T:
$|C_c| / (|C_e| – |C_i|) = 1 / (2 - 0) = 0.5$

Program P7.8

```
1    begin
2        int x, y, z;
3        input (x, y);
4        if (x < 0 and y < 0)
5            z = foo1 (x, y);
6        else
7            z = foo2 (x, y);
8        output (z);
9        end
```

Add the following test case to T:

t_3: <x=3, y=4>

Check that the enhanced test set T is adequate with respect to the condition coverage criterion and possibly reveals an error in the program.

Under what conditions will a possible error at line 7 be revealed by t_3?

- **foo1(3,4) ≠ foo2(3,4)**

Mathur, Example P7.21

```
1) begin
2)    float x=0, p, e, d=1, c, t;
3)    input (p,e);
4)    c=2*p;
5)    if (c≥2) {
6)       output(“Error”);
7)    }
8)    else {
9)       while (d>e) {
10)         d=d/2; t=c-(2*x+d);
11)         if (t≥0) {
12)            x=x+d;
13)            c=2*(c-(2*x+d));
14)         }
15)         else {
16)            c=2*c;
17)         }
18)      }
19)      output(“SQRT of p=“, x);
20)   }
21) end
```

Square root for 0≤p<1 to accuracy e

T={<0.5, 0.001>, <3.0, 0.001>}

Statement coverage?

Decision coverage?

Condition coverage?

Note: added float t to example.

Note: lines 12 and 13 must be interchanged to have a correct version of this program (Mathur).

Mathur, Example P7.21, Statement Coverage

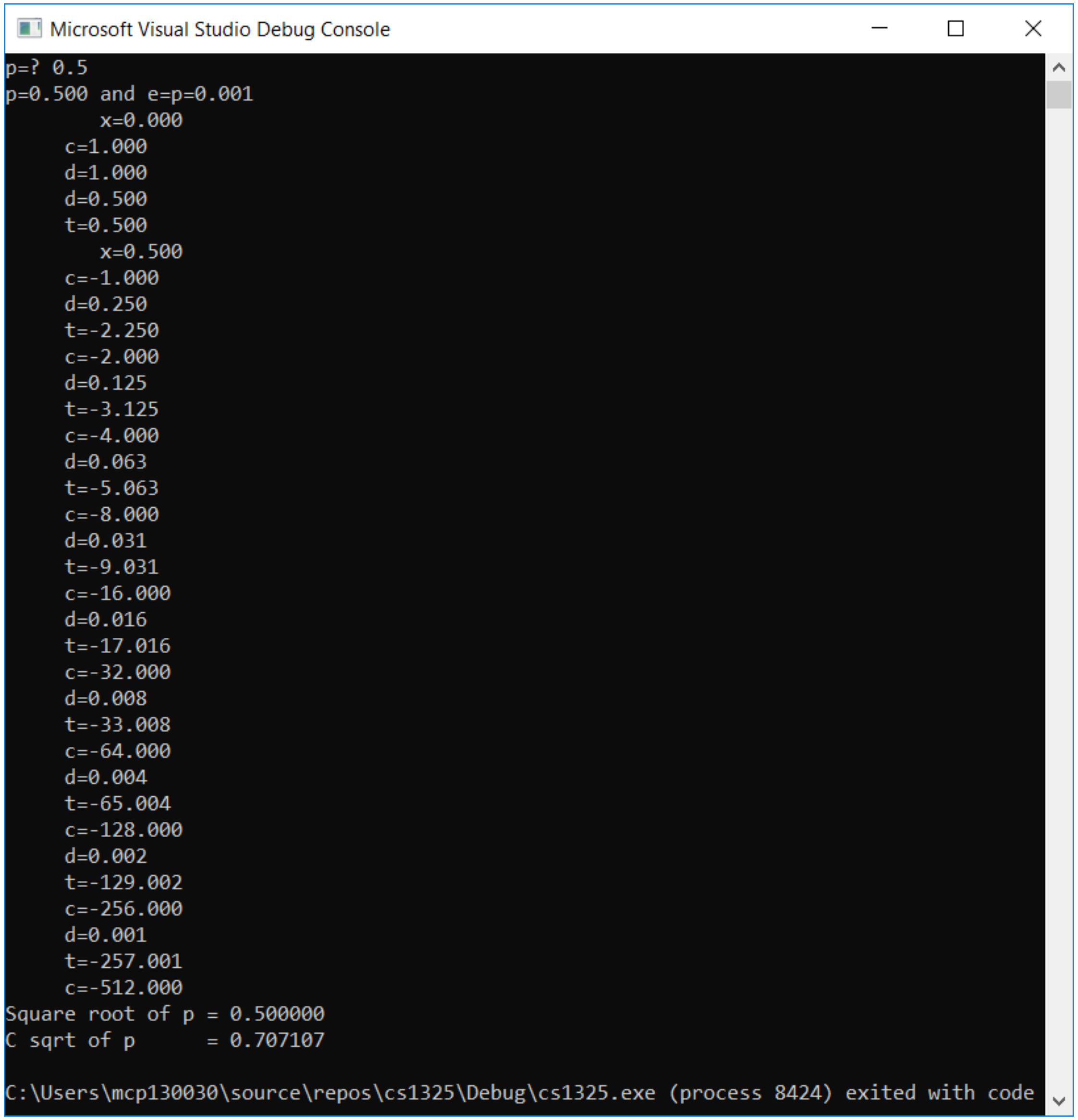
```
~~1) begin~~
2)    float x=0, p, e, d=1, c, t;
3)    input (p,e);
4)    c=2*p;
5)    if (c≥2) {
6)       output(“Error”);
~~7)    }~~
~~8)    else {~~
9)       while (d>e) {
10)         d=d/2; t=c-(2*x+d);
11)         if (t≥0) {
12)            x=x+d;
13)            c=2*(c-(2*x+d));
~~14)        }~~
~~15)        else {~~
16)            c=2*c;
~~17)        }~~
~~18)     }~~
19)       output(“SQRT of p=“, x);
~~20)   }~~
~~21)end~~
```

<u>Domain</u>

**D_S = { 2, 3, 4, 5, 6,
9, 10a, 10b, 11, 12,
13, 16, 19}**

T={<0.5, 0.001>, <3.0, 0.001>}

```
~~1) begin~~
2)      float x=0, p, e, d=1, c, t;
3)      input (p,e);
4)      c=2*p;
5)      if (c≥2) {
6)         output("Error");
~~7)      }~~
~~8)      else {~~
9)         while (d>e) {
10)           d=d/2; t=c-(2*x+d);
11)           if (t≥0) {
12)              x=x+d;
13)              c=2*(c-(2*x+d));
~~14)          }~~
~~15)          else {~~
16)              c=2*c;
~~17)          }~~
~~18)       }~~
19)        output("SQRT of p=", x);
~~20)     }~~
~~21) end~~
```



```
p=? 0.5
p=0.500 and e=p=0.001
        x=0.000
     c=1.000
     d=1.000
     d=0.500
     t=0.500
        x=0.500
     c=-1.000
     d=0.250
     t=-2.250
     c=-2.000
     d=0.125
     t=-3.125
     c=-4.000
     d=0.063
     t=-5.063
     c=-8.000
     d=0.031
     t=-9.031
     c=-16.000
     d=0.016
     t=-17.016
     c=-32.000
     d=0.008
     t=-33.008
     c=-64.000
     d=0.004
     t=-65.004
     c=-128.000
     d=0.002
     t=-129.002
     c=-256.000
     d=0.001
     t=-257.001
     c=-512.000
Square root of p = 0.500000
C sqrt of p      = 0.707107

C:\Users\mcp130030\source\repos\cs1325\Debug\cs1325.exe (process 8424) exited with code
```

T={<0.5, 0.001>, <3.0, 0.001>}

```
~~1) begin~~
2)     float x=0, p, e, d=1, c, t;
3)     input (p,e);
4)     c=2*p;
5)     if (c≥2) {
6)        output("Error");
~~7)       }~~
~~8)     else {~~
9)        while (d>e) {
10)          d=d/2; t=c-(2*x+d);
11)          if (t≥0) {
12)             x=x+d;
13)             c=2*(c-(2*x+d));
~~14)            }~~
~~15)         else {~~
16)             c=2*c;
~~17)            }~~
~~18)         }~~
19)       output("SQRT of p=", x);
~~20)     }~~
~~21) end~~
```

Trace of t_1: 2, 3 (0.5, 0.001),
4, 5 (1≥2 f),
9 (1.0>0.001 t), 10a, 10b,
11 (0.5≥0 t), 12, 13,
9 (0.5>0.001 t), 10a, 10b,
11 (-2.25≥0 f), 16,
9(0.25>0.001 t), 10a, 10b,
11(-3.125≥0 f), 16, …
9(0.002>0.001 t), 10a, 10b,
11(-257.001≥0 f), 16,
9(0.001>0.001) f), 19

D_S = { ~~2, 3, 4, 5,~~ 6, ~~9, 10a, 10b, 11, 12, 13, 16, 19~~}

Cover line 6 in t_2:
C_S = 13/13 = 100%

Mather, Example P7.21, Decision and Condition Coverage

```
1)  begin
2)      float x=0, p, e, d=1, c, t;
3)      input (p,e);
4)      c=2*p;
5)      if (c≥2) {
6)          output(“Error”);
7)      }
8)      else {
9)          while (d>e) {
10)             d=d/2; t=c-(2*x+d);
11)             if (t≥0) {
12)                 x=x+d;
13)                 c=2*(c-(2*x+d));
14)             }
15)             else {
16)                 c=2*c;
17)             }
18)         }
19)         output(“SQRT of p=“, x);
20)     }
21) end
```

Decisions at lines 5, 9, 11

All three decisions are covered.

All three decisions are simple predicates
➔ all three conditions are covered

Short Circuits

```
1    if (x < 0 and y < 0)
2        z = foo(x, y);
```

The condition at line 1 evaluates to false when x ≥ 0 regardless of the value of y.

Another condition, such as x < 0 OR y < 0, evaluates to true regardless of the value of y, when x < 0.

With this in view, compilers often generate code that uses short circuit evaluation of compound conditions.

Here is a possible translation:

```
if (x < 0 and y < 0)
    z = foo(x, y);
```

```
if (x < 0)
    if (y < 0)
        z = foo (x, y)
```

We now see two decisions, one corresponding to each simple condition in the if statement.

Condition and Decision Coverage

When a decision is composed of a compound condition, decision coverage does not imply that each simple condition within a compound condition has taken both values true and false.

Condition coverage ensures that each component simple condition within a condition has taken both values true and false.

Condition coverage does not require each decision to have taken both outcomes.

- **Condition/decision coverage is also known as branch condition coverage.**

Mathur, Example 7.15

Program P7.9

```
1    begin
2        int x, y, z;
3        input (x, y);
4        if (x < 0 or y < 0)
5            z = foo1 (x, y);
6        else
7            z = foo2 (x, y);
8        output (z);
9        end
```

Note that the difference between P7.8 and P7.9 is the OR on line 4.

Consider program P7.9 and two test sets.

T_1 = {t_1: <x=-3,y=2>,
t_2: <x=4,y=2>}

T_2 = {t_1:<x=-3,y=2>,
t_2: <x=4,y=-2>}

Confirm that T_1 is adequate with respect to to decision coverage but not condition coverage.

Confirm that T_2 is adequate with respect to condition coverage but not decision coverage.

Condition / Decision Coverage

The <u>condition/decision</u> coverage (aka <u>branch condition</u> coverage in IEEE 29119:4) of T with respect to (P,R) is computed as

$(|C_c| + |D_c|) / [(|C_e| - |C_i|) + (|D_e| - |D_i|)]$

- **$|C_c|$ is the number of simple conditions covered**
- **$|D_c|$ is the number of decisions covered**
- **$|C_e|$ and $|D_e|$ are the number of simple conditions and decisions respectively**
- **$|C_i|$ and $|D_i|$ are the number of infeasible simple conditions and decisions, respectively**

T is considered adequate with respect to the condition/decision coverage criterion if the condition/decision coverage of T with respect to (P,R) is 1.

Mathur, Example 7.16

Program P7.8

```
begin
    int x, y, z;
    input (x, y);
    if (x < 0 and y < 0)
        z = foo1 (x, y);
    else
        z = foo2 (x, y);
    output (z);
    end
```

Check that the following test set is adequate with respect to the condition/decision coverage criterion.

$$\mathbf{T = \{t_1: <x=-3,y=-2>,}$$
$$\mathbf{t_2: <x=4,y=2>\}}$$

Multiple Conditions

Consider a compound condition with two or more simple conditions.

Using condition coverage on some compound condition C implies that each simple condition within C has been evaluated to true and false.

Does it imply that all combinations of the values of the individual simple conditions in C have been exercised?

- ***No***

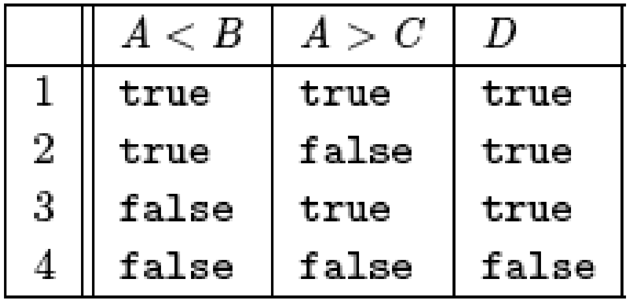
Mathur, Example 7.17

Consider D = (A < B) OR (A > C)

- **composed of two simple conditions A < B and A > C**

The four possible combinations of the outcomes of these two simple conditions are enumerated in the table.

Consider T = {t_1: <A=2,B=3,C=1>,
t_2: <A=2,B=1,C=3>}



	$A < B$	$A > C$	D
1	`true`	`true`	`true`
2	`true`	`false`	`true`
3	`false`	`true`	`true`
4	`false`	`false`	`false`

Does T cover all four combinations?

Consider T' = {t_1: <A=2,B=3,C=1>, t_2: <A=2,B=3,C=5>,
t_3: <A=2,B=1,C=1>, t_4: <A=2,B=1,C=3>}

Does T' cover all four combinations?

Number of Combinations for Multiple Conditions

Suppose that the program under test contains a total of n decisions.

- **Assume that each decision contains k_1, k_2, …, k_n simple conditions.**
- **Each decision has several combinations of values of its constituent simple conditions.**

For example, decision i will have 2^{ki} combinations.

The total number of combinations to be covered is

$$\sum_{i=1}^{n} 2^{ki}$$

Multiple Condition Coverage

The multiple condition coverage (aka branch condition combination coverage in IEEE 29119:4) of T with respect to (P,R) is computed as $|C_c| / (|C_e| - |C_i|)$

- $|C_c|$ is the number of combinations covered
- $|C_e|$ is the total number of combinations in the program
- $|C_i|$ is the number of infeasible simple combinations

T is considered adequate with respect to the multiple condition coverage criterion if the condition coverage of T with respect to (P,R) is 1.

Mathur, Example 7.18

Program P7.10

```
begin
    int A,B,C,S=0;
    input (A,B,C);
    if (A < B and A > C)
        S = f1(A,B,C);
    if (A < B and A ≤ C)
        S = f2(A,B,C);
    if (A ≥ B and A ≤ C)
        S = f4(A,B,C);
    output (S);
end
```

Consider program P7.10 with specifications in the table.

	A < B	A > C	S
1	**true**	**true**	**f1 (A,B,C)**
2	**true**	**false**	**f2 (A,B,C)**
3	**false**	**true**	**f3 (A,B,C)**
4	**false**	**false**	**f4 (A,B,C)**

There is an obvious error in the program.

- **computation of S for one of the four combinations, line 3 in the table, has been left out**

Program P7.10

```
1    begin
2        int A,B,C,S=0;
3        input (A,B,C);
4        if (A < B and A > C)
             S = f1(A,B,C);
5        if (A < B and A ≤ C)
             S = f2(A,B,C);
6        if (A ≥ B and A ≤ C)
             S = f4(A,B,C);
7        output (S);
8    end
```

3 decisions
6 conditions
12 combinations

Consider
T = {t_1: <A=2,B=3,C=1>,
t_2: <A=2,B=1,C=3>}

Is T adequate with respect to condition coverage?

- **Yes**

Decision coverage?

- **Not at line 5**

Does it reveal the error?

- **No**

Program P7.10

```
1    begin
2        int A,B,C,S=0;
3        input (A,B,C);
4        if (A < B and A > C)
             S = f1(A,B,C);
5        if (A < B and A ≤ C)
             S = f2(A,B,C);
6        if (A ≥ B and A ≤ C)
             S = f4(A,B,C);
7        output (S);
8    end
```

Consider

T' = {t_1: <A=2,B=3,C=1>,
t_2: <A=2,B=1,C=3>,
t_3: <A=2,B=3,C=5>}

Is T' adequate with respect to decision coverage?

- **Yes**

Does it reveal the error?

- **No**

Is T' adequate with respect to multiple condition coverage?

- **…No…**

A<B	A>C	T	A<B	A≤C	T	A≥B	A≤C	T
true	true	t_1	true	true	t_3	true	true	t_2
true	false	t_3	true	false	t_1	true	false	--
false	true	--	false	true	t_2	false	true	t_3
false	false	t_2	false	false	--	false	false	t_1

Program P7.10

```
1   begin
2       int A,B,C,S=0;
3       input (A,B,C);
4       if (A < B and A > C)
           S = f1(A,B,C);
5       if (A < B and A ≤ C)
           S = f2(A,B,C);
6       if (A ≥ B and A ≤ C)
           S = f4(A,B,C);
7       output (S);
8   end
```

Consider

T’’ = {t_1: <A=2,B=3,C=1>,
t_2: <A=2,B=1,C=3>,
t_3: <A=2,B=3,C=5>,
t_4: <A=2,B=1,C=1>}

Is T’’ adequate with respect to multiple condition coverage?

- **Yes**

Does it reveal the error?

- **Yes**

Program P7.10

```
1   begin
2       int A,B,C,S=0;
3       input (A,B,C);
4       if (A < B and A > C)
            S = f1(A,B,C);
5       if (A < B and A ≤ C)
            S = f2(A,B,C);
6       if (A ≥ B and A ≤ C)
            S = f4(A,B,C);
7       output (S);
8   end
```

Note that since all three decisions in P7.10 use the same set of variables, we only need to analyze one decision to obtain a test set adequate with respect to multiple condition coverage.

Linear Code Sequence and Jump (LCSAJ)

Execution of sequential programs that contain at least one condition proceeds in pairs where

- **the first element of the pair is a sequence of statements (a block) executed one after the other**
- **terminated by a jump to the next such pair (another block)**

A Linear Code Sequence and Jump is a program unit comprised of a textual code sequence that terminates in a jump to the beginning of another code sequence and jump.

LCSAJ Triples

LCSAJ is represented as a triple (X, Y, Z)

X and Y are, respectively, locations of the first and the last statements and Z is the location to which the statement at Y jumps.

The last statement in an LCSAJ (X, Y, Z) is a jump and Z may be program exit.

- **When control arrives at statement X, follows through to statement Y, and then jumps to statement Z, we say that the LCSAJ (X, Y, Z) is <u>traversed</u> (covered, exercised).**

Mathur, Example 7.19

Program P7.11

```
1   begin
2         int x, y, p;
3         input (x, y);
4         if (x < 0)
5             p = g (y);
6         else
7             p = g (y*y);
8    end
```

Consider Program P7.11.

LCSAJ	**Start**	**End**	**Jump**
1	**1**	**6**	**exit**
2	**1**	**4**	**7**
3	**7**	**8**	**exit**

T = {t_1: <x=-5,y=2>,
t_2: <x=9,y=2>}

t_1 – LCSAJ (1, 6, exit)
t_2 – LCSAJ (1, 4, 7) → (7, 8, exit)

T covers all three LCSAJs

Mathur, Example 7.20

Program P7.12

```
1    begin
2    // Compute x**y given
        non-negative integers
        x and y
3        int x, y, p;
4        input (x, y);
5        p = 1;
6        count = y;
7        while (count > 0) {
8            p = p * x;
9            count = count – 1;
10       }
11       output (p);
12   end
```

In class exercise

- **find all LCSAJs for P7.12**

LCSAJ	**Start**	**End**	**Jump**
1	**1**	**10**	**7**
2	**7**	**10**	**7**
3	**7**	**7**	**11**
4	**1**	**7**	**11**
5	**11**	**12**	**exit**

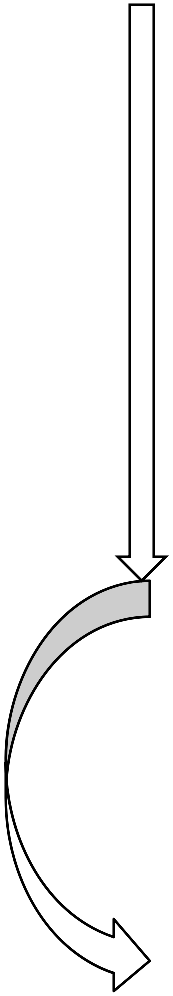
T = {t_1: <x=5,y=0>,
t_2: <x=5,y=2>}

t_1 – LCSAJ (1, 7, 11) → (11, 12, exit)
t_2 – LCSAJ (1, 10, 7) → (7, 10, 7) → (7, 7, 11) → (11, 12, exit)

LCSAJs for an IF

A succeeding predicate gives

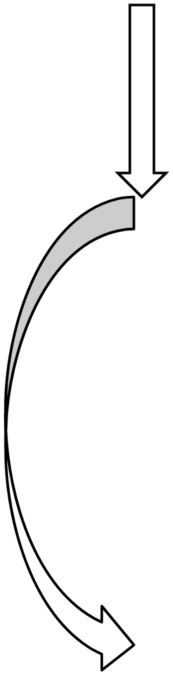
(1, 8, 13)



```
1)     ~~~~~~~~
2)     ~~~~~~~~
3)     if (predicate)
4)     {
5)        ~~~~~~~~
6)        ~~~~~~~~
7)     }
8)     else
9)     {
10)       ~~~~~~~~
11)       ~~~~~~~~
12)    }
13)    ~~~~~~~~
14)    ~~~~~~~~
```

A failing predicate gives

(1, 3, 9)

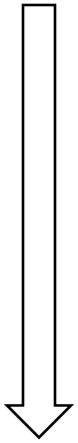


```
1)      ~~~~~~~~~
2)      ~~~~~~~~~
3)      if (predicate)
4)      {
5)          ~~~~~~~~~
6)          ~~~~~~~~~
7)      }
8)      else
9)      {
10)         ~~~~~~~~~
11)         ~~~~~~~~~
12)     }
13)     ~~~~~~~~~
14)     ~~~~~~~~~
```

The else branch of a failing predicate implies

(9, 14, exit)

assuming that 14 is an end… it could also be a jump to somewhere else…



```
1)      ~~~~~~~~
2)      ~~~~~~~~
3)      if (predicate)
4)      {
5)         ~~~~~~~~
6)         ~~~~~~~~
7)      }
8)      else
9)      {
10)        ~~~~~~~~
11)        ~~~~~~~~
12)     }
13)     ~~~~~~~~
14)     ~~~~~~~~
```

IFs and LCSAJs

If an IF is true, then the flow into the THEN part of the loop is <u>part of</u> the incoming linear sequence.

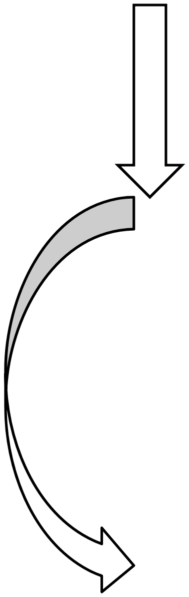
- **ELSE is treated as a GOTO at end of the THEN**

The ELSE part of an IF begins a new linear sequence.

LCSAJs for a WHILE

A loop that executes 0 times gives

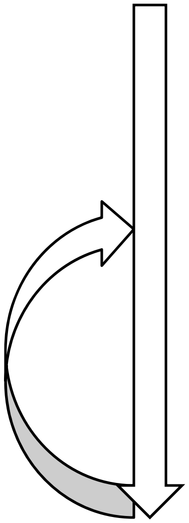
(1, 3, 8)



```
1)      ~~~~~~~~~
2)      ~~~~~~~~~
3)      while (predicate)
4)      {
5)          ~~~~~~~~~
6)          ~~~~~~~~~
7)      }
8)      ~~~~~~~~~
9)      ~~~~~~~~~
```

The first loop execution gives

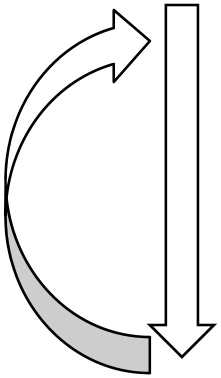
(1, 7, 3)



```
1)      ~~~~~~~~~
2)      ~~~~~~~~~
3)      while (predicate)
4)      {
5)          ~~~~~~~~~
6)          ~~~~~~~~~
7)      }
8)      ~~~~~~~~~
9)      ~~~~~~~~~
```

The loop body that executes n times gives

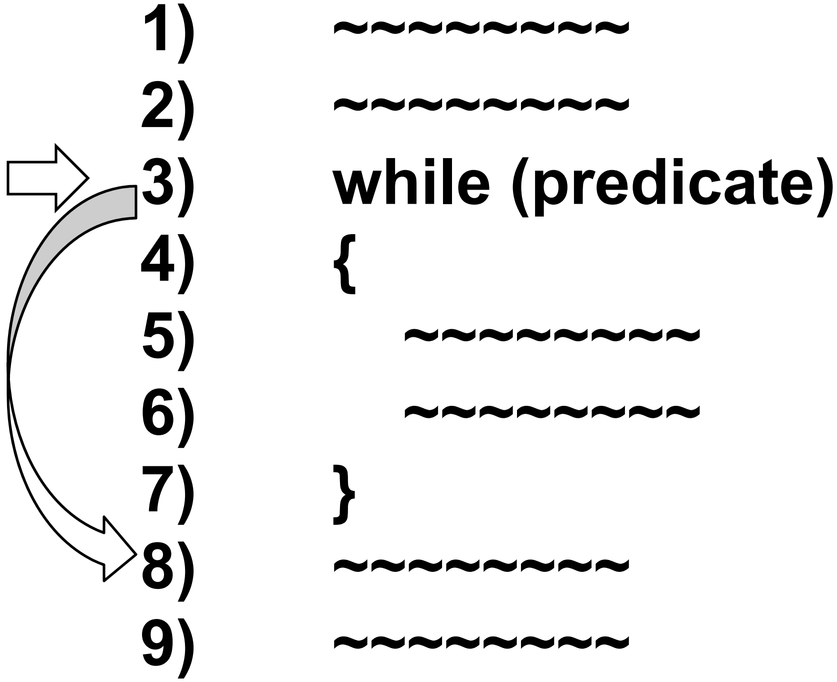
(3, 7, 3)



```
1)      ~~~~~~~~~
2)      ~~~~~~~~~
3)      while (predicate)
4)      {
5)          ~~~~~~~~~
6)          ~~~~~~~~~
7)      }
8)      ~~~~~~~~~
9)      ~~~~~~~~~
```

The last loop execution (of n) gives

(3, 3, 8)



```
1)      ~~~~~~~~~
2)      ~~~~~~~~~
3)      while (predicate)
4)      {
5)          ~~~~~~~~~
6)          ~~~~~~~~~
7)      }
8)      ~~~~~~~~~
9)      ~~~~~~~~~
```

Loops and LCSAJs

The loop body will be a linear sequence.

The loop body is part of the incoming linear sequence during the first execution of the loop.

For a pre-test loop, the loop header is a single-line linear sequence.

Labeling Conventions

Assuming the use of {} to delimit blocks.

- **start blocks at the “{“ ...**
- **jump past the “}” to the next line…**
- **note that you may be jumping past one } to an enclosing } … }}**

Make sure that the line you’re jumping to is the start of a linear sequence

- **may find missing sequences this way**

LCSAJ Coverage

The LCSAJ coverage of a test set T with respect to (P,R) is computed as

$$\frac{\textbf{Number of LCSAJs traversed}}{\textbf{Total number of feasible LCSAJs}}$$

T is considered adequate with respect to the LCSAJ coverage criterion if the LCSAJ coverage of T with respect to (P,R) is 1.

LCSAJ Example

1 (1, 7, exit)

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output ("Boundary condition failure
                 on inputs.");
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)

```
 Program P
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output (“Boundary condition failure
                 on inputs.”);
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output (“Boundary condition failure
                 on inputs.”);
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output ("Boundary condition failure
                 on inputs.");
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output ("Boundary condition failure
                 on inputs.");
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)
6	(19, 21, 26)

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output ("Boundary condition failure
                 on inputs.");
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)
6	(19, 21, 26)
7	(19, 25, 29)

Note that this is an infeasible LCSAJ since Y==0 in this else clause (legal inputs) is impossible.

```
 Program P
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output ("Boundary condition failure
                 on inputs.");
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)
6	(19, 21, 26)
7	(19, 25, 29)
8	(26, 32, exit)

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output ("Boundary condition failure
                 on inputs.");
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)
6	(19, 21, 26)
7	(19, 25, 29)
8	(26, 32, exit)
9	(29, 32, exit)

Note that the only way to get to this LCSAJ is from the infeasible LCSAJ 7, so it is infeasible also.

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output (“Boundary condition failure
                 on inputs.”);
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)
6	(19, 21, 26)
7	(19, 25, 29)
8	(26, 32, exit)
9	(29, 32, exit)
10	(30, 32, exit)

```
 Program P
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output (“Boundary condition failure
                 on inputs.”);
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)               Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)              Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)               Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

LCSAJs

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)
6	(19, 21, 26)
7	(19, 25, 29)
8	(26, 32, exit)
9	(29, 32, exit)
10	(30, 32, exit)
11	(17, 18, 30)

Done.

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output (“Boundary condition failure
                 on inputs.”);
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)              Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)               Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

Are All LCSAJs Listed?

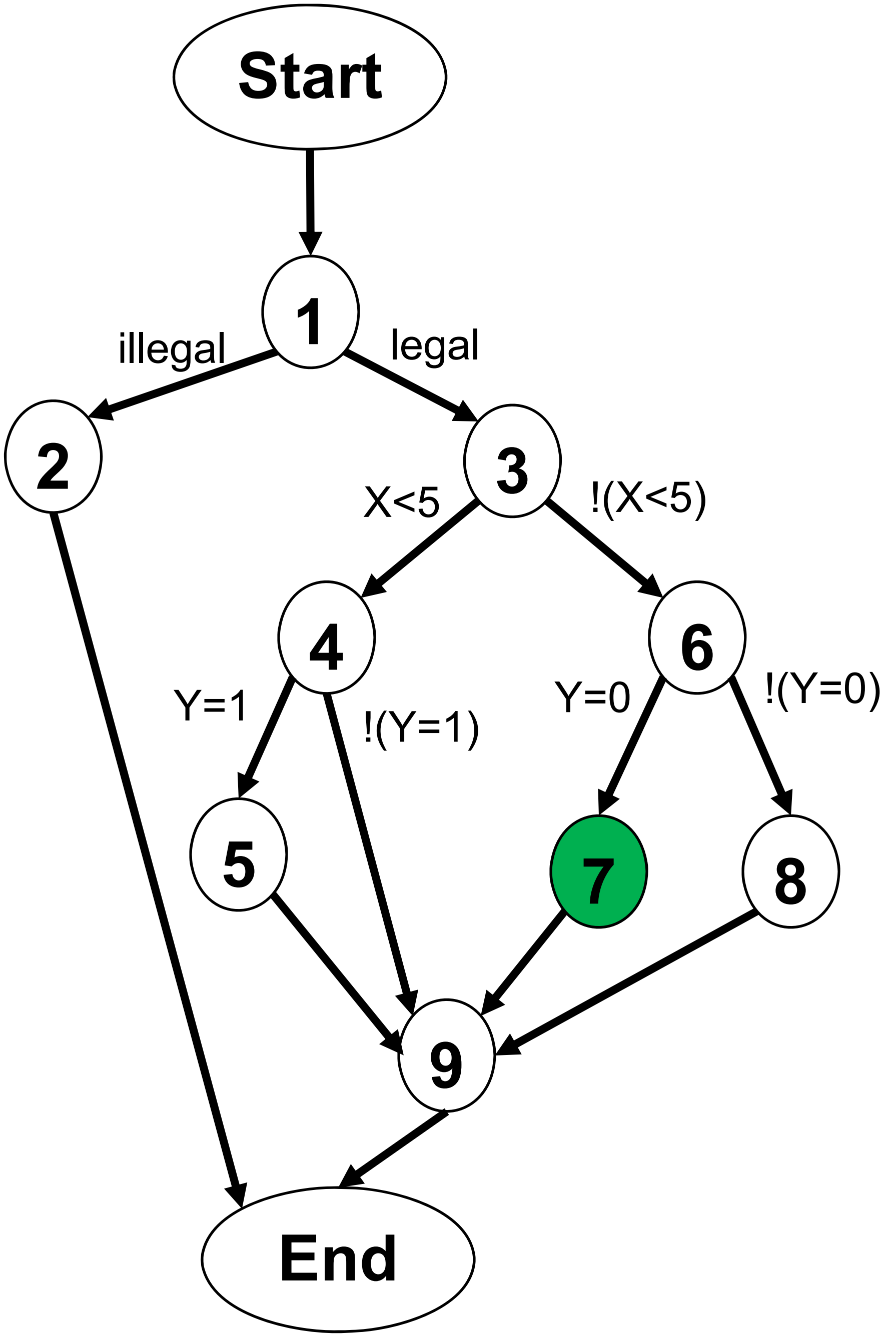
Note that LCSAJ #11 was added in when I started checking the sequences.

A (sometimes feasible) sanity check…
What are all the paths through the code?

- **(1, 7, exit)**
- **(1, 3, 8) → (8, 10, 19) → (19, 21, 26) → (26, 32, exit)**
- **(1, 3, 8) → (8, 10, 19) → (19, 25, 29) → (29, 32, exit)**
- **(1, 3, 8) → (8, 13, 17) → (17, 18, 30) → (30, 32, exit)**
- **(1, 3, 8) → (8, 18, 30) → (30, 32, exit)**

Also note the CFG on the next slide.

Block	LOC
1	1,2,3
2	5
3	9,10
4	12,13
5	15
6	20,21
7	23 (infeasible)
8	27
9	30



Block-based paths through the CFG

- S-1-2-E
- S-1-3-4-5-9-E
- S-1-3-4-9-E
- S-1-3-6-7-9-E
- S-1-3-6-8-9-E

LCSAJ coverage of T = {t_1 = <4, 2>, t_2 = <3, 1>}

t_1 : (1, 3, 8) →
(8, 13, 17) →
(17, 18, 30) →
(30, 32, exit)

t_2 : (1, 3, 8) →
(8, 18, 30) →
(30, 32, exit)

1	(1, 7, exit)
2	(1, 3, 8)
3	(8, 10, 19)
4	(8, 13, 17)
5	(8, 18, 30)
6	(19, 21, 26)
7	(19, 25, 29)
8	(26, 32, exit)
9	(29, 32, exit)
10	(30, 32, exit)
11	(17, 18, 30)

Program P

```
1)  integer X, Y, Z;
2)  input (X, Y);
3)  if (X<0 or X>8 or Y<1 or Y>3)
4)  {
5)       output ("Boundary condition failure
                 on inputs.");
6)  } // end input check
7)  else
8)  {
9)       Z = 0;
10)      if (X < 5)
11)      {
12)           Z = X + Y;
13)           if (Y == 1)
14)          {
15)                Z = X ^ 2;
16)          } // end if (Y==1)
17)      } // end if (X<5)
18)     else
19)     {
20)          Z = Z - X;
21)          if (Y == 0)
22)          {
23)               Z = Z * Z;
24)           } // end if (Y==2)
25)          else
26)          {
27)                Z = Z + X;
28)          } // end else !(Y==2)
29)     } // end else !(X<5)
30)     output (X,Y,Z);
31) } // end else legal inputs
32) end;
```

$\| D_{LCSAJ} \| = 11$	total LCSAJs
$\| D_{I} \| = 2$	infeasible LCSAJs
$\| D_{C} \| = 5$	traversed LCSAJs

LCSAJ Coverage = 5 / (11-2) = 5 / 9 = 56%

Modified Condition/Decision (MC/DC) Coverage

Obtaining multiple condition coverage might become expensive when there are many embedded simple conditions.

- **when a compound condition C contains n simple conditions, the maximum number of tests required to cover C is 2^n**

n	Min tests	Time to execute
1	2	2 ms
4	16	16 ms
8	256	256 ms
16	65,536	65.5 sec
32	4,294,967,296	49.5 days

Compound Conditions and MC/DC

MC/DC coverage requires that every compound condition in a program must be tested by demonstrating that each simple condition within the compound condition has an independent effect on its outcome.

MC/DC coverage is a weaker criterion than the multiple condition coverage criterion.

- **for a compound condition with two simple conditions, multiple condition coverage would require four tests**
- **MC/DC requires three tests**
- **size of an MC/DC-adequate test set grows linearly in the number of simple conditions**

A Simple MC/DC Procedure

Hold all the simple conditions but C_x in C constant and vary C_x

Do so for every combination of the simple conditions in the truth table excluding C_x

Select two tests from the truth table that demonstrate the independent effect of C_x on C

MC/DC Coverage for Compound Conditions with Two Simple Conditions

Mathur, Table 7.9

Condition: C_a = C_1 and C_2

Test	C_1	C_2	C	Comments
t_1	true	true	true	Tests t_1 and t_2 cover C_2
t_2	true	false	false	
t_3	false	true	false	Tests t_1 and t_3 cover C_1

Each simple condition affects the outcome of C independently.

- **holding C_1 true in t_1 and t_2 while varying C_2 affects the outcome C**
- **holding C_2 true in t_1 and t_3 while varying C_1 affects the outcome C**

Condition: $C_a = C_1$ or C_2

Test	C_1	C_2	C	Comments
t_4	false	true	true	Tests t_4 and t_5 cover C_2
t_5	false	false	false	
t_6	true	false	true	Tests t_5 and t_6 cover C_1

Each simple condition affects the outcome of C independently.

- holding C_1 false in t_4 and t_5 while varying C_2 affects the outcome C
- holding C_2 false in t_5 and t_6 while varying C_1 affects the outcome C

Condition: $C_a = C_1$ xor C_2

Test	C_1	C_2	C	Comments
t_7	true	true	false	Tests t_7 and t_8 cover C_2
t_8	true	false	true	
t_9	false	false	false	Tests t_8 and t_9 cover C_1

Each simple condition affects the outcome of C independently.

- **holding C_1 true in t_7 and t_8 while varying C_2 affects the outcome C**
- **holding C_2 false in t_8 and t_9 while varying C_1 affects the outcome C**

MC/DC Coverage for (C_1 and C_2 and C_3)

Mathur, Table 7.10

Condition: $C_a = C_1$ and C_2 and C_3

Test	C_1	C_2	C_3	C_a	Comments
t_1	true	true	true	true	Tests t_1 and t_2 cover C_3
t_2	true	true	false	false	
t_3	true	false	true	false	Tests t_1 and t_3 cover C_2
t_4	false	true	true	false	Tests t_1 and t_4 cover C_1

Each simple condition affects the outcome of C_a independently.

- **holding C_1 and C_2 true in t_1 and t_2 while varying C_3**
- **holding C_1 and C_3 true in t_1 and t_3 while varying C_2**
- **holding C_2 and C_3 true in t_1 and t_4 while varying C_1**

MC/DC Coverage for (C_1 or C_2 or C_3)

Mathur, Table 7.10

Condition: C_b = C_1 or C_2 or C_3

Test	C_1	C_2	C_3	C_b	Comments
t_5	false	false	false	false	Tests t_5 and t_6 cover C_3
t_6	false	false	true	true	
t_7	false	true	false	true	Tests t_5 and t_7 cover C_2
t_8	true	false	false	true	Tests t_5 and t_8 cover C_1

Each simple condition affects the outcome of C_b independently.

- **holding C_1 and C_2 false in t_5 and t_6 while varying C_3**
- **holding C_1 and C_3 false in t_5 and t_7 while varying C_2**
- **holding C_2 and C_3 false in t_5 and t_8 while varying C_1**

MC/DC or MCDC (DO-178B)

Modified Condition/Decision Coverage

- **each decision tries every possible outcome**
- **each condition in a decision takes on every possible outcome**
- **each entry and exit point is invoked**
- **each condition in a decision is shown to independently affect the outcome of the decision**

A condition is shown to independently affect a decisions outcome by

- **varying just that condition**
- **while holding fixed all other possible conditions**

MC/DC Coverage
Generating Tests for Compound Conditions

Given C = C_1 and C_2 and C_3.

Create a table with five columns and four rows.

Label the columns as Test, C_1, C_2, C_3, and C, from left to right.

An optional column labeled "Comments" may be added.

The column labeled Test contains rows labeled by test case numbers t_1 through t_4.

The remaining entries are empty.

Test	C_1	C_2	C_3	C	Comments
t_1					
t_2					
t_3					
t_4					

Copy all entries in columns C_1, C_2, and C from the table for simple conditions into columns C_2, C_3, and C of the empty table.

Test	C_1	C_2	C_3	C	Comments
t_1		true	true	true	
t_2		true	false	false	
t_3		false	true	false	
t_4					

Fill the first three rows in the column marked C_1 with true and the last row with false.

Test	C_1	C_2	C_3	C	Comments
t_1	true	true	true	true	
t_2	true	true	false	false	
t_3	true	false	true	false	
t_4	false				

Fill the last row under columns labeled C_2, C_3, and C with true, true, and false, respectively.

Test	C_1	C_2	C_3	C	Comments
t_1	true	true	true	true	t_1 and t_2 cover C_3
t_2	true	true	false	false	
t_3	true	false	true	false	t_1 and t_3 cover C_2
t_4	false	true	true	false	t_1 and t_4 cover C_1

We now have a table containing MC/DC adequate tests for $C = (C_1 \text{ AND } C_2 \text{ AND } C_3)$ derived from tests for $C = (C_1 \text{ AND } C_2)$

This procedure can be extended to derive tests for any compound condition using tests for a simpler compound condition.

MC/DC Coverage

A test set T for program P written to meet requirements R, is considered adequate with respect to the MC/DC coverage criterion if, upon the execution of P on each test in T, the following requirements are met.

- **Each block in P has been covered.**
- **Each simple condition in P has taken both true and false values.**
- **Each decision in P has taken all possible outcomes.**
- **Each simple condition within a compound condition C in P has been shown to independently affect the outcome of C.**
  - ***This is the MC part of the coverage.***

Analysis of MC/DC Coverage

The first three requirements above correspond to block, condition, and decision coverage, respectively.

The fourth requirement corresponds to modified condition (MC) coverage.

The MC/DC coverage criterion is a mix of four coverage criteria based on the flow of control.

MC/DC needs only *n+1* test cases to test a decision that contains *n* conditions (Kandl 2010).

With regard to the second requirement, conditions that are not part of a decision, such as

$$A = (p < q) \text{ OR } (x > y)$$

are also included in the set of conditions to be covered.

With regard to the fourth requirement, a condition such as

(A AND B) OR (C AND A)

poses a problem.

It is not possible to keep the first occurrence of A fixed while varying the value of its second occurrence.

The first occurrence of A is said to be coupled to its second occurrence.

In such cases an adequate test set need only demonstrate the independent effect of any one occurrence of the coupled condition

Problems with MC/DC

Two difficult issues with MC/DC

- **short circuit operators**
  - **relax the requirement that conditions be held constant if conditions are not evaluated due to a short-circuit operator (Chilenski 1994)**
  - **consider the condition operands of short-circuit operators as separate decisions (DO-248B)**
- **multiple occurrences of a condition**
  - **Unique Cause MCDC: interpret the term "condition" to mean "uncoupled condition"**
  - **Masking MCDC: permit more than one condition to vary at once, using an analysis of the logic of the decision to ensure that only the condition of interest influences the outcome**

MC/DC Coverage Adequacy

Let C_1, C_2, .., C_N be the conditions in P.

- **n_i denote the number of simple conditions in C_i**
- **e_i the number of simple conditions shown to have independent affect on the outcome of C_i**
- **f_i the number of infeasible simple conditions in C_i**

The MC coverage of T for program P subject to requirements R, denoted by MC_c, is computed as follows.

$$\mathrm{MC}_c = \frac{\sum_{i=1}^{i=N} e_i}{\sum_{i=1}^{i=N} (e_i - f_i)}$$

Test set T is considered adequate with respect to the MC coverage criterion if MC_c= 1.

Error Types in a Compound Condition

Missing condition

- **One or more simple conditions is missing from a compound condition.**
  - **For example, the correct condition should be (x<y AND done) but the condition coded is (done).**

Incorrect Boolean operator

- **One or more Boolean operators is incorrect.**
  - **For example, the correct condition is (x<y AND done) which has been coded as (x<y OR done).**

Mixed

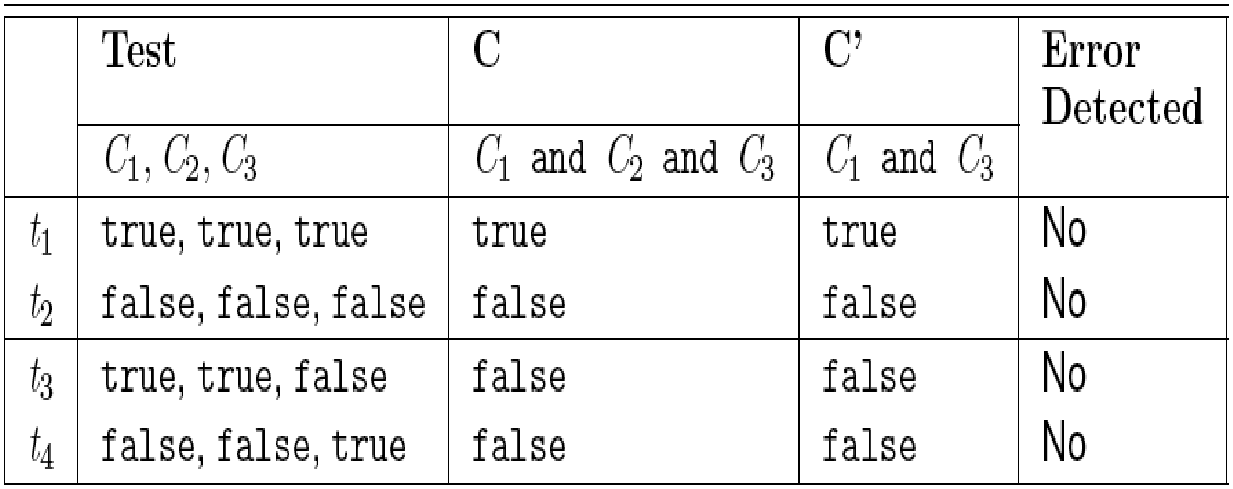
- **One or more simple conditions is missing and one or more Boolean operators is incorrect.**
  - **For example, the correct condition should be (x<y AND z*x ≥ y AND d="South") has been coded as (x<y OR z*x ≥ y).**

Mathur, Example 7.24

Suppose that condition $C = C_1$ AND C_2 AND C_3 has been coded as $C' = C_1$ AND C_3.

Four tests that form an MC/DC adequate set are in the following table.

The test set does not reveal the coding error.



	Test	C	C'	Error Detected
	C_1, C_2, C_3	C_1 and C_2 and C_3	C_1 and C_3	
t_1	true, true, true	true	true	No
t_2	false, false, false	false	false	No
t_3	true, true, false	false	false	No
t_4	false, false, true	false	false	No

MC/DC and Condition Coverage

Satisfying the MC/DC adequacy criteria does not necessarily imply that errors made while coding conditions will be revealed.

An MC/DC-adequate test is likely to reveal more errors than a decision or condition coverage adequate test.

- **with the emphasis on “likely”**

MC/DC and Short Circuit Evaluation

Consider C = C_1 AND C_2

The outcome of the above condition does not depend on C_2 when C_1 is false.

When using short-circuit evaluation, condition C_2 is not evaluated if C_1 evaluates to false.

Thus the combination (C$_1$ = false and C$_2$ = true) or the combination (C$_1$ = false and C$_2$ = false) may be infeasible if the programming language allows, or requires as in C, short circuit evaluation.

Tracing Test Cases to Requirements

When enhancing a test set to satisfy a given coverage criterion, it is desirable to ask the following question:

- **What portions of the requirements are tested when the program under test is executed against the newly added test case?**

The task of relating the new test case to the requirements is known as test trace-back.

  - **requirements traceability matrix, bi-directional traceability**

- **Assists us in determining whether or not the new test case is redundant.**
- **It has the likelihood of revealing errors and ambiguities in the requirements.**
- **It assists with the process of documenting tests against requirements.**

Coverage Goal for Release
(Bullseye.com)

Using statement coverage, decision coverage, or condition/decision coverage you generally want to attain 80-90% coverage or more before releasing.

Avoid setting a goal lower than 80%.

Condition/decision coverage is the best general-purpose metric for C, C++, and Java.

Summary – Things to Remember

Statement, block, decision, condition coverage

Coupled conditions

Short circuit evaluation

Condition/decision (branch condition) coverage

Multiple condition coverage

MC/DC coverage

Questions and Answers