F. Tip and M. Weintraub

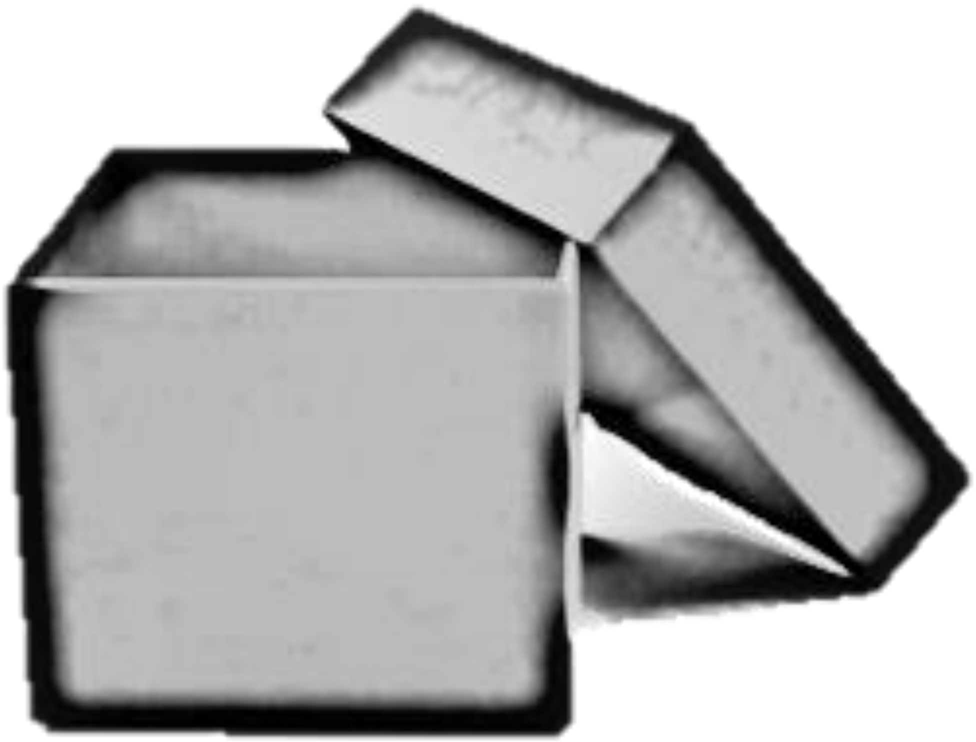
STRUCTURAL TESTING

AKA White Box Testing

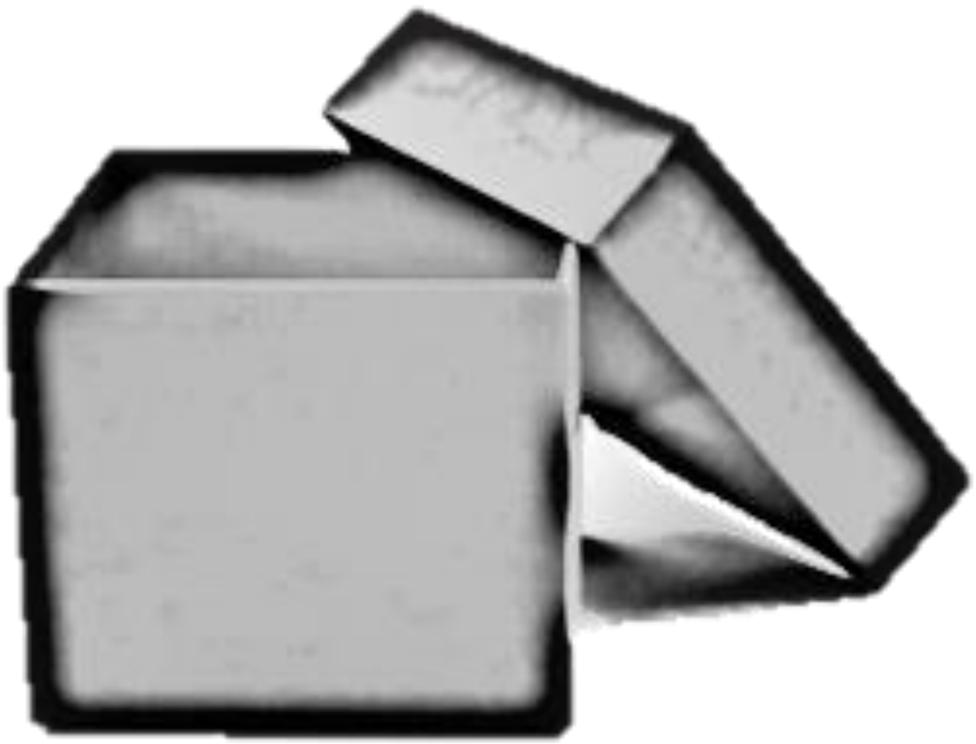
Thanks go to Andreas Zeller for allowing incorporation of his materials

STRUCTURAL TESTING

- Testing based on the structure of the code
- Test covers as much implemented behavior as possible



WHY DO STRUCTURAL TESTING?



- Defects may lurk in the darkness of code parts that are never executed.
- *Code parts* may be
  - a statement,
  - function,
  - transition,
  - condition…
- Attractive because it can be automated and it can be finer grained than functional testing

STRUCTURAL TESTING COMPLEMENTS FUNCTIONAL TESTING

Run functional tests first, then measure what is missing

Structural testing can cover low-level details missed in high-level specifications

BACK TO OUR ROOTS

```
class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a,
                 double b,
                 double c)
    { … }
    // Result: values for x
    double root_one, root_two;
}
```

- For which values for *a, b, c* should we test?
- If a, b, c, are 32-bit integers, there are $(2^{32})^3 \approx 10^{28}$ legal inputs
- *At 1,000,000,000,000 tests/s (10^{12} tests/s), you still need ~2.5 billion years to test everything*

THE CODE BEHIND THE INTERFACE

```
// Solve ax² + bx + c = 0

    public roots(double a, double b, double c)
    {
        double q = b * b - 4 * a * c;

        if (q > 0 && a ≠ 0) {
            // code for handling two roots
        }
        else if (q == 0) {
            // code for handling one root
        }
        else {
            // code for handling no roots
        }
    }
```

THE CODE BEHIND THE INTERFACE

The specification

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
// Solve ax² + bx + c = 0

    public roots(double a, double b, double c)
    {
        double q = b * b - 4 * a * c;

        if (q > 0 && a ≠ 0) {
            // code for handling two roots
        }
        else if (q == 0) {
            // code for handling one root
        }
        else {
            // code for handling no roots
        }
    }
```

Three cases to test

THE TEST CASES

The specification

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
// Solve ax² + bx + c = 0

    public roots(double a, double b, double c)
    {
        double q = b * b - 4 * a * c;

        if (q > 0 && a ≠ 0) {
            // code for handling two roots
        }
        else if (q == 0) {
            // code for handling one root
        }
        else {
            // code for handling no roots
        }
    }
```

$Case_0$: (a, b, c) = (3, 4, 1)

$Case_1$: (a, b, c) = (0, 0, 1)

$Case_2$:(a, b, c) = (3, 2, 1)

Finding appropriate input values may require significant domain skills.

FILLING IN THE CODE

```
// Solve ax² + bx + c = 0

    public roots(double a, double b, double c)
    {
        double q = b * b - 4 * a * c;

        if (q > 0 && a ≠ 0) {
            // code for handling two roots
        }
        else if (q == 0) {
            // code for handling one root

        }
        else {
            // code for handling no roots
        }
    }
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$Case_1$: (a, b, c) = (0, 0, 1)

FILLING IN THE CODE REVEALS A DEFECT

```
// Solve ax² + bx + c = 0

    public roots(double a, double b, double c)
    {
        double q = b * b - 4 * a * c;

        if (q > 0 && a ≠ 0) {
            // code for handling two roots
        }
        else if (q == 0) {
            // code for handling one root
                x = (-b) / (2 * a);
        }
        else {
            // code for handling no roots
        }
    }
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Case$_1$: *(a, b, c) = (0, 0, 1)*

code must handle a = 0

EXPRESSING STRUCTURE

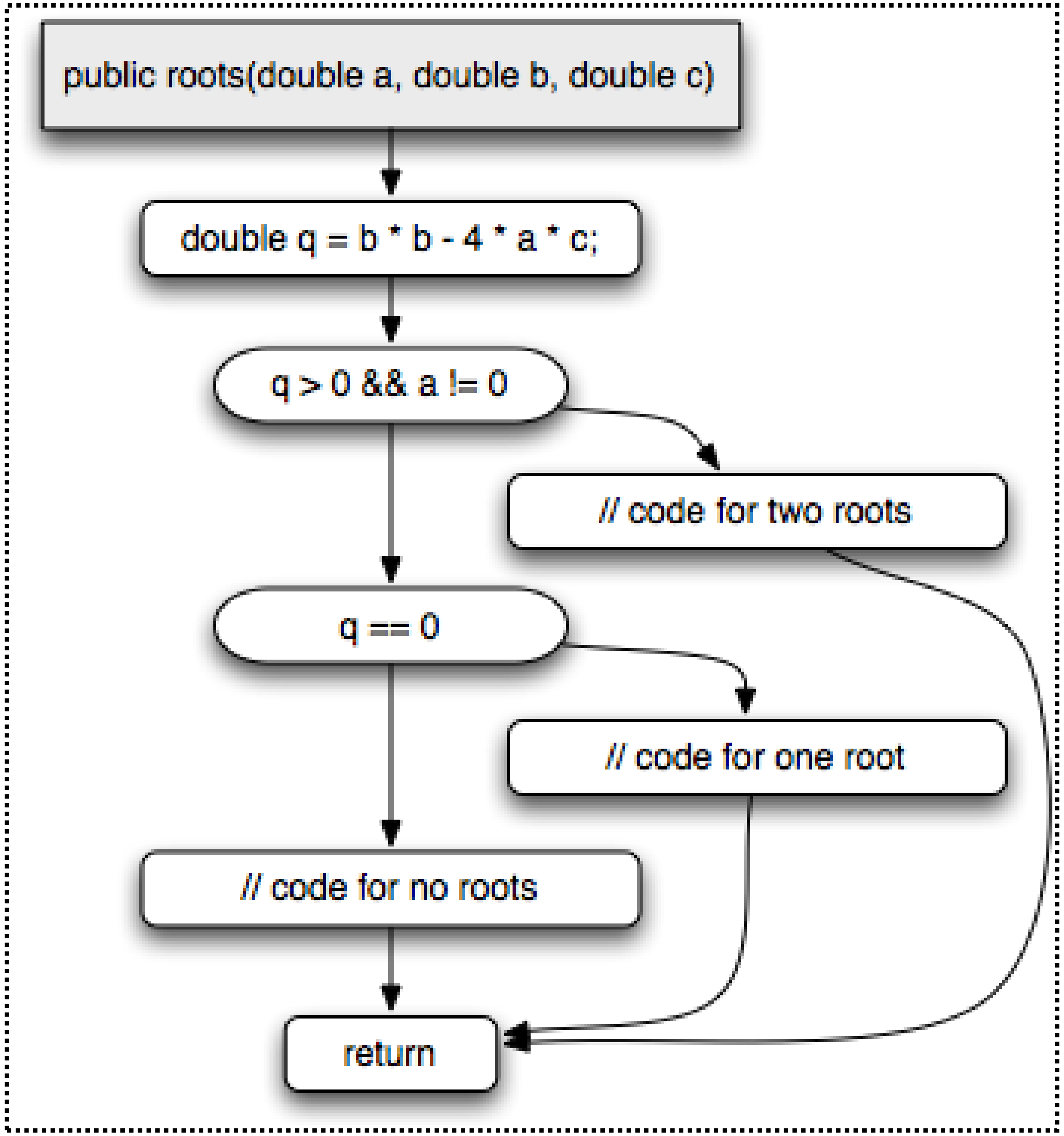
```
// Solve ax² + bx + c = 0

    public roots(double a, double b, double c)
    {
        double q = b * b - 4 * a * c;

        if (q > 0 && a ≠ 0) {
            // code for handling two r
        }
        else if (q == 0) {
            // code for handling one r
                x = (-b) / (2 * a);
        }
        else {
            // code for handling no ro
        }
    }
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What is important is the program structure.

The failure occurs only if
1. a specific condition is true AND
2. a specific branch is taken.

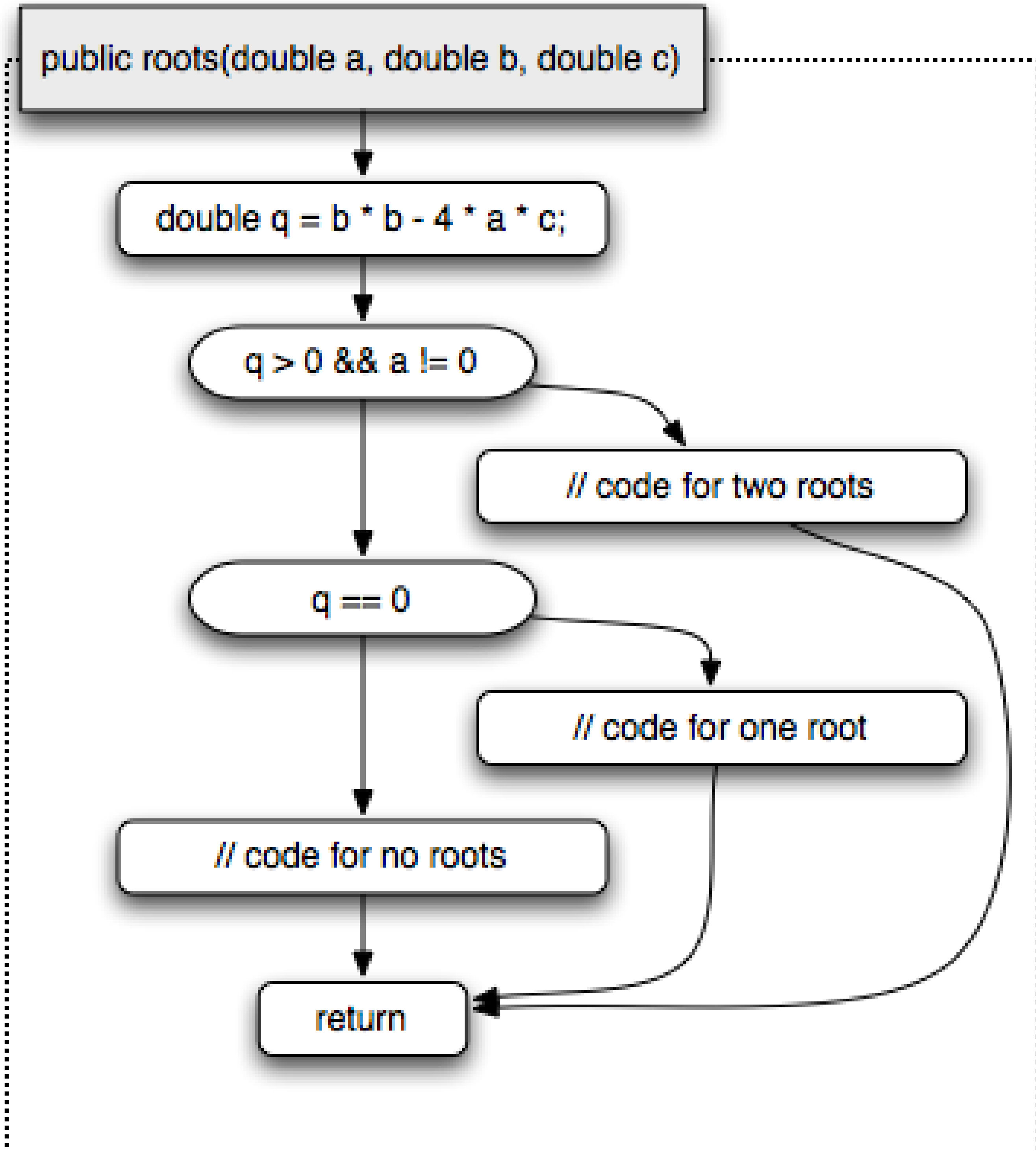
CONTROL FLOW GRAPH (CFG)



A control flow graph expresses *paths of program execution*

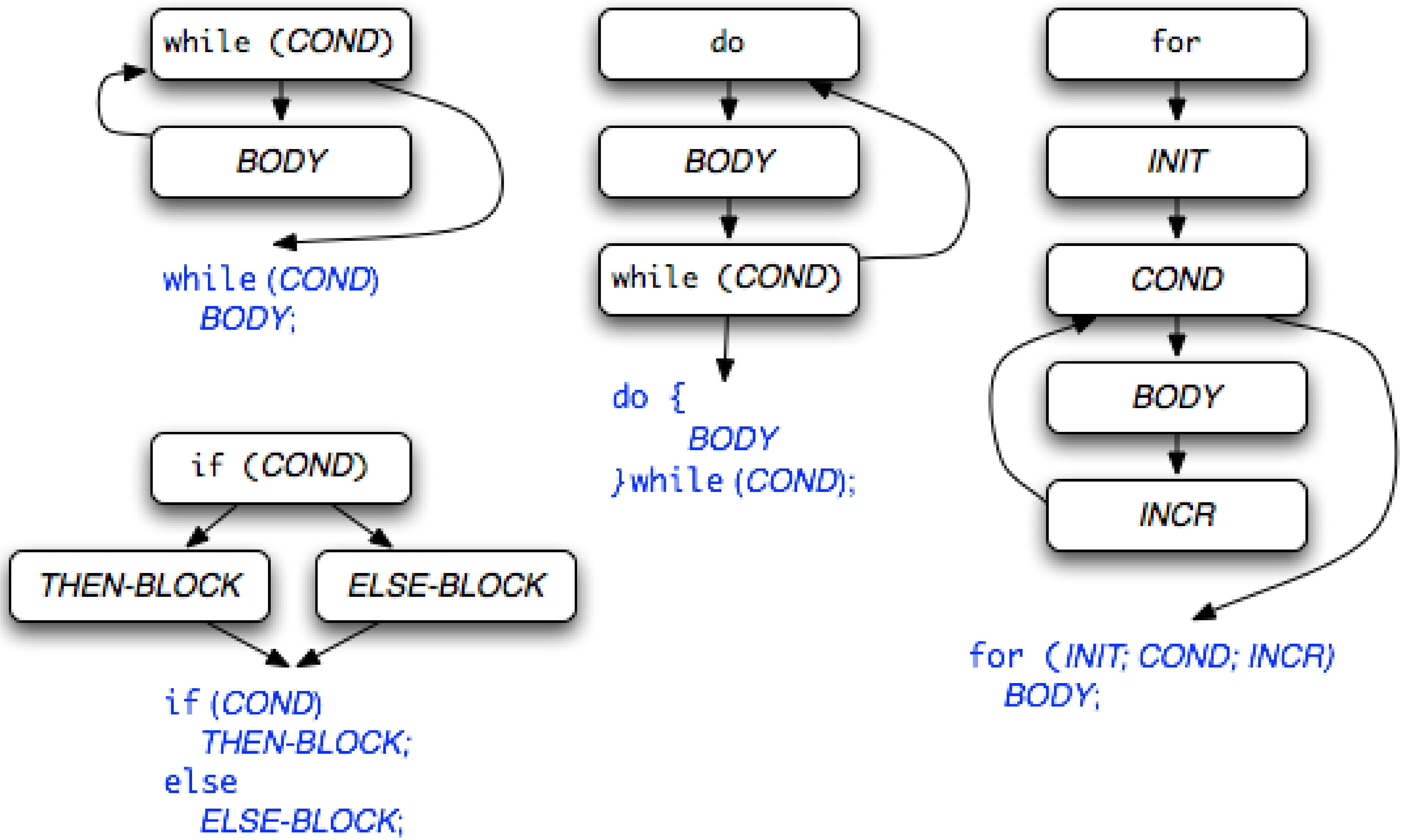
1. *Nodes are basic blocks – sequences of statements with one entry and one exit point*
2. *Edges represent control flow – the possibility that the program execution proceeds from the end of one basic block to the beginning of another*

TEST ADEQUACY CRITERIA



- The CFG can serve as an *adequacy criterion* for test cases
- The more parts that are covered (executed) by tests, the better the chance that a test uncovers a defect
- *Parts* can be: nodes, edges, paths, conditions, …

CONTROL FLOW PATTERNS



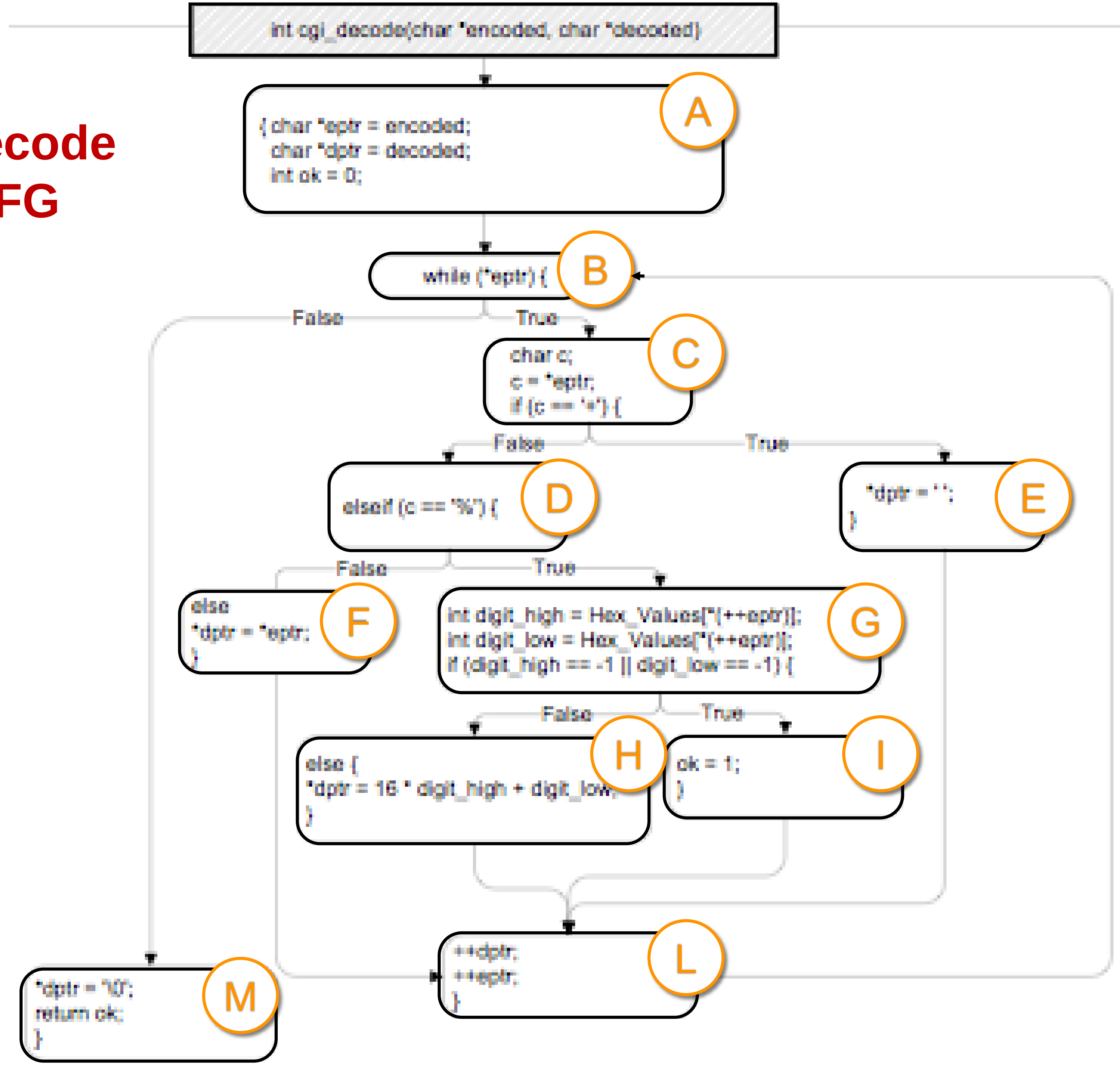
CGI_DECODE

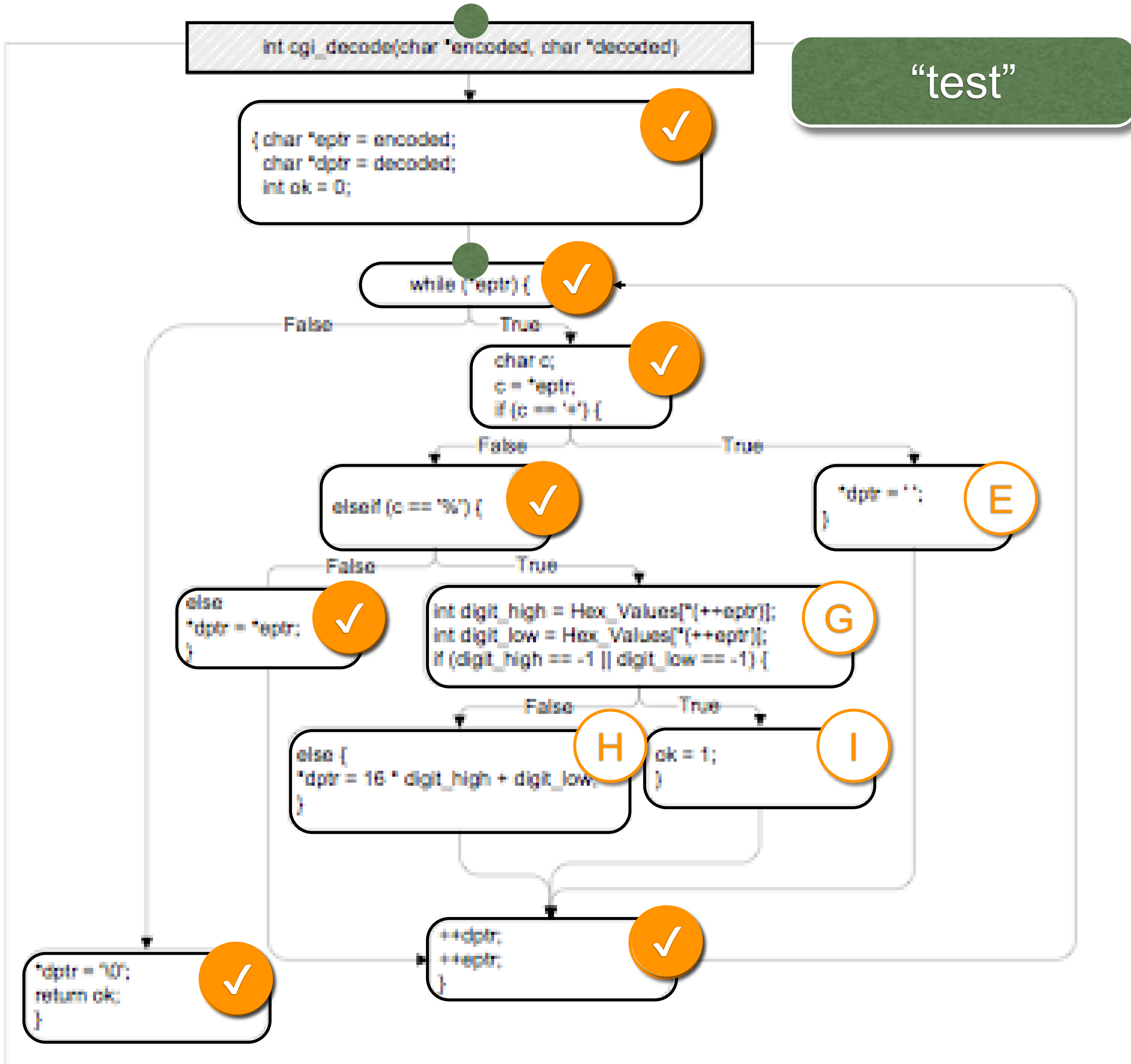
```
/**
 * @title cgi_decode
 * @desc
 * Translate a string from the CGI encoding to plain ascii text
 * '+' becomes space, %xx becomes byte with hex value xx,
 * other alphanumeric characters map to themselves
 *
 * returns 0 for success, positive for erroneous input
 * 1 = bad hexadecimal digit
 */
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
```

A

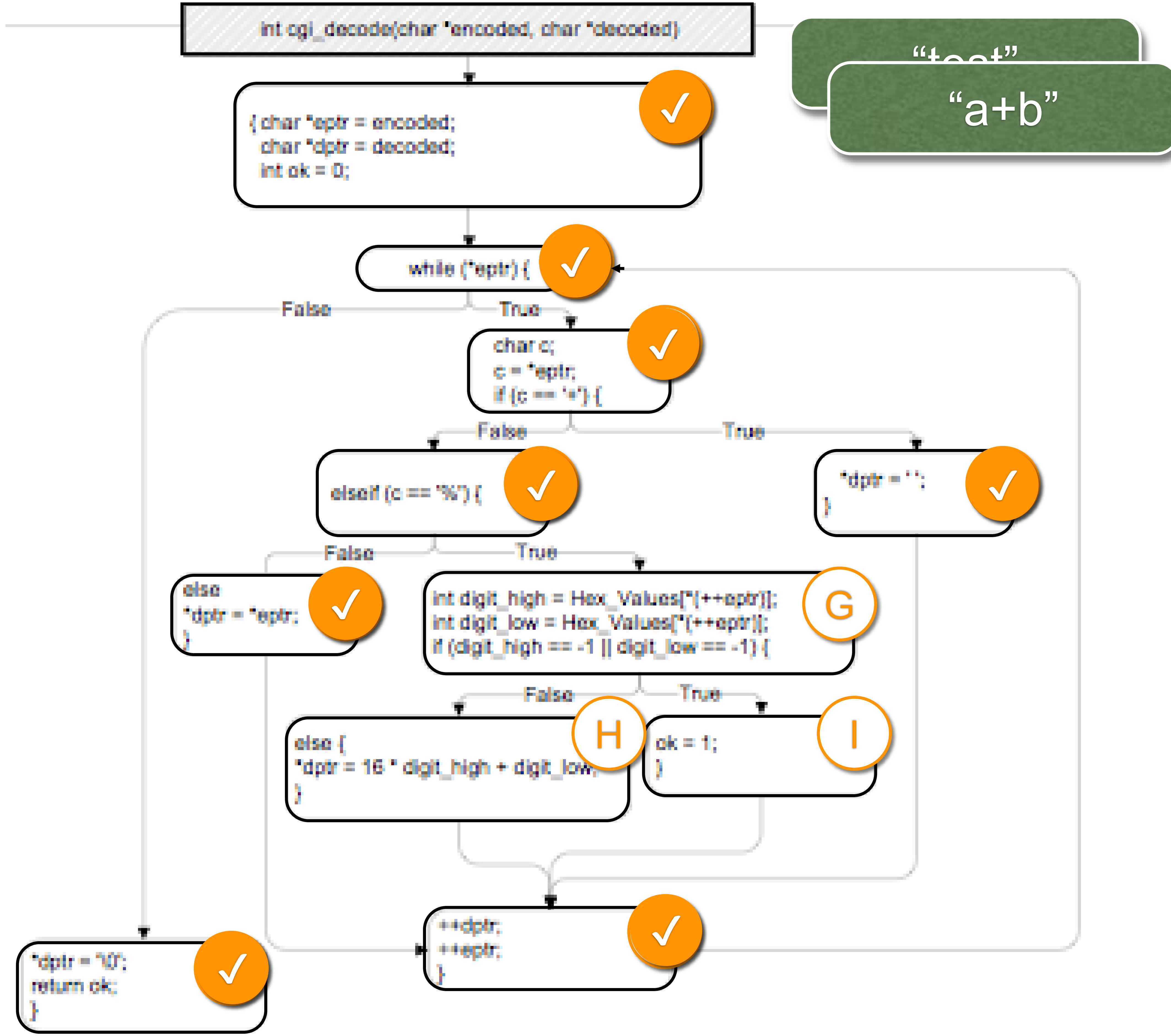
```
    while (*eptr)  /* loop to end of string ('\0' character) */
    {
        char c;
        c = *eptr;
        if (c == '+') {  /* '+' maps to blank */
            *dptr = ' ';
        } else if (c == '%') { /* '%xx' is hex for char xx */
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low  = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1)
                ok = 1; /* Bad return code */
            else
                *dptr = 16 * digit_high + digit_low;
        } else { /* All other characters map to themselves */
           *dptr = *eptr;
        }
        ++dptr; ++eptr;
    }

    *dptr = '\0';   /* Null terminator for string */
    return ok;
}
```

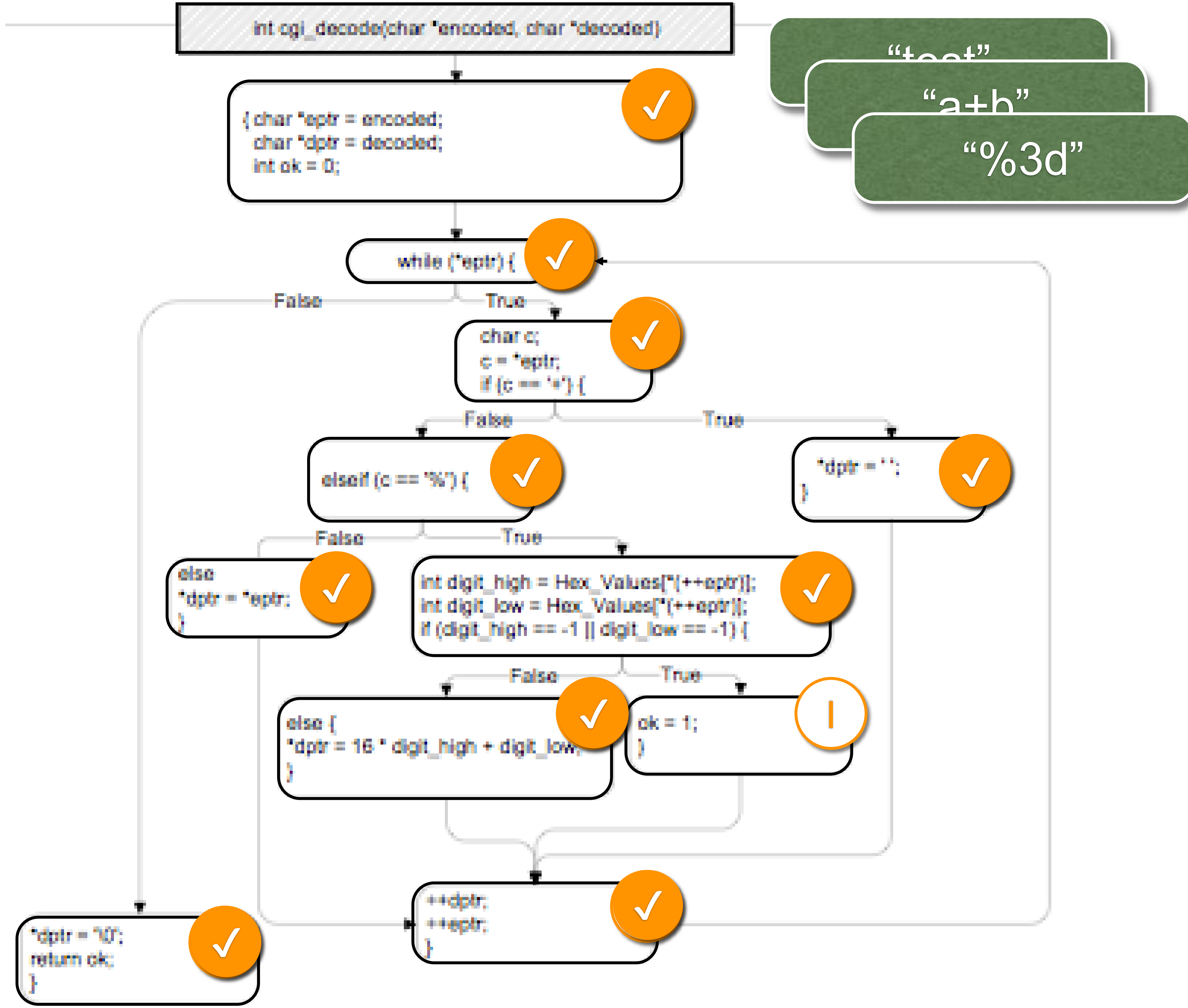
cgi_decode as a CFG



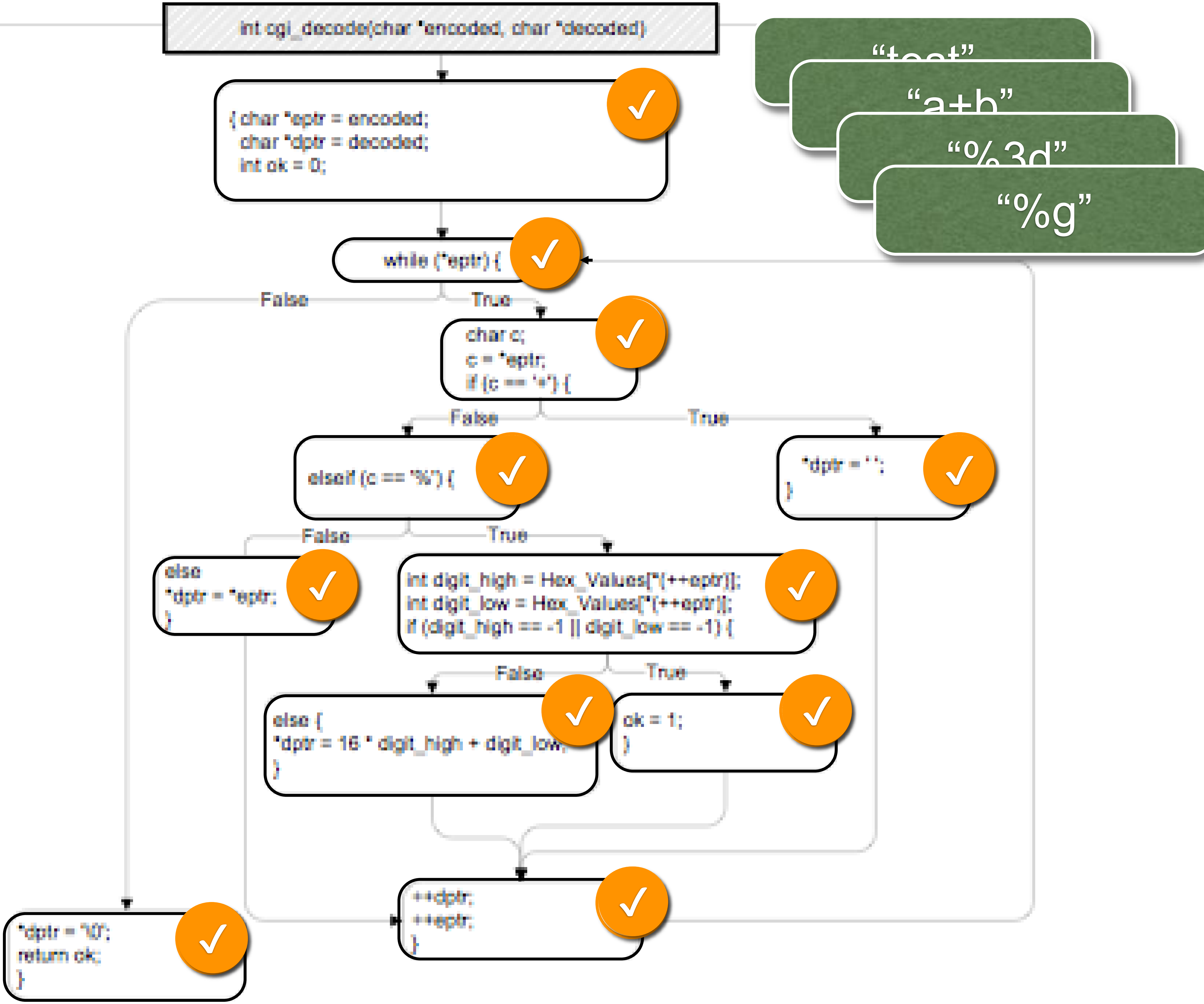


"test"
int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
*dptr = '\0';
return ok;
}
++dptr;
++eptr;
}


"a+b"
int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}



int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
"%3d"



int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
"test"
"a+b"
"%3d"
"%g"

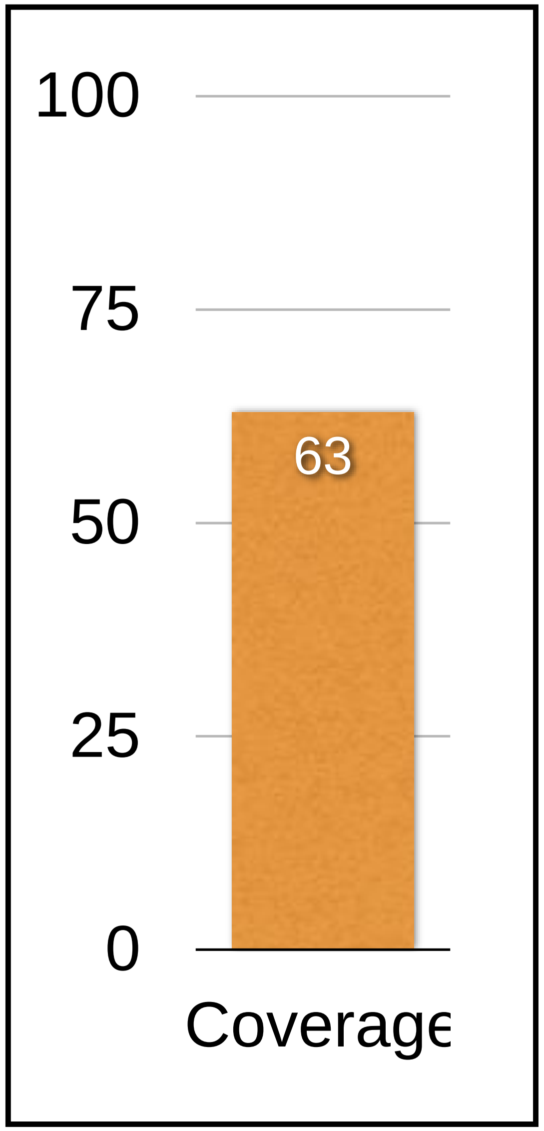
TEST ADEQUACY CRITERIA

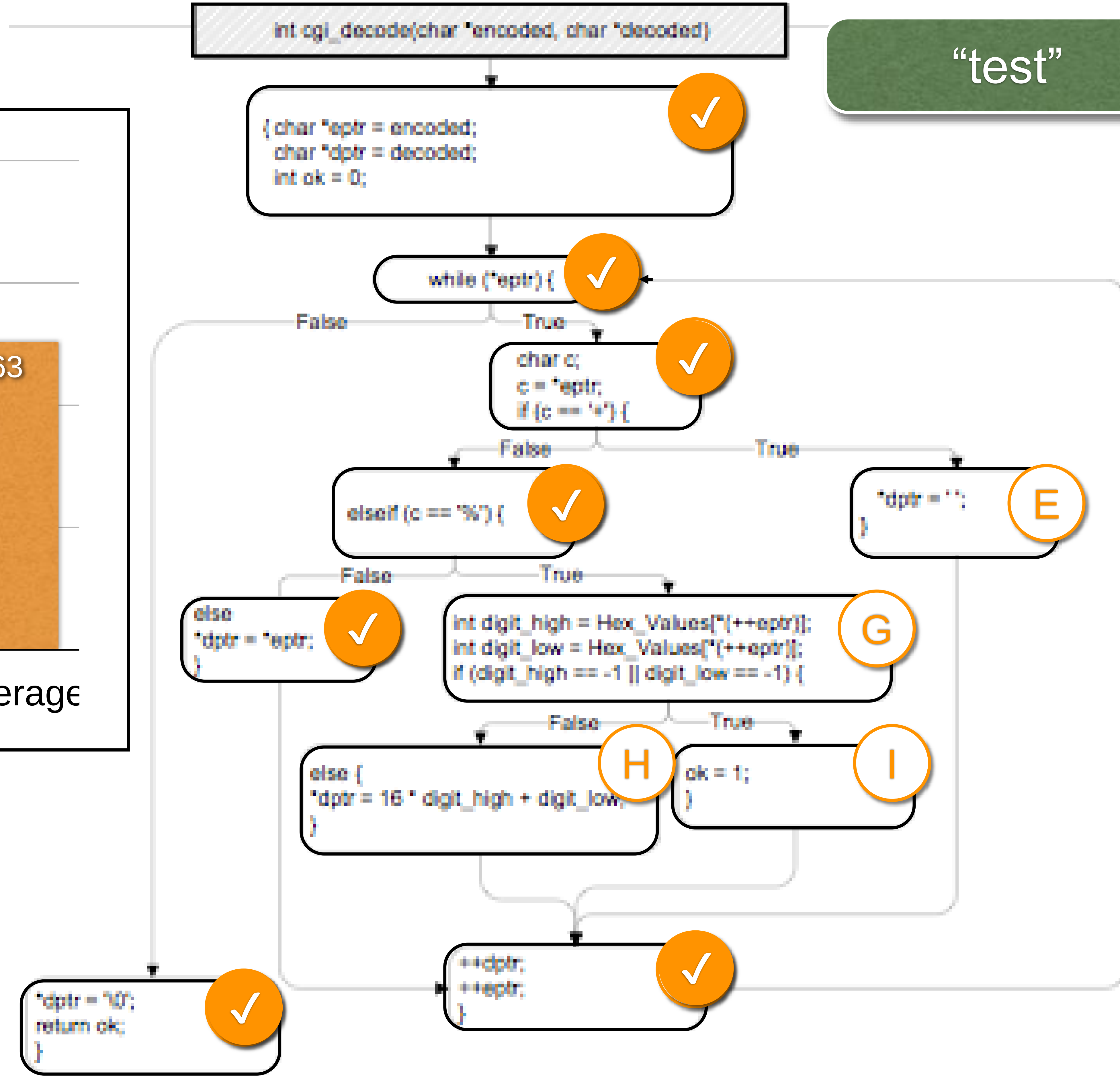
- How do we know a test suite is “good enough”?
- A *test adequacy criterion* is a Boolean predicate for a pair ⟨*program, test suite*⟩
- Usually expressed in form of a rule – e.g., `all statements must be covered`

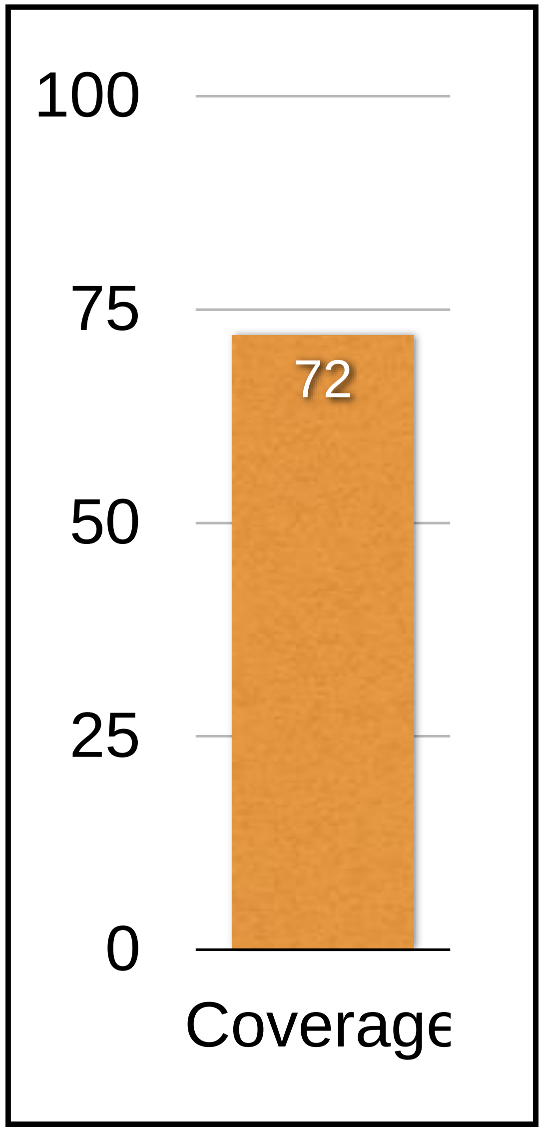
STATEMENT TESTING

- *Adequacy criterion: each statement (or node in the CFG) must be executed at least once*
- *Rationale: a defect in a statement can only be revealed by executing the defect*
- *Coverage:* $\dfrac{\text{\# executed statements}}{\text{\# statements}}$

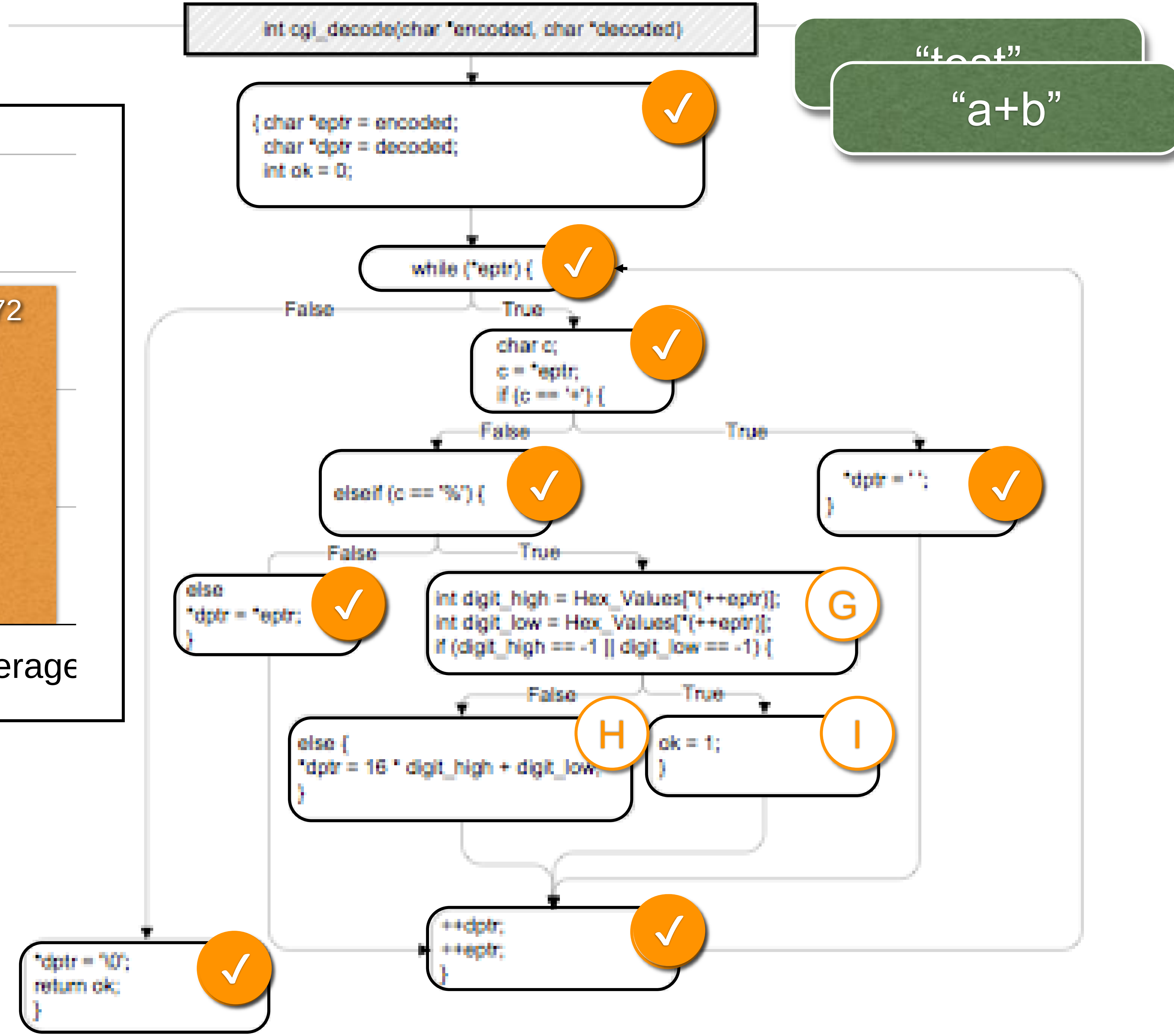
“test”


100
75
50
25
0
63
Coverage

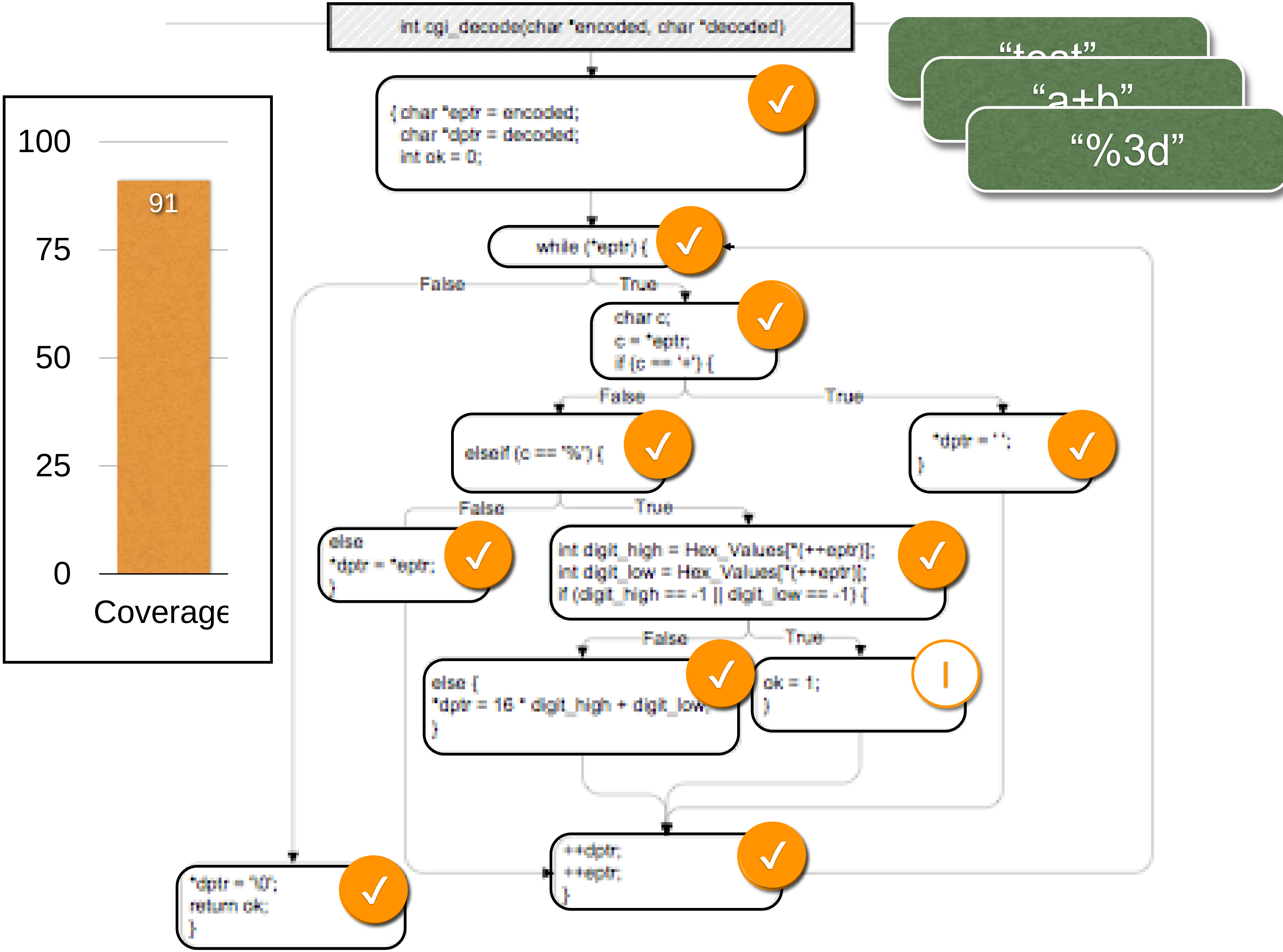

int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}



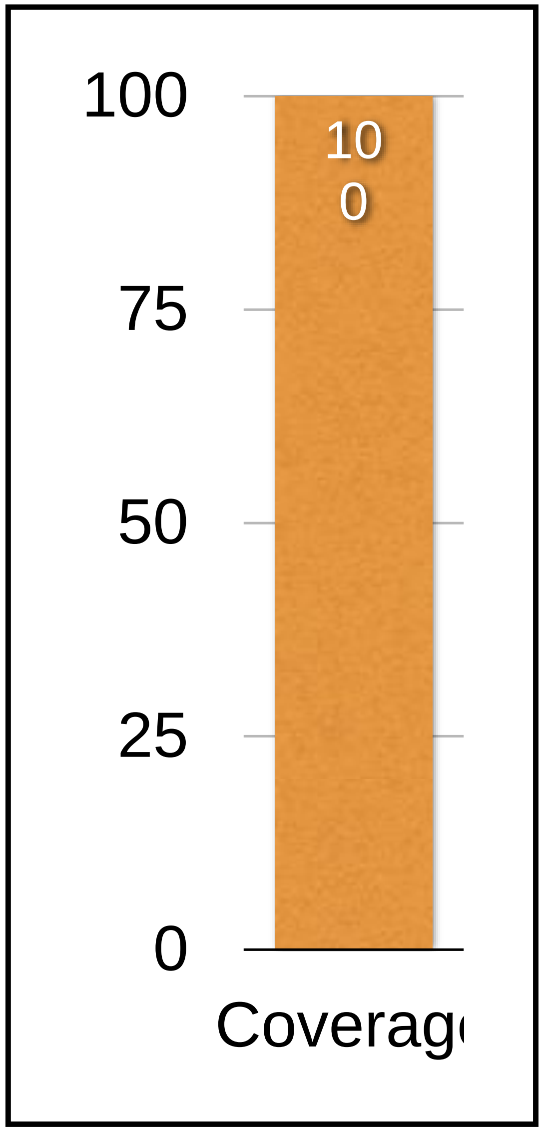
100
75
50
25
0
72
Coverage

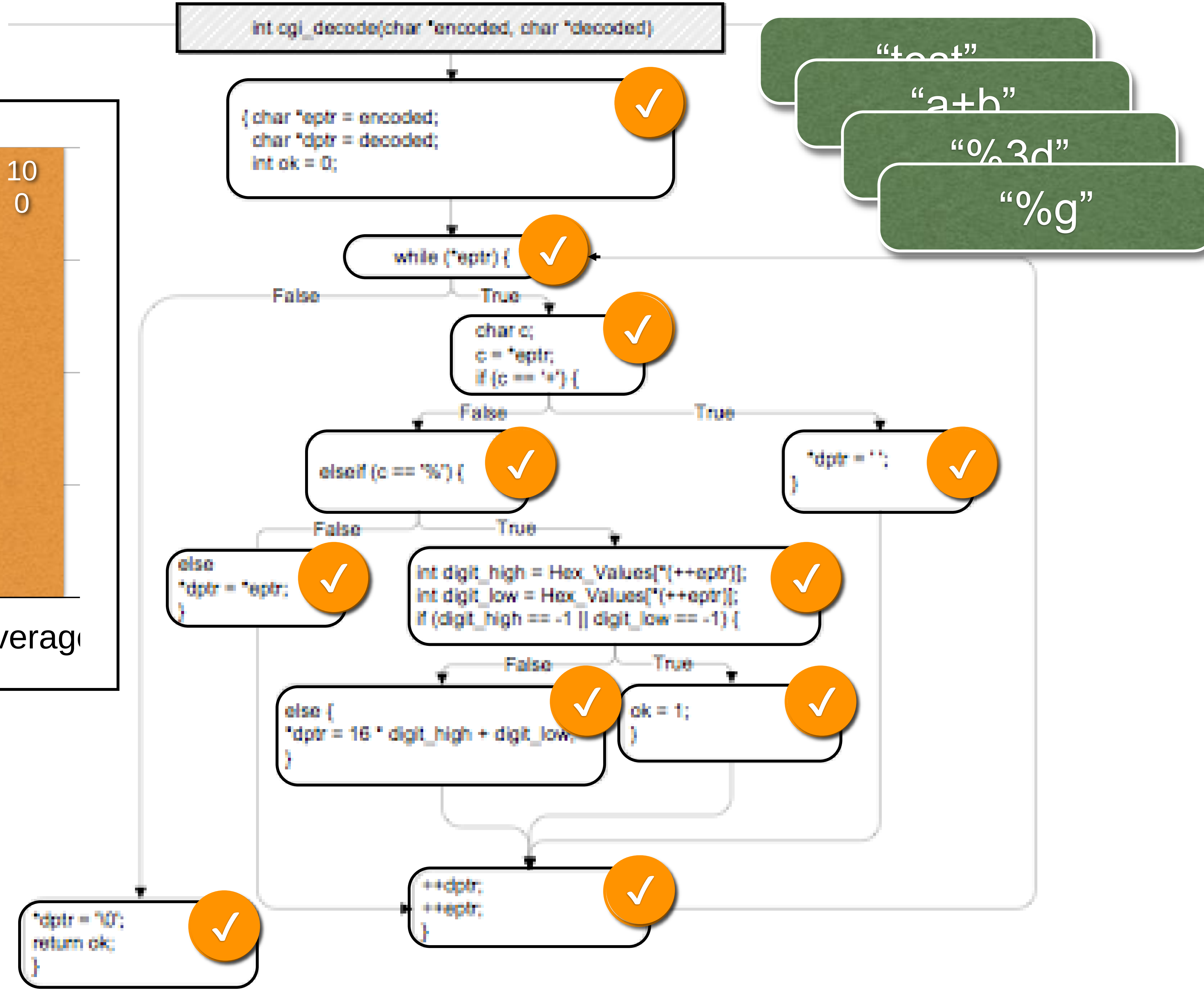


"test"
"a+b"
int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}



100
75
50
25
0
91
Coverage
int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
"a+b"
"%3d"


100
75
50
25
0
10
0
Coverag


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
“%g”

COMPUTING COVERAGE

- Coverage is computed automatically while the program executes
- Requires *instrumentation* at compile time

  For example with GCC, use options
  `-ftest-coverage -fprofile-arcs`

- After execution, *coverage tool* assesses and summarizes results

  Again with GCC,
  use `gcov source-file` to obtain readable `.gcov` file

GCOV COVERAGE OUTPUT FOR cgi_decode

Pippin: cgi_encode — less — 80×24

Number of executions

```
        4:   18:    int ok = 0;
        -:   19:
       38:   20:    while (*eptr)  /* loop to end of string ('\0' character) */
        -:   21:    {
        -:   22:        char c;
       30:   23:        c = *eptr;
       30:   24:        if (c == '+') {  /* '+' maps to blank */
        1:   25:            *dptr = ' ';
       29:   26:        } else if (c == '%') { /* '%xx' is hex for char xx */
        3:   27:            int digit_high = Hex_Values[*(++eptr)];
        3:   28:            int digit_low  = Hex_Values[*(++eptr)];
        5:   29:            if (digit_high == -1 || digit_low == -1)
        2:   30:                ok = 1; /* Bad return code */
        -:   31:            else
        1:   32:                *dptr = 16 * digit_high + digit_low;
        -:   33:        } else { /* All other characters map to themselves */
       26:   34:           *dptr = *eptr;
        -:   35:        }
       30:   36:        ++dptr; ++eptr;
        -:   37:    }
        4:   38:    *dptr = '\0';   /* Null terminator for string */
        4:   39:    return ok;
        -:   40:}
(END)
```

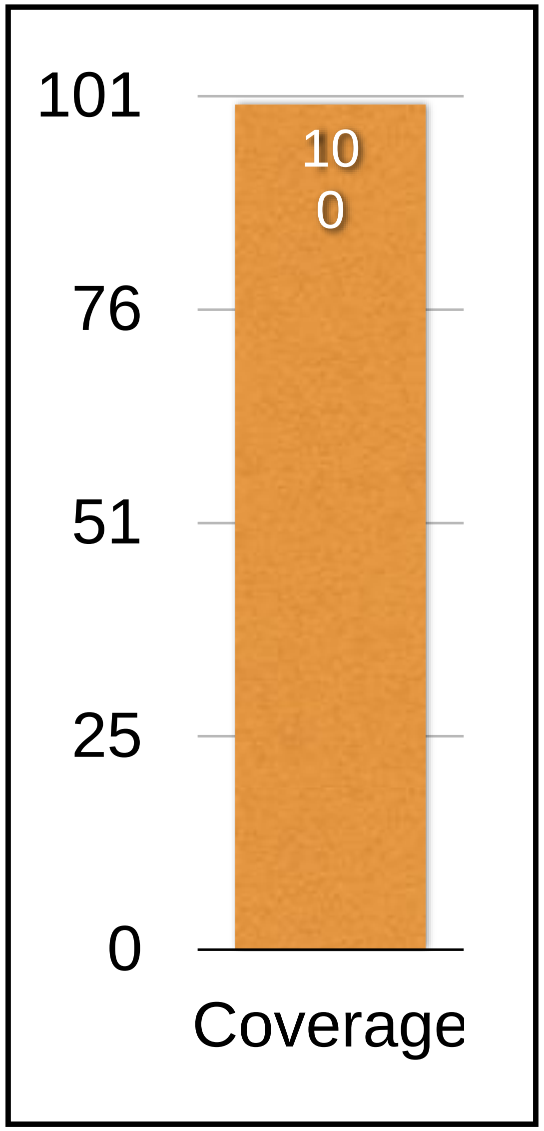
ADEQUACY OF A TEST SUITE

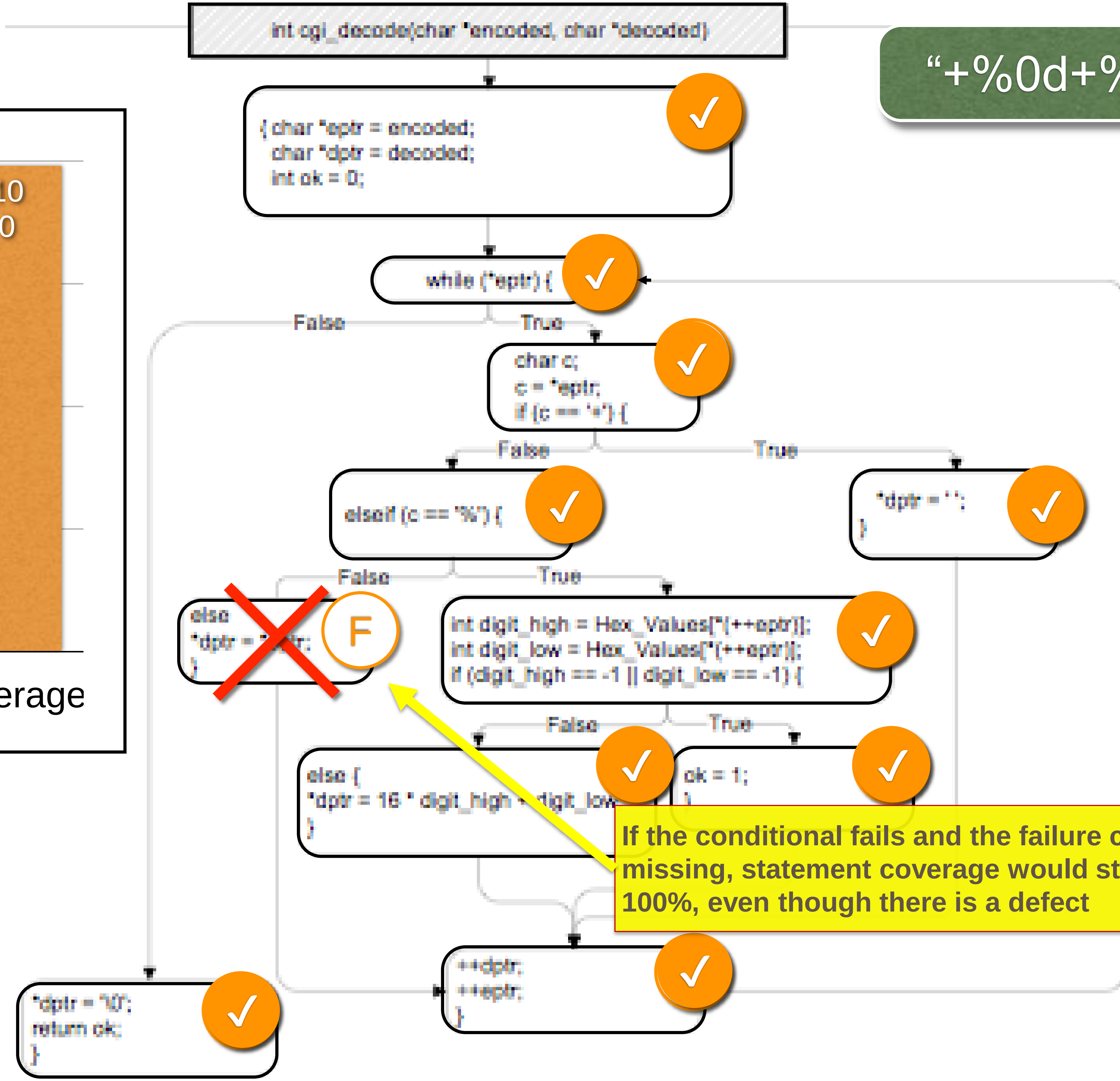
1. Statement testing is a simple criterion

2. Branch testing is another a criterion.
   - *It subsumes statement testing.*

→*if the branch testing criterion is satisfied by a pair ⟨program, test suite⟩, then so is the statement testing criterion for the same pair.*

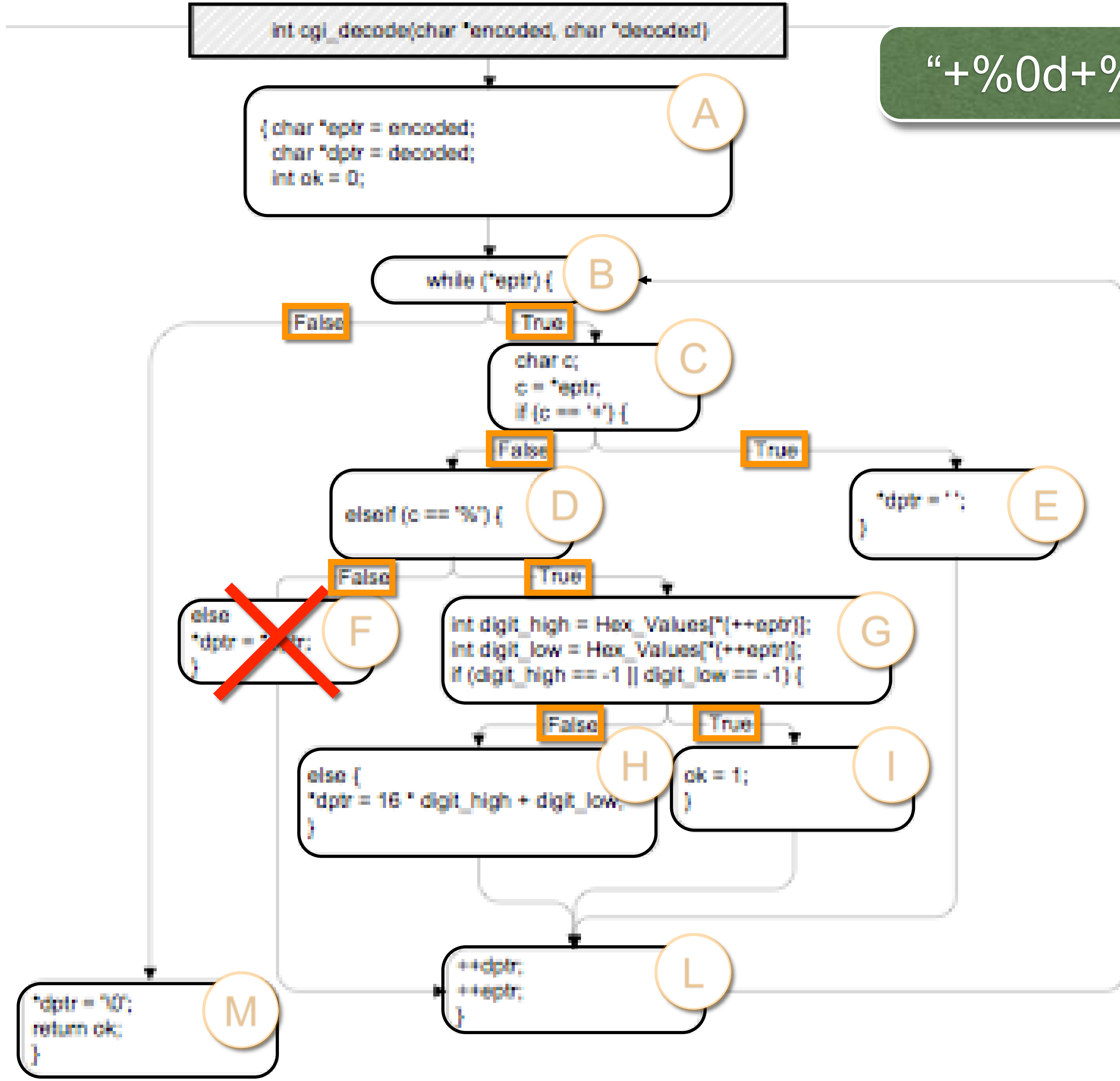
“+%0d+%4j”





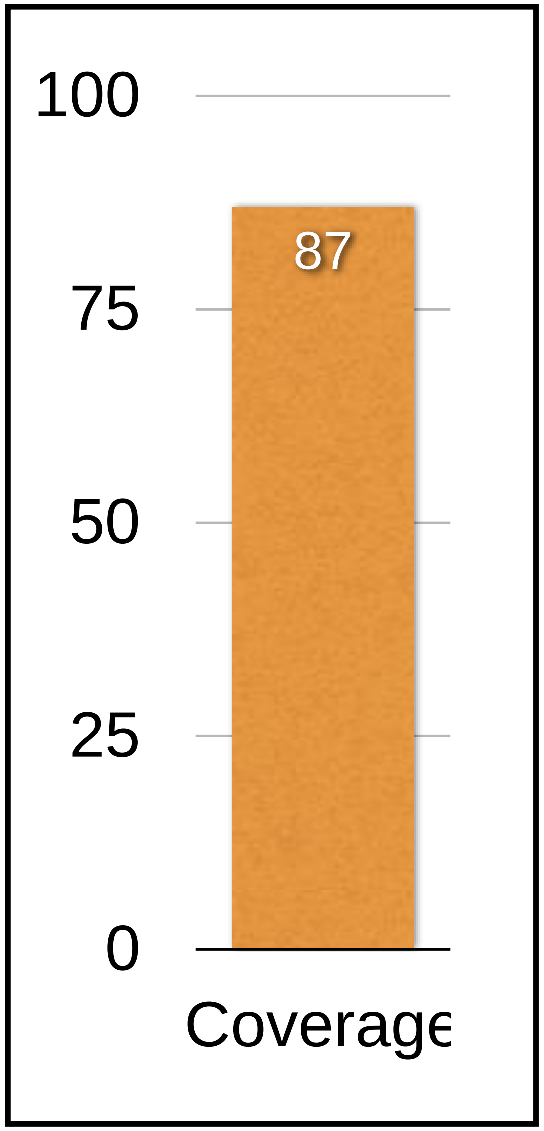
If the conditional fails and the failure case coding is missing, statement coverage would still get to 100%, even though there is a defect

“+%0d+%4j”

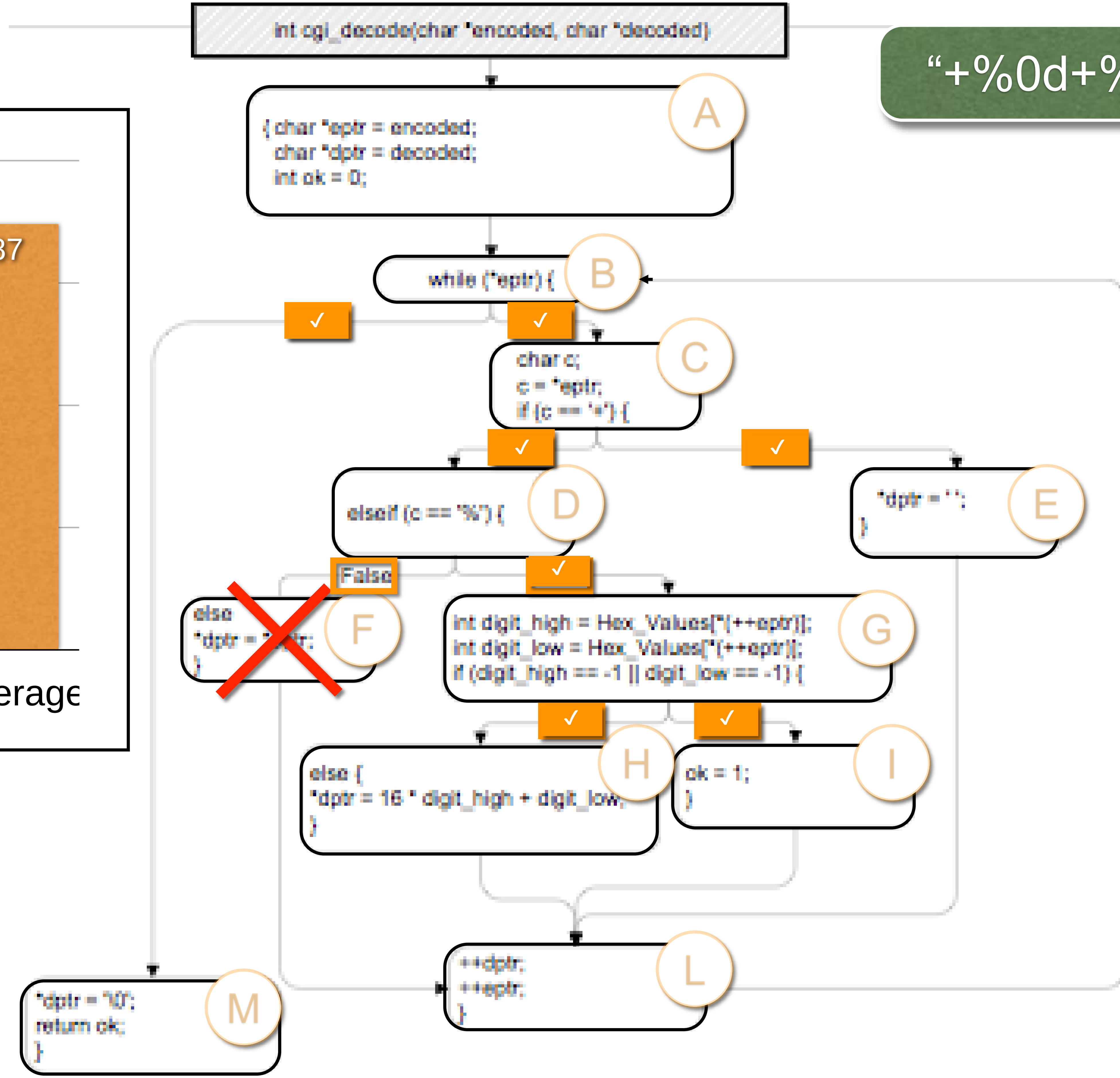


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
False
True
char c;
c = *eptr;
if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
True
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M

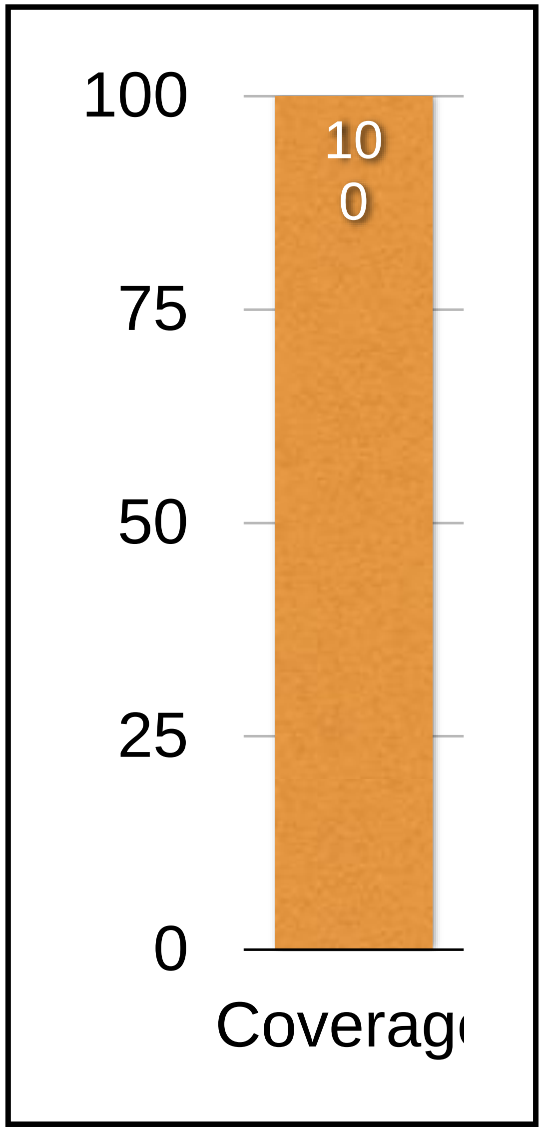
“+%0d+%4j”

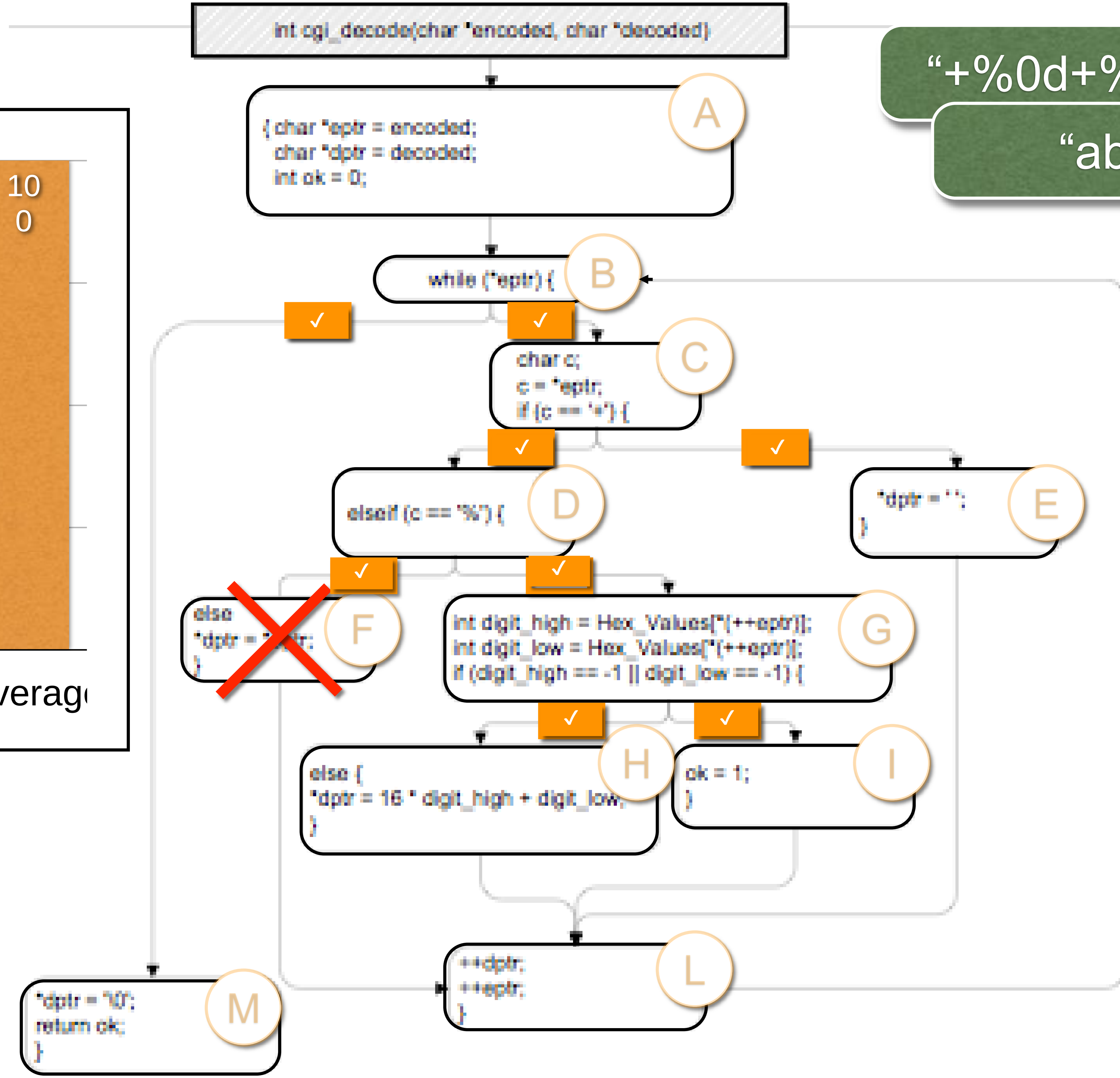


100
75
50
25
0
87
Coverage



int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
char c;
c = *eptr;
if (c == '+') {
C
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
else
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M
✓


100
75
50
25
0
10
0
Coverag


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
char c;
c = *eptr;
if (c == '+') {
C
elseif (c == '%') {
D
*dptr = ' ';
}
E
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M

"+%0d+%4j"

"abc"

BRANCH TESTING

- Adequacy criterion:

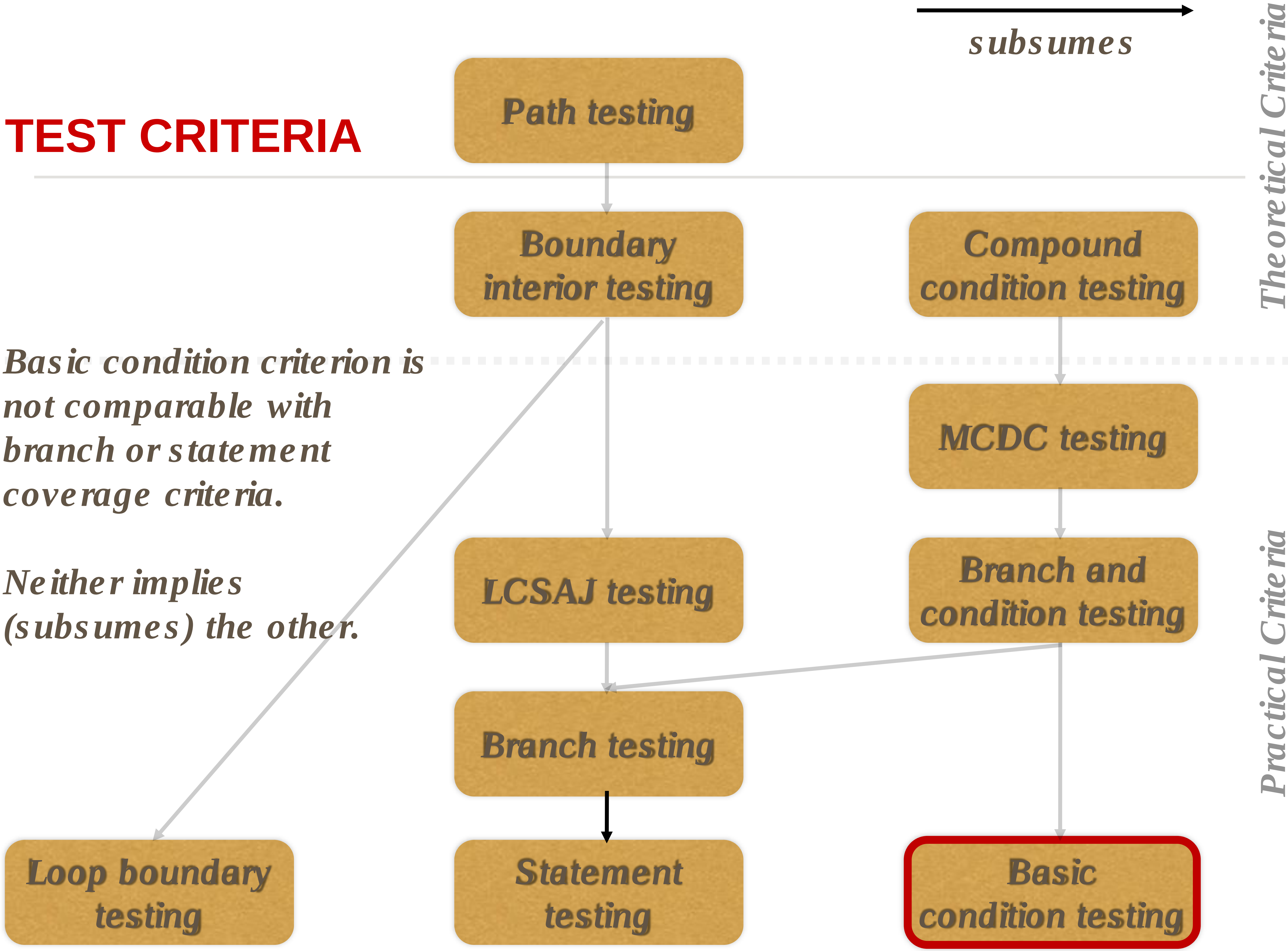
  `each branch in the CFG must be executed at least once`

- Coverage: $\dfrac{\texttt{\# executed branches}}{\texttt{\# branches}}$

- Subsumes statement testing criterion because traversing all edges implies traversing all nodes

- Most widely used criterion in industry

TEST CRITERIA



Basic condition criterion is not comparable with branch or statement coverage criteria.

Neither implies (subsumes) the other.

CONDITION TESTING

- Consider

  `(digit_high == 1 || digit_low == -1)`

- Branch adequacy criterion can be achieved by changing only `digit_low`
  - *i.e., the defective sub-expression may never determine the result*

- Faulty sub-condition is never tested although we tested both outcomes of the branch

- Key idea: cover *individual conditions in compound boolean expressions*

  e.g., both parts of `digit_high == 1 || digit_low == -1`

CONDITION TESTING

- ***Adequacy criterion***
  `each basic condition must be evaluated at least once`

- ***Coverage:***

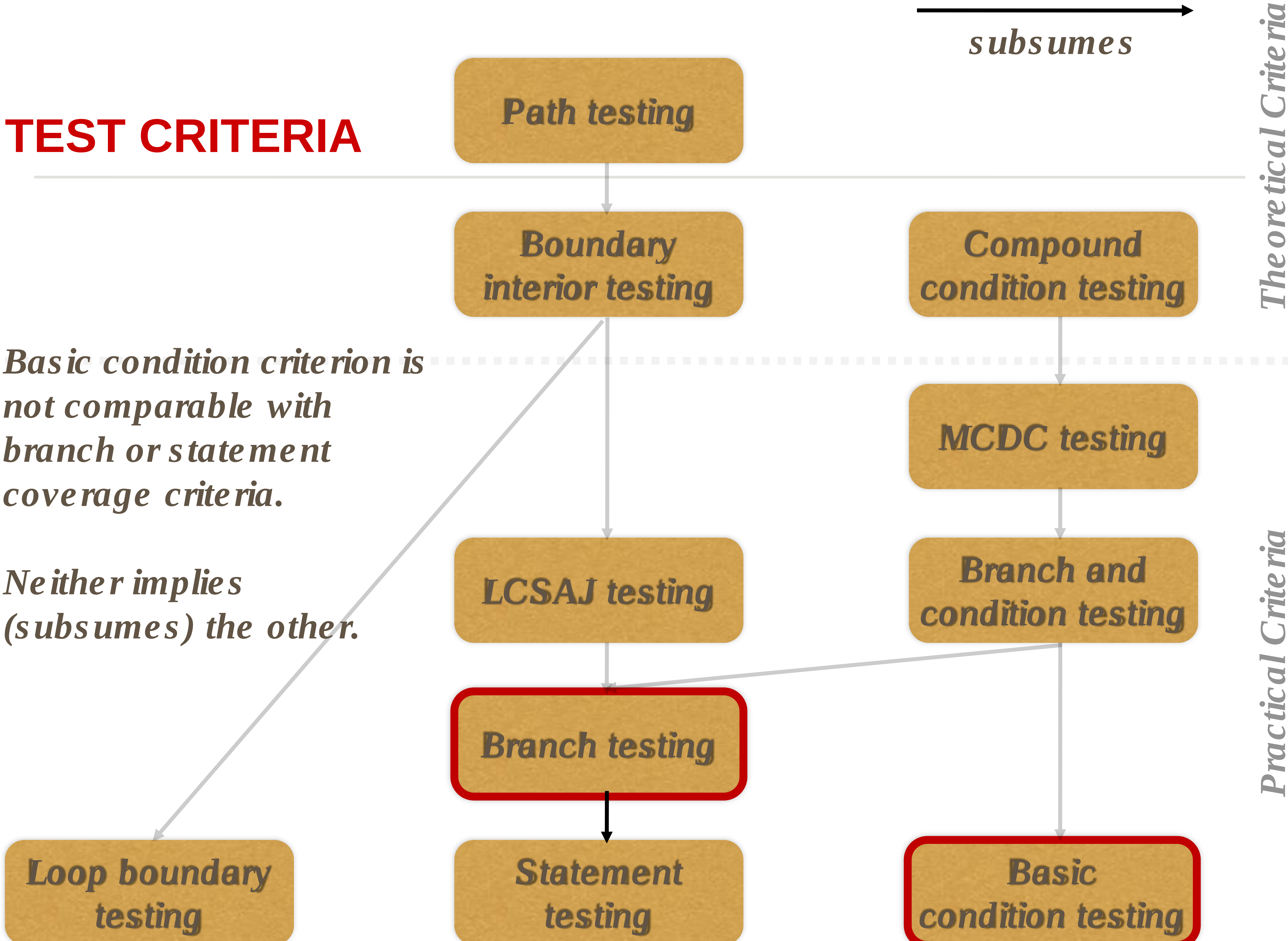
$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

- In cgi_decode, Test Case "test+%9k%k9" gives 100% basic condition ***coverage, but only 87% branch coverage***

TEST CRITERIA

Basic condition criterion is not comparable with branch or statement coverage criteria.

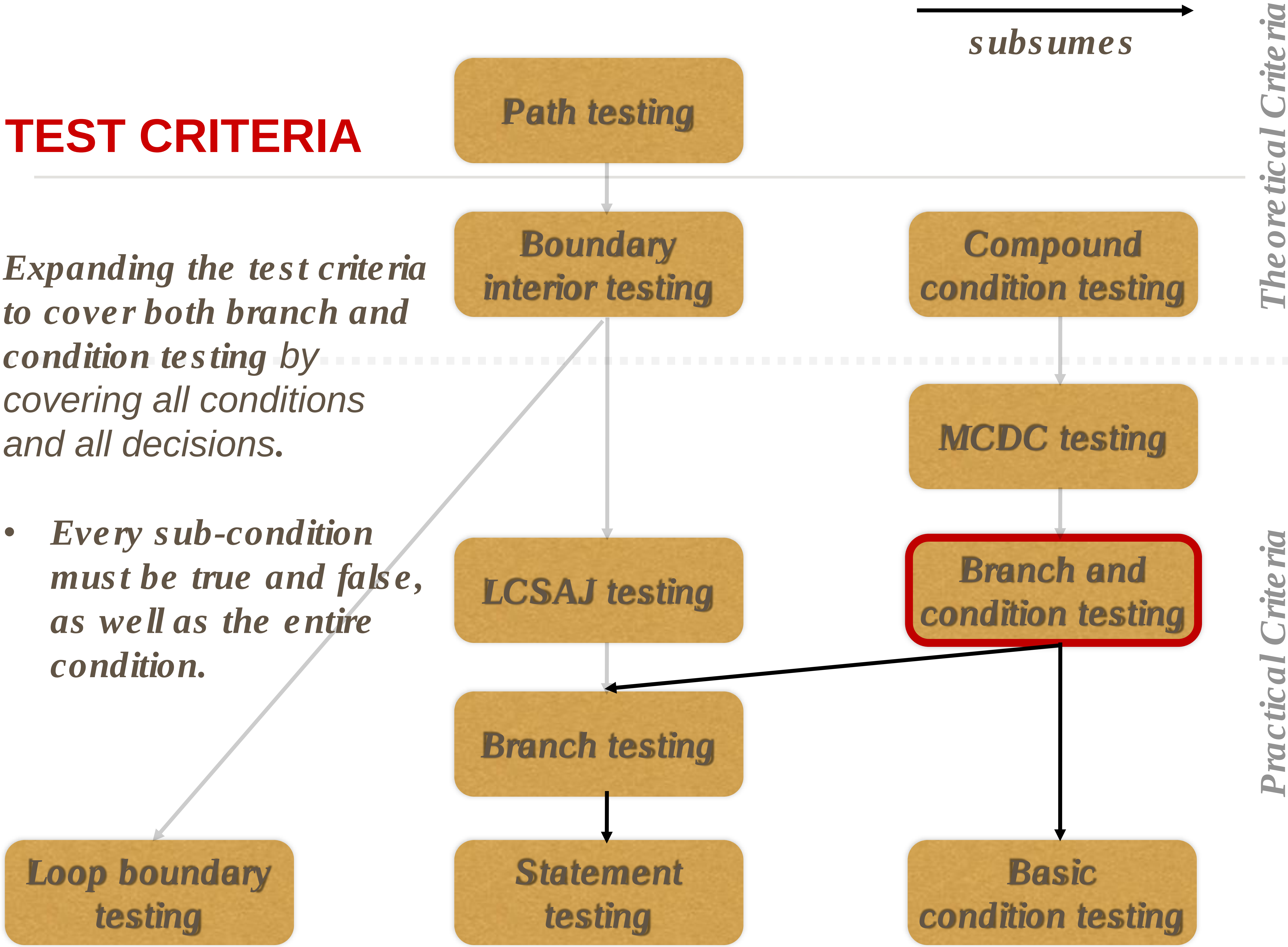
Neither implies (subsumes) the other.



TEST CRITERIA

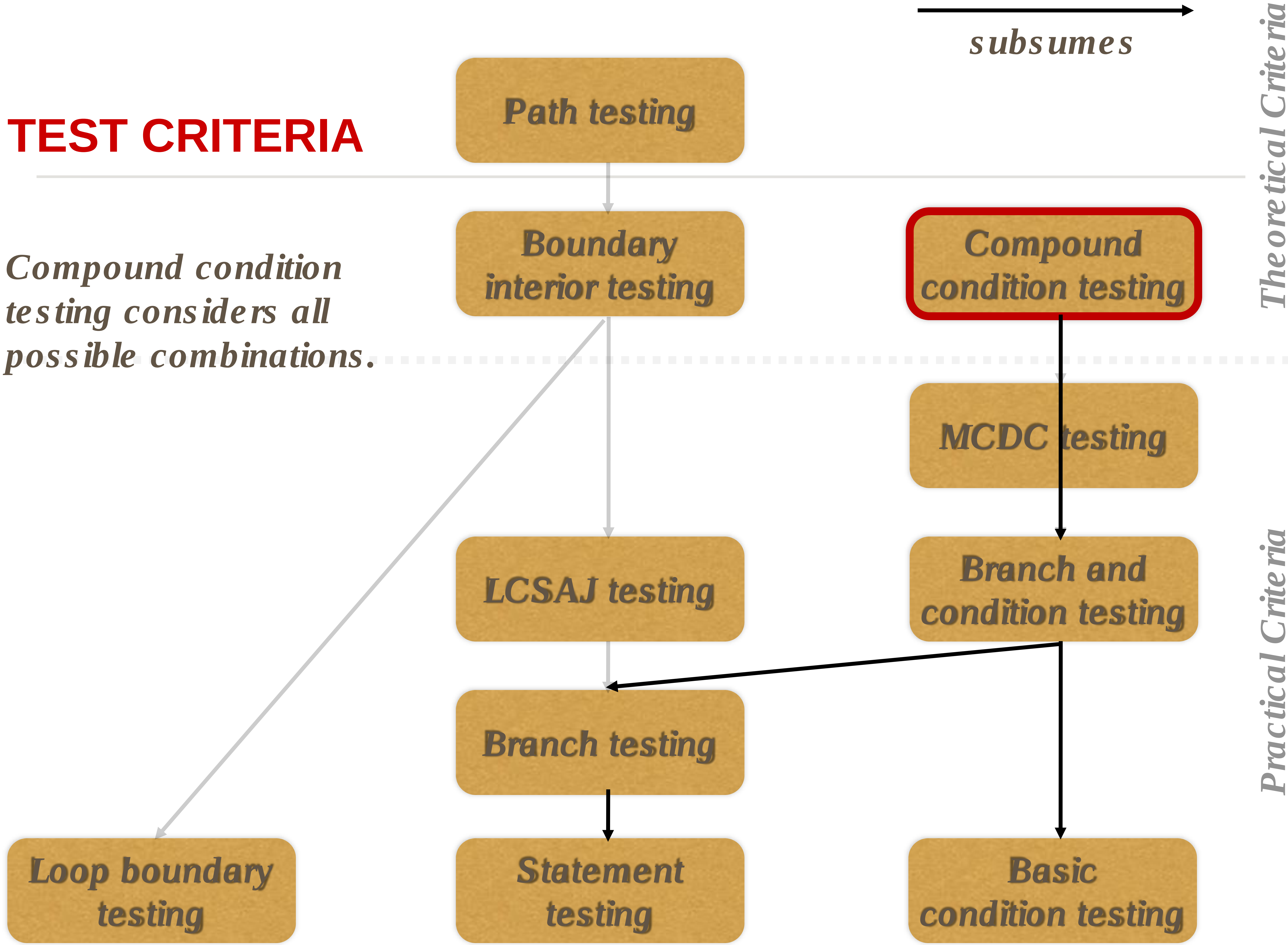
Expanding the test criteria to cover both branch and condition testing by covering all conditions and all decisions.

- *Every sub-condition must be true and false, as well as the entire condition.*



TEST CRITERIA

Compound condition testing considers all possible combinations.



COMPOUND CONDITION TESTING EXAMPLE

- Consider

  $(((a \lor b) \land c) \lor d) \land e)$

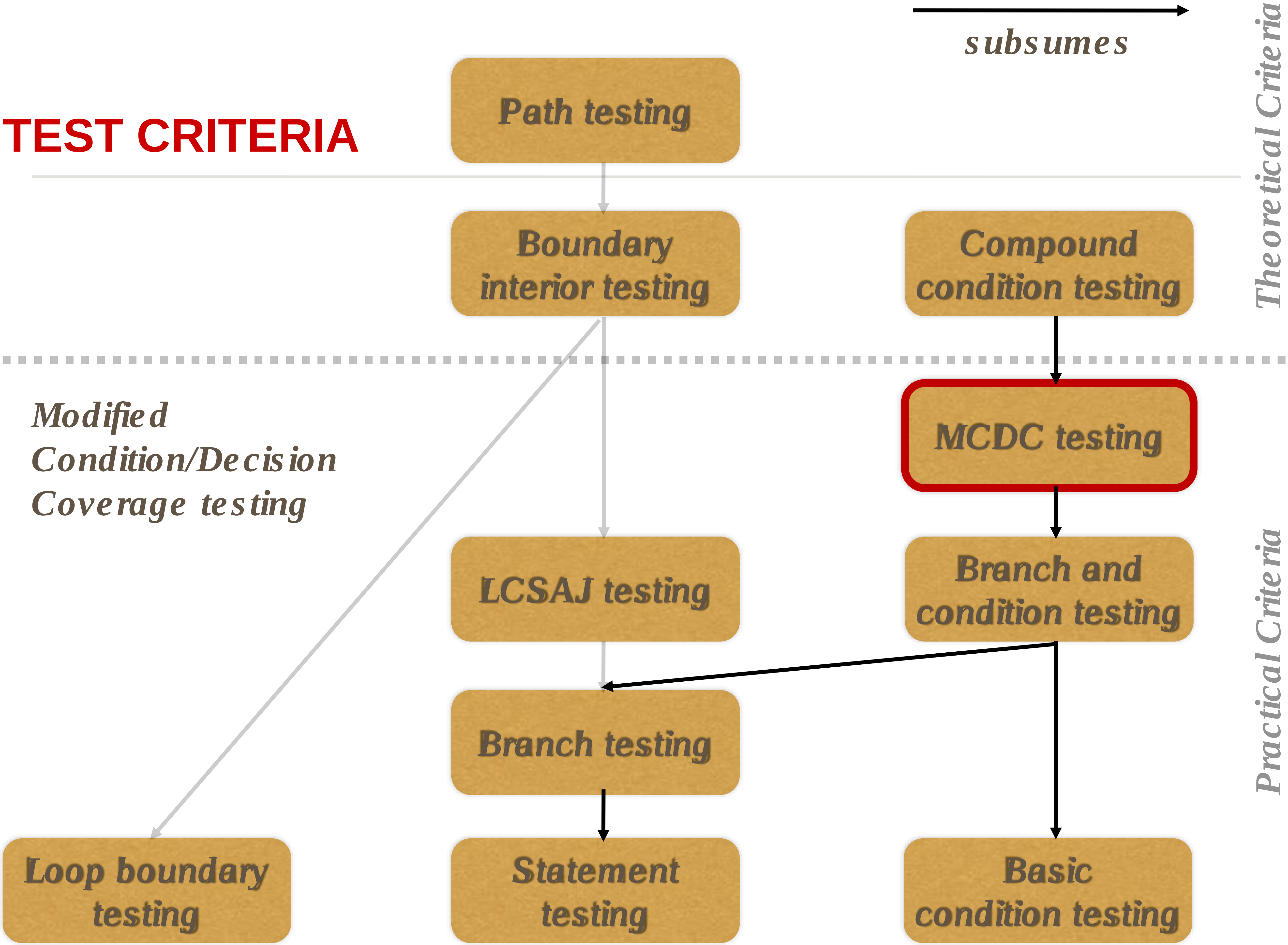
- This requires 13 tests to cover all possible combinations

- In general, this involves a combinatorial explosion!
  - Why compound condition testing is a theoretical, rather than a practical, criterion

Test Case	a	b	c	d	e
(1)	True	–	True	–	True
(2)	False	True	True	–	True
(3)	True	–	False	True	True
(4)	False	True	False	True	True
(5)	False	False	–	True	True
(6)	True	–	True	–	False
(7)	False	True	True	–	False
(8)	True	–	False	True	False
(9)	False	True	False	True	False
(10)	False	False	–	True	False
(11)	True	–	False	False	–
(12)	False	True	False	False	–
(13)	False	False	–	False	–

TEST CRITERIA

Modified Condition/Decision Coverage testing



MCDC TESTING
MODIFIED CONDITION DECISION COVERAGE

- Key idea: Test *important combinations* of conditions, avoiding exponential blowup

- A combination is “important” if each basic condition is shown to independently affect the outcome of each decision

MC/DC TESTING
MODIFIED CONDITION DECISION COVERAGE

- For each basic condition *C*, we need two test cases: T_1 and T_2
- Values of all *evaluated* conditions except *C* are the same
- Compound condition as a whole evaluates to
  TRUE for T_1 and FALSE for T_2
- A good balance of thoroughness and test size (and therefore widely used)
- used in avionics software development guidance DO-178B, DO-178C to ensure adequate testing of the most critical (Level A) software

MC/DC TESTING
MODIFIED CONDITION DECISION COVERAGE

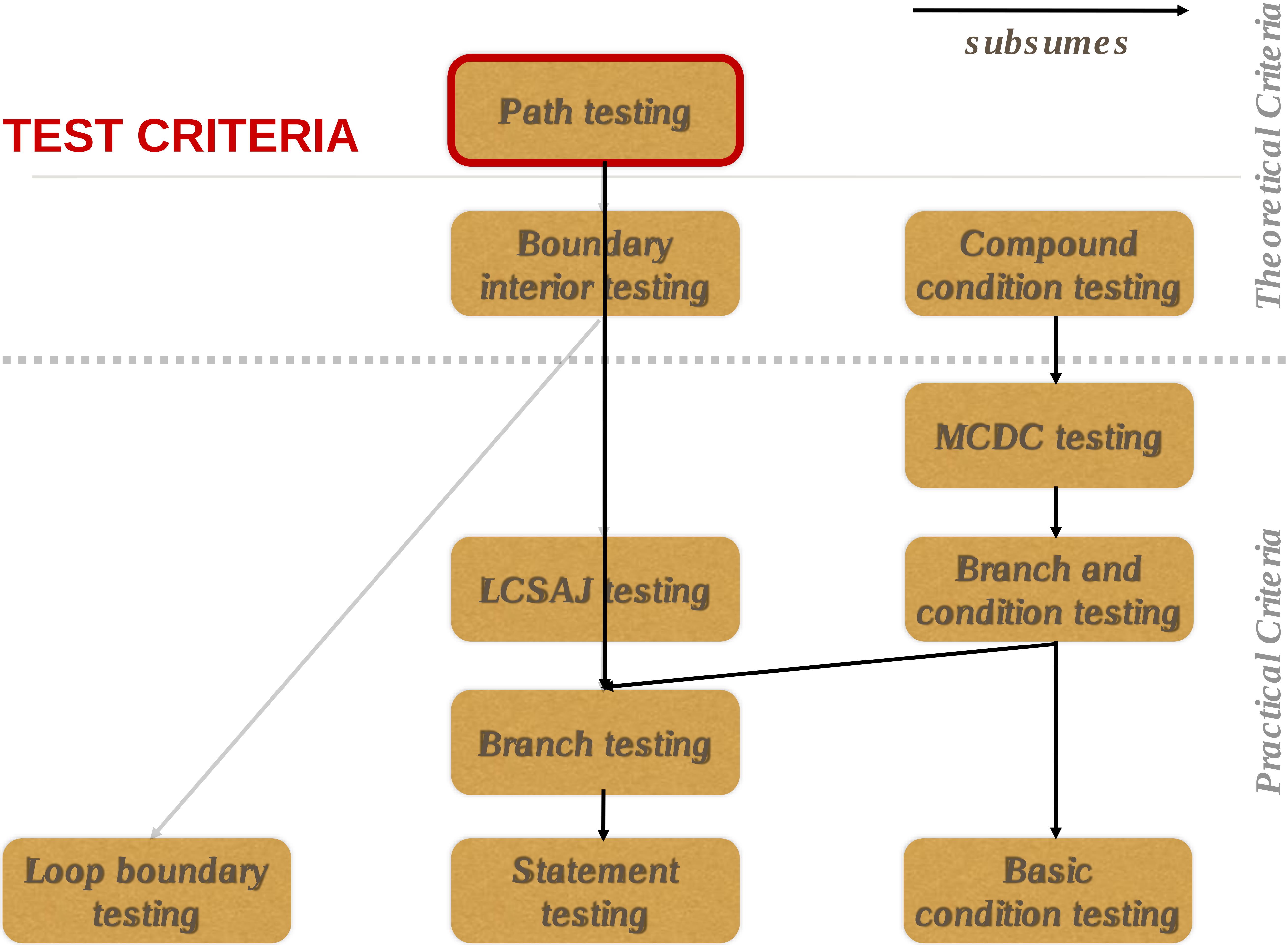
For `(((a ∨ b) ∧ c) ∨ d) ∧ e)`

We need six tests to cover MCDC combinations to effect 100% coverage

	a	b	c	d	e	Decision
(1)	True	–	True	–	True	True
(2)	False	True	True	–	True	True
(3)	True	–	False	True	True	True
(6)	True	–	True	–	False	False
(11)	True	–	False	False	–	False
(13)	False	False	–	False	–	False

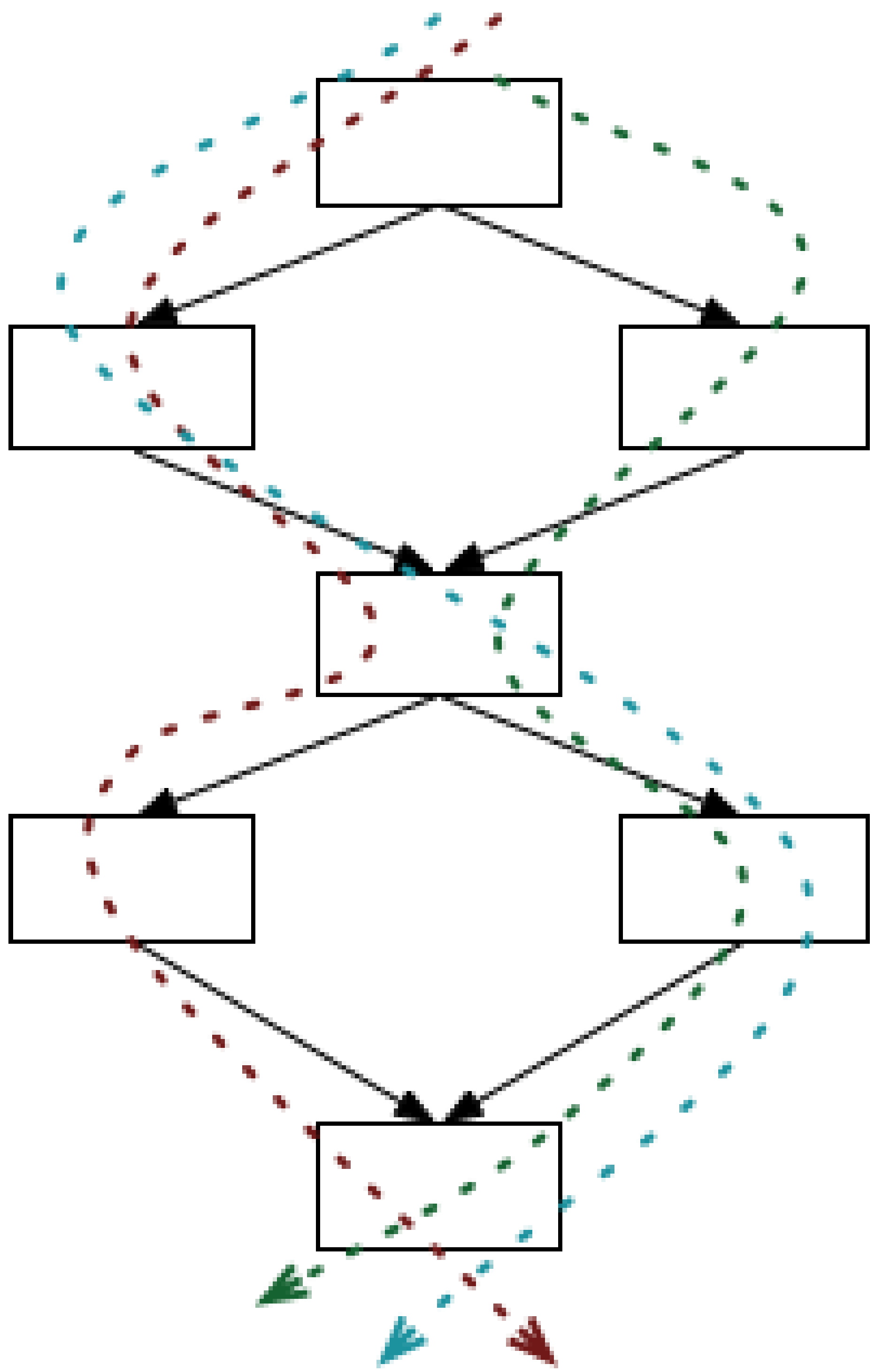
Underlined values independently affect the outcome of the decision.

TEST CRITERIA



BEYOND INDIVIDUAL BRANCHES: PATH TESTING

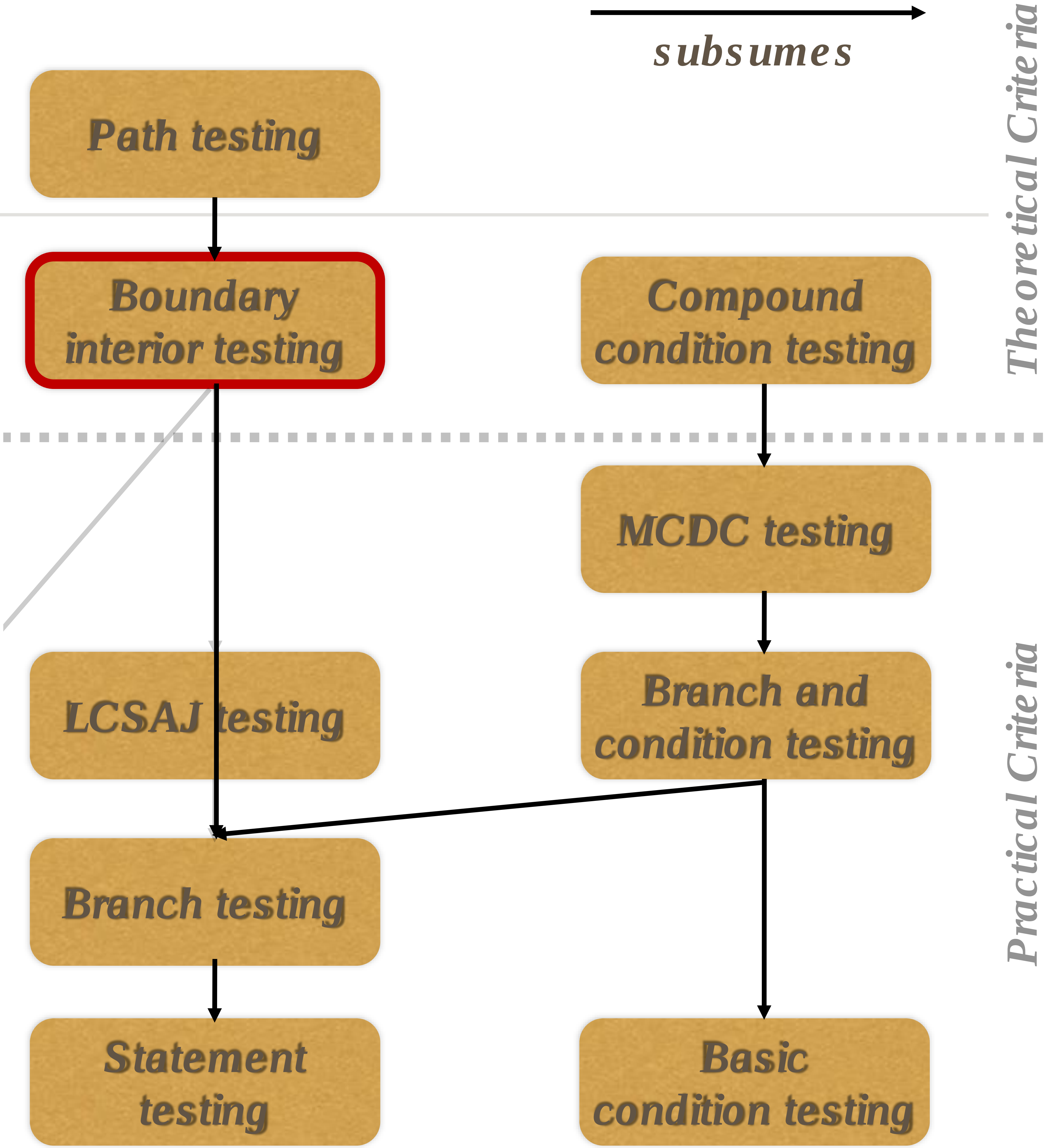
- Key idea: explore all paths in the code
  - i.e., sequences of branches
- Since loops give rise to an unbounded number of paths, this is generally not feasible and therefore just a theoretical criterion.
- Its advantage, though, is that it subsumes almost all criteria

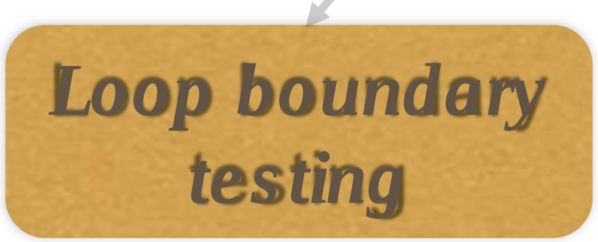


TEST CRITERIA

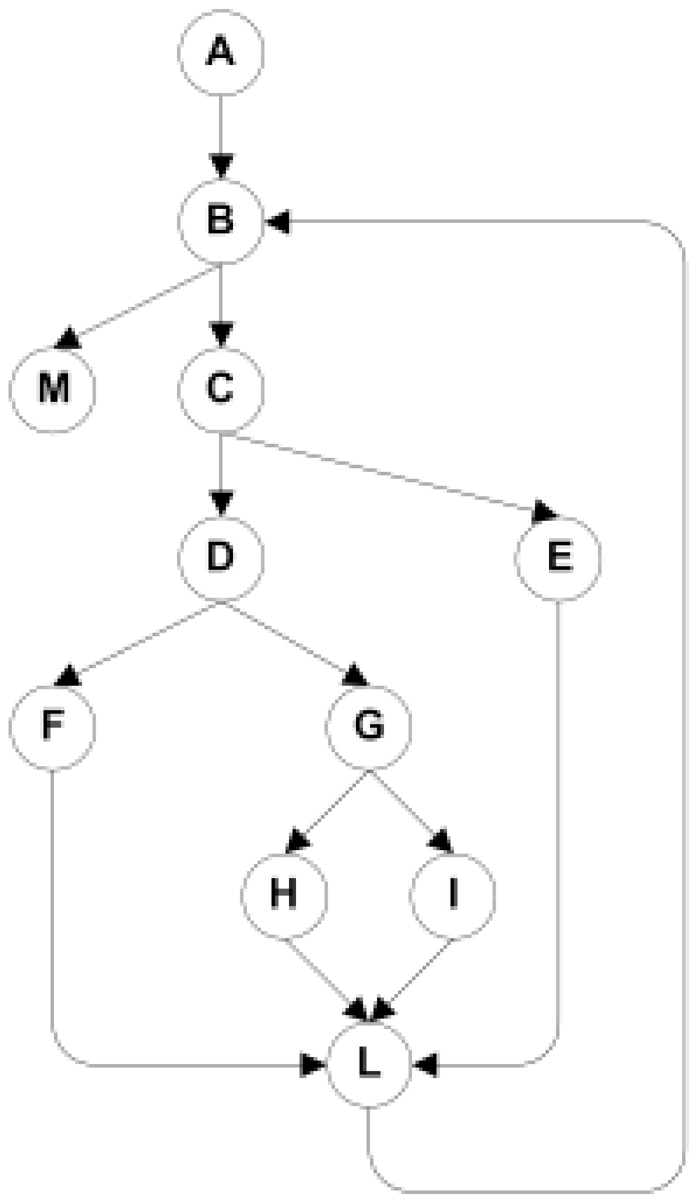
Boundary interior testing groups together paths that differ only in the sub-path they follow when repeating the body of a loop.

In other words, we follow each path in the CFG up to the first repeated node.

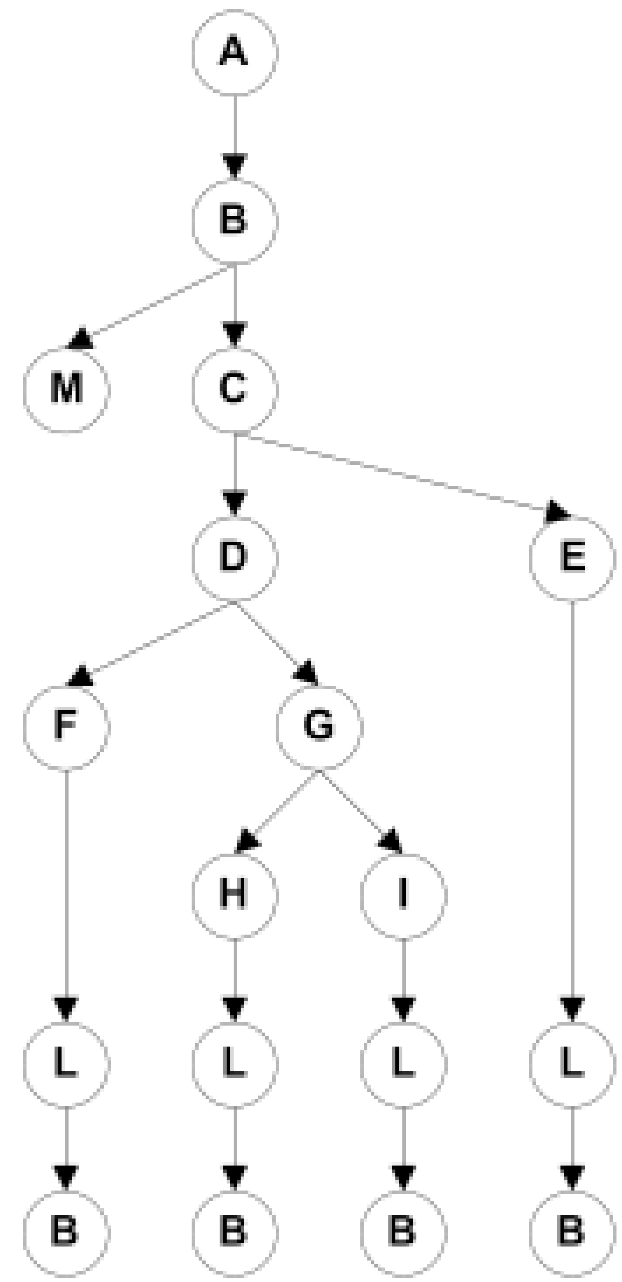




BOUNDARY INTERIOR ADEQUACY FOR cgi_decode

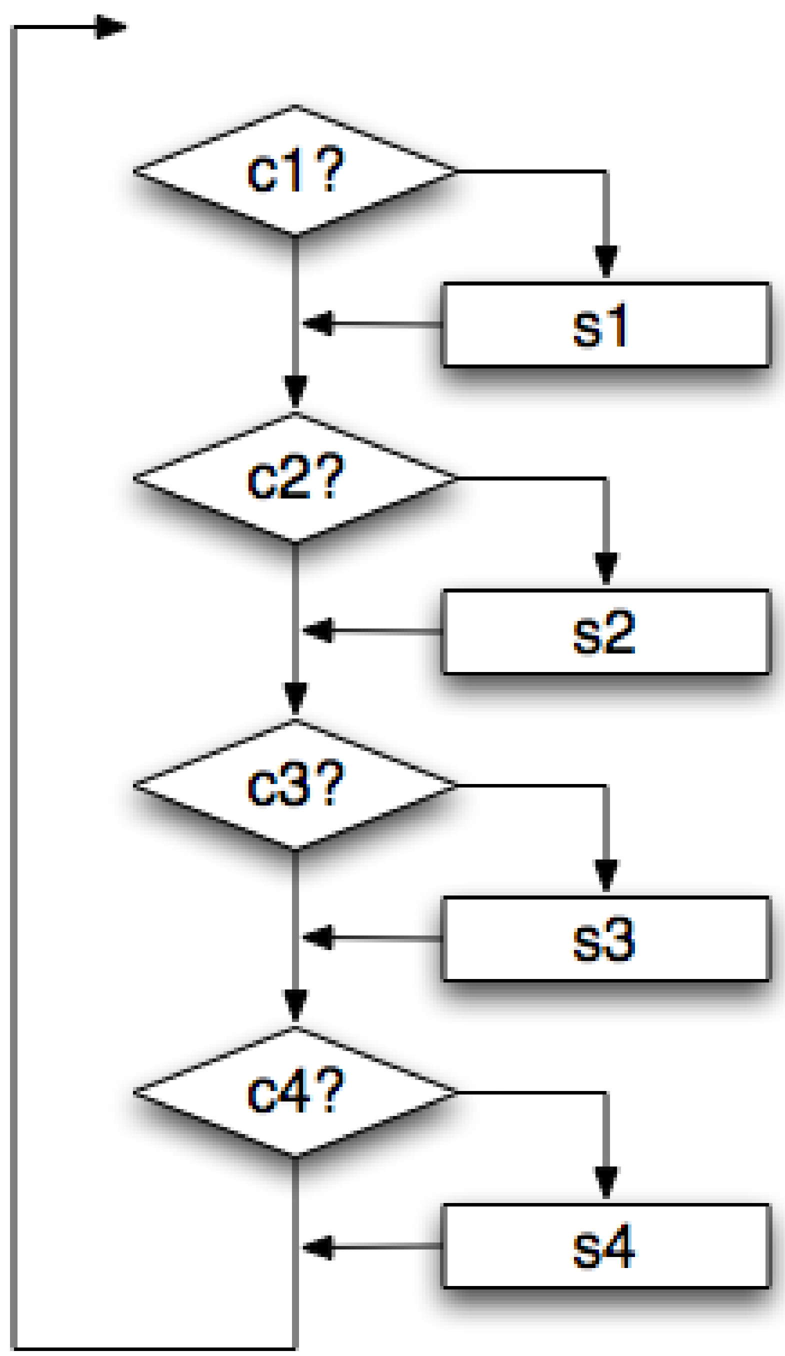


Original CFG



Paths to be covered

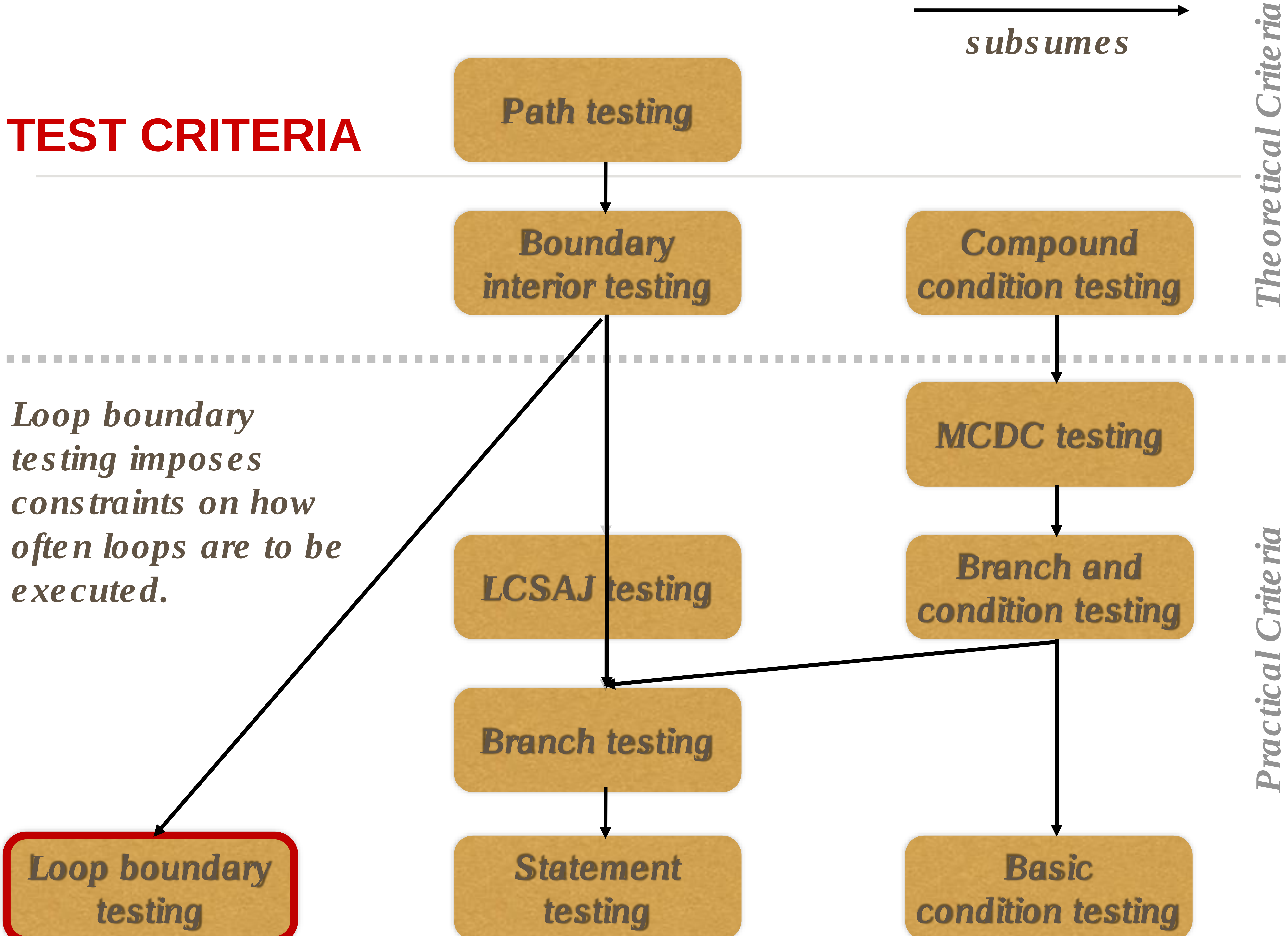
BOUNDARY INTERIOR TESTING: ISSUES



- The number of paths may still grow exponentially
  - In this example, there are 24 = 16 paths
- Forcing paths may be infeasible or even impossible if conditions are not independent
- Therefore, boundary interior testing belongs more to the “theoretical” criteria.

TEST CRITERIA

Loop boundary testing imposes constraints on how often loops are to be executed.

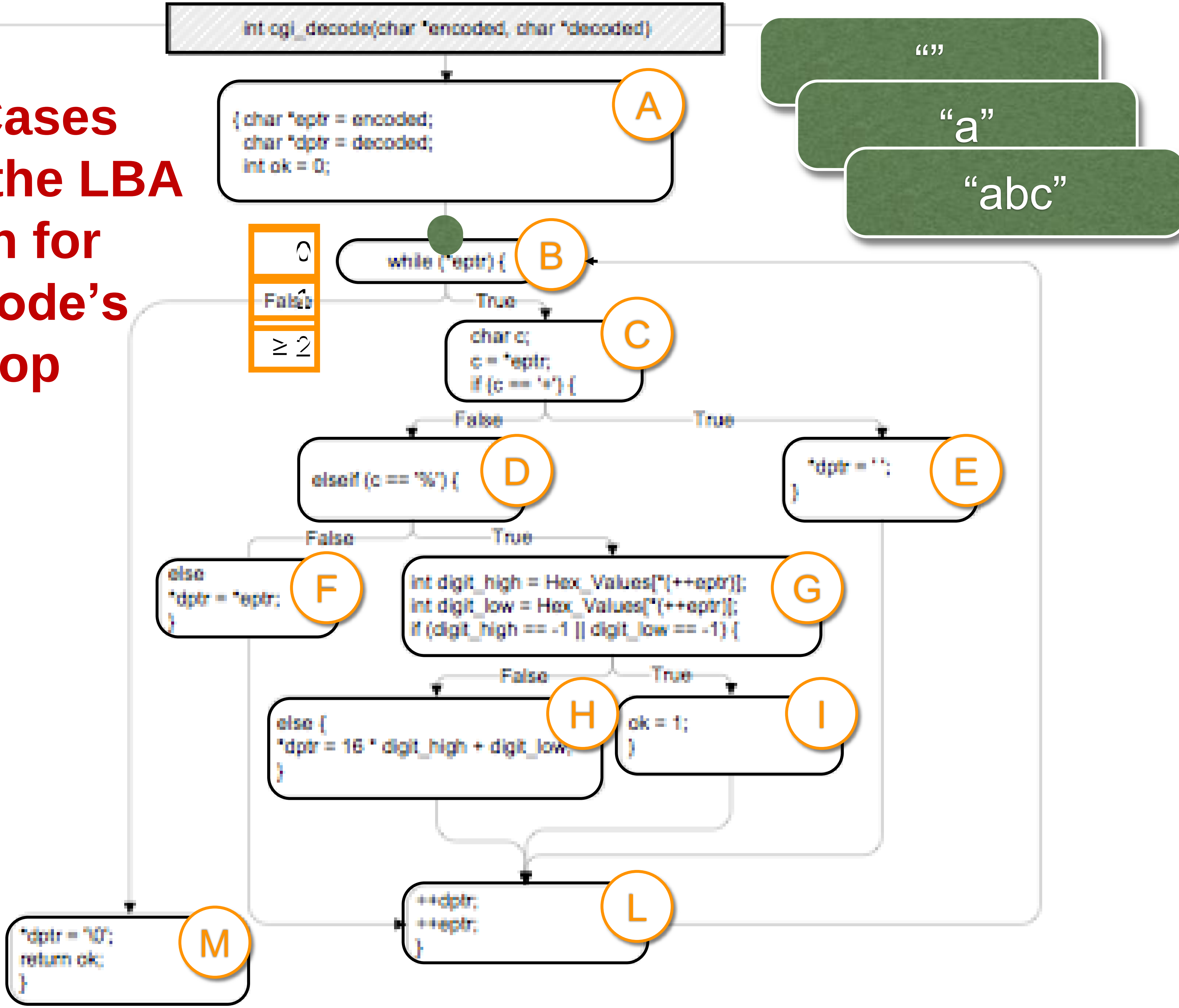


LOOP BOUNDARY ADEQUACY

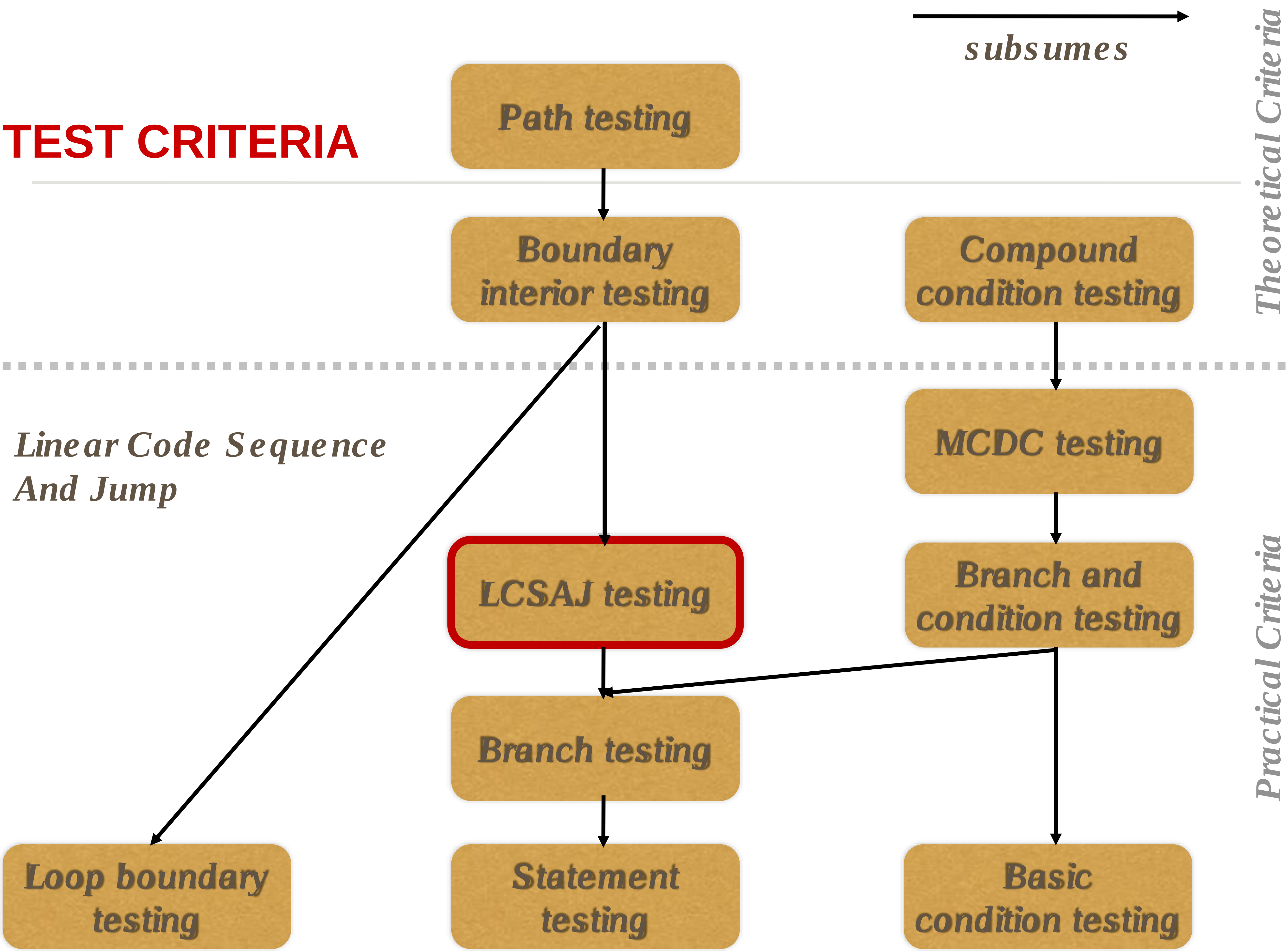
- A test suite satisfies this criterion if for every loop L:
  - There is a test case that iterates L zero times
  - There is a test case that iterates L once
  - There is a test case that iterates L more than once

- Typically combined with other adequacy criteria such as MCDC

3 Test Cases Satisfy the LBA Criterion for cgi_decode's Main Loop



TEST CRITERIA



LCSAJ ADEQUACY
LINEAR CODE SEQUENCE AND JUMP

- Testing all paths up to a fixed length

- A LCSAJ is a sequential subpath in the CFG starting and ending in a branch

LCSAJ length	*corresponds to*
1	*statement coverage*
2	*branch coverage*
n	*coverage of n consecutive LCSAJs*
∞	*path coverage*

SATISFYING CRITERIA

Test criteria are not always satisfiable:

1. *Statements* may not be executed because of *defensive programming* or *code reuse*

2. *Conditions* may not be satisfiable because of *interdependent conditions*

3. *Paths* may not be executable because of *interdependent decisions*

SATISFYING CRITERIA

- Reaching specific code can be very hard!
  - Even the best-designed, best-maintained systems may contain unreachable code

- A large amount of unreachable code/paths/conditions is a serious *maintainability problem*

- Options:
  - Allow coverage less than 100%,
  - Require justification for exceptions

MORE TESTING CRITERIA EXAMPLES/OPTIONS

- Object-oriented testing
  - Every transition in the object’s FSM must be covered
  - Every method pair in the object’s FSM must be covered
- Interclass testing
  - Every interaction between two objects must be covered
- Data flow testing
  - Every definition-use pair of a variable must be covered

DATA FLOW TESTING:
COMPUTING THE WRONG VALUE LEADS TO FAILURE ONLY WHEN THAT VALUE IS LATER USED

- Typical data flow testing criterion

  the tests must exercise every pair (definition, uses) of a variable