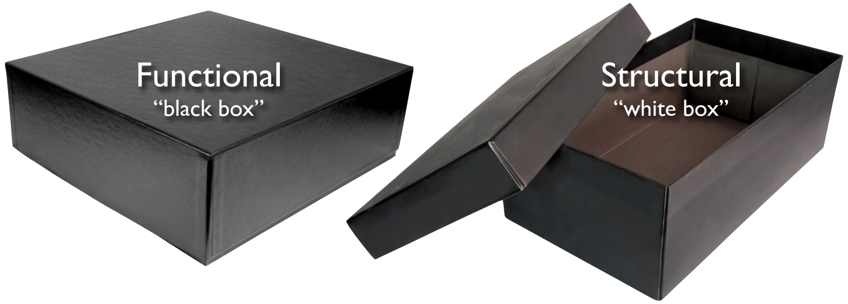
# Structural Testing

Software Engineering
Andreas Zeller • Saarland University

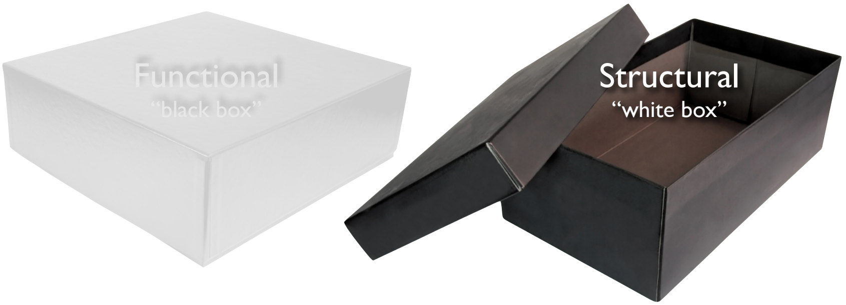
# Testing Tactics




- Tests based on *spec*
- Test covers as much *specified* behavior as possible

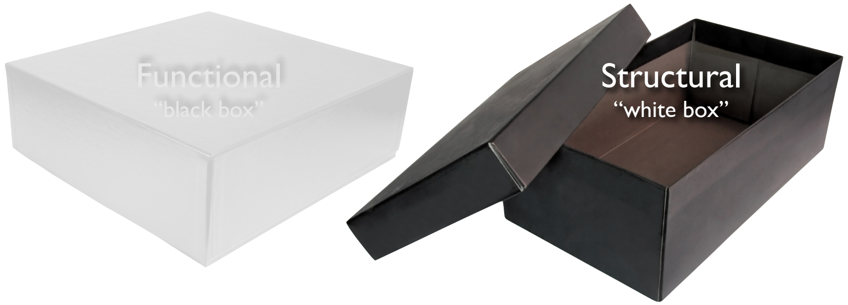
- Tests based on *code*
- Test covers as much *implemented* behavior as possible

# Why Structural?




- If a part of the program is never executed, a defect may loom in that part
  A “part” can be a statement, function, transition, condition…
- Attractive because automated

# Why Structural?




- Complements functional tests
  Run functional tests first, then measure what is missing
- Can cover low-level details missed in high-level specification

# A Challenge

```
class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Result: values for x
    double root_one, root_two;
}
```

- Which values for *a, b, c* should we test?
  assuming a, b, c, were 32-bit integers, we'd have $(2^{32})^3 \approx 10^{28}$ legal inputs
  with 1.000.000.000.000 tests/s, we would still require 2.5 billion years

# The Code

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

```
// 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
    }
}
```

Test this case

and this

and this!

# The Test Cases

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

```
// 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
    }
}
```

(a, b, c) = (3, 4, 1)

(a, b, c) = (0, 0, 1)

(a, b, c) = (3, 2, 1)

# A Defect

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

```
// 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) {
        x = (-b) / (2 * a);
    }

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

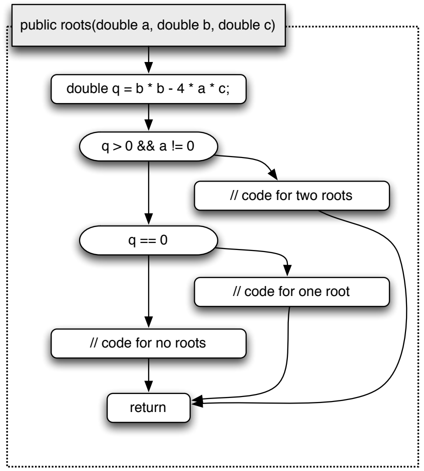
code must handle a = 0

(a, b, c) = (0, 0, 1)

# Expressing Structure

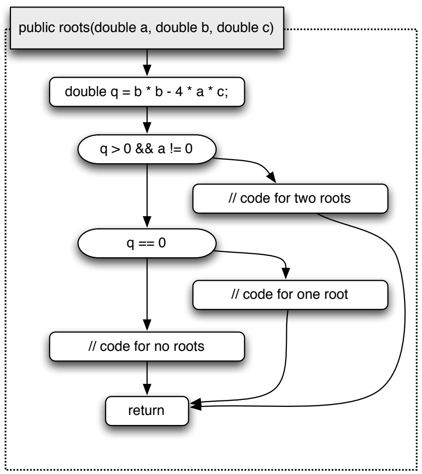
```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

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

# Control Flow Graph



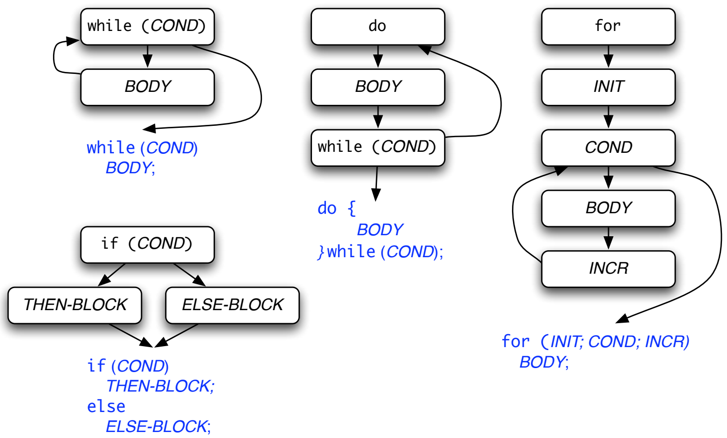

- A *control flow graph* expresses paths of program execution
- *Nodes* are *basic blocks* – sequences of statements with one entry and one exit point
- *Edges* represent *control flow* – the possibility that the program execution proceeds from the end of one basic block to the beginning of another

# Structural Testing




- The CFG can serve as an *adequacy criterion* for test cases
- The more parts are covered (executed), the higher the chance of a test to uncover a defect
- "parts" can be: nodes, edges, paths, conditions…

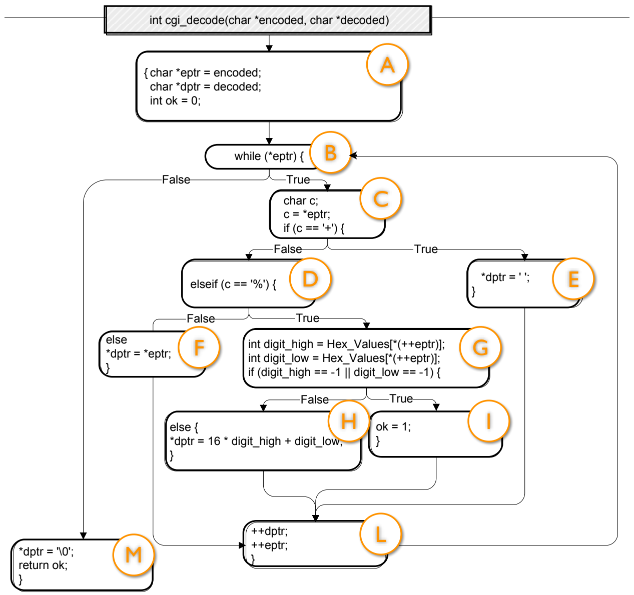
# Control Flow Patterns

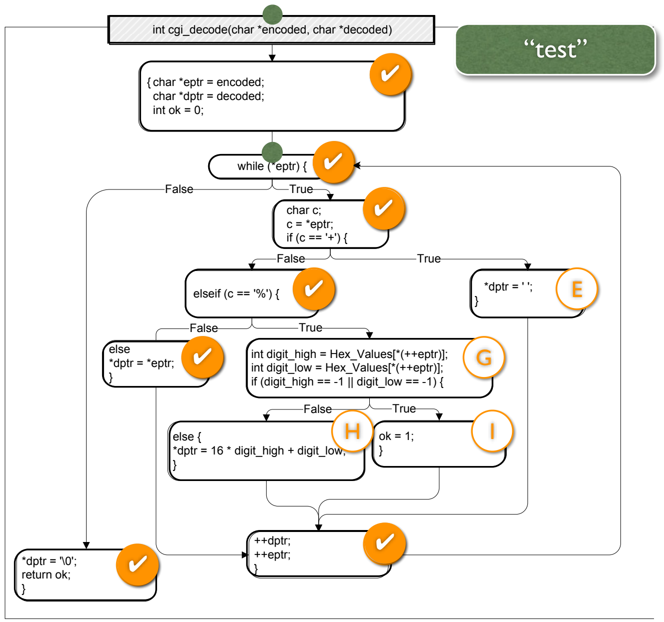


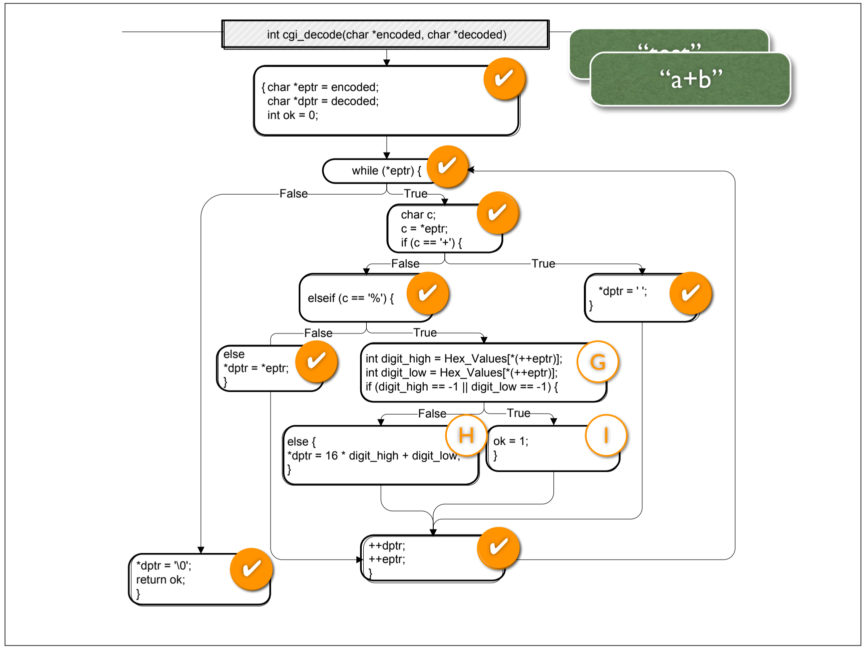
# cgi_decode

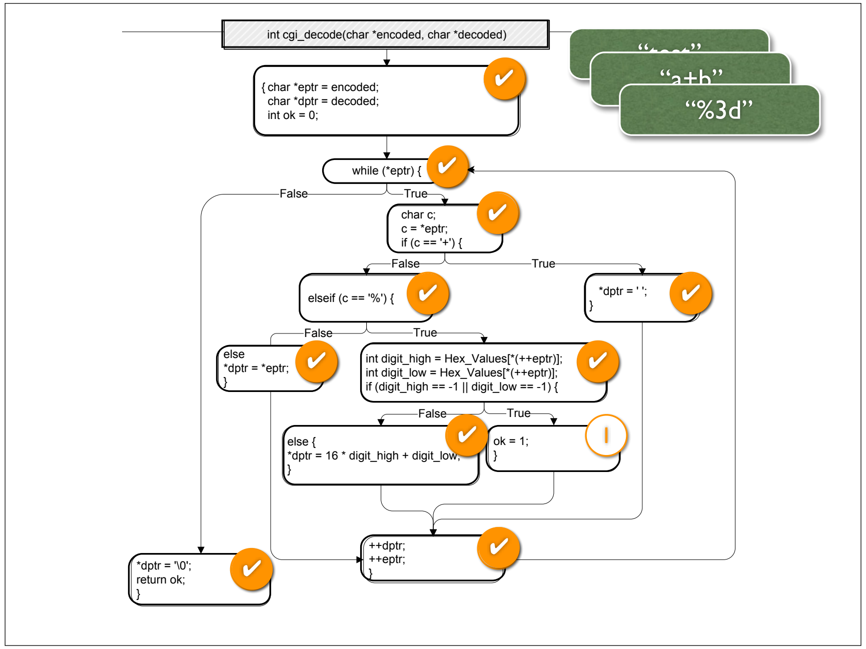
```
/**
  * @title cgi_decode
  * @desc
  * Translate a string from the CGI encoding to plain ascii text
  * '+' becomes space, %xx becomes byte with hex value xx,
  * other alphanumeric characters map to themselves
  *
  * returns 0 for success, positive for erroneous input
  * 1 = bad hexadecimal digit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

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;  A
    int ok = 0;
```

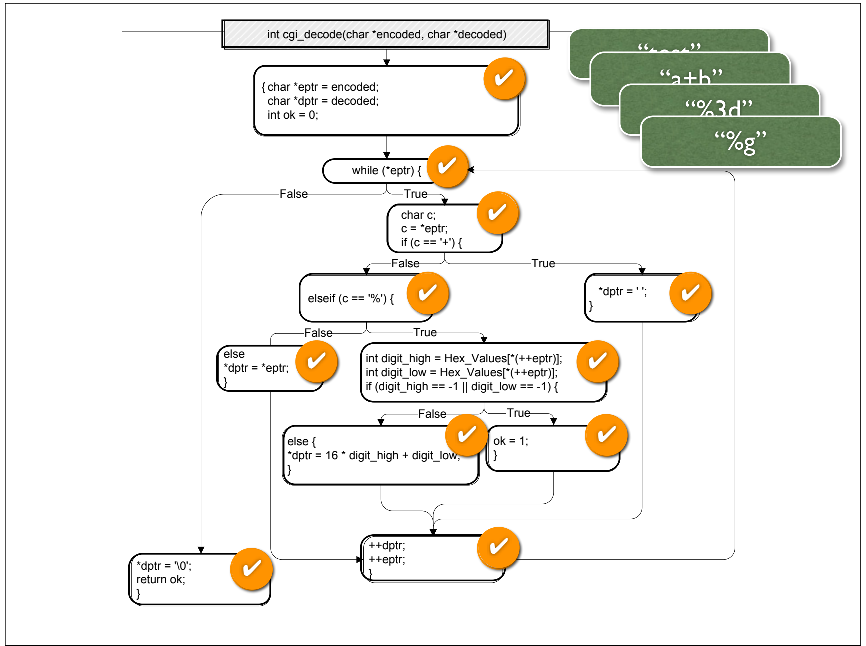
```
    while (*eptr)  /* loop to end of string ('\0' character) */  B
    {
        char c;  C
        c = *eptr;
        if (c == '+') {  /* '+' maps to blank */
            *dptr = ' ';  E
        } else if (c == '%') { /* '%xx' is hex for char xx */  D
            int digit_high = Hex_Values[*(++eptr)];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

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





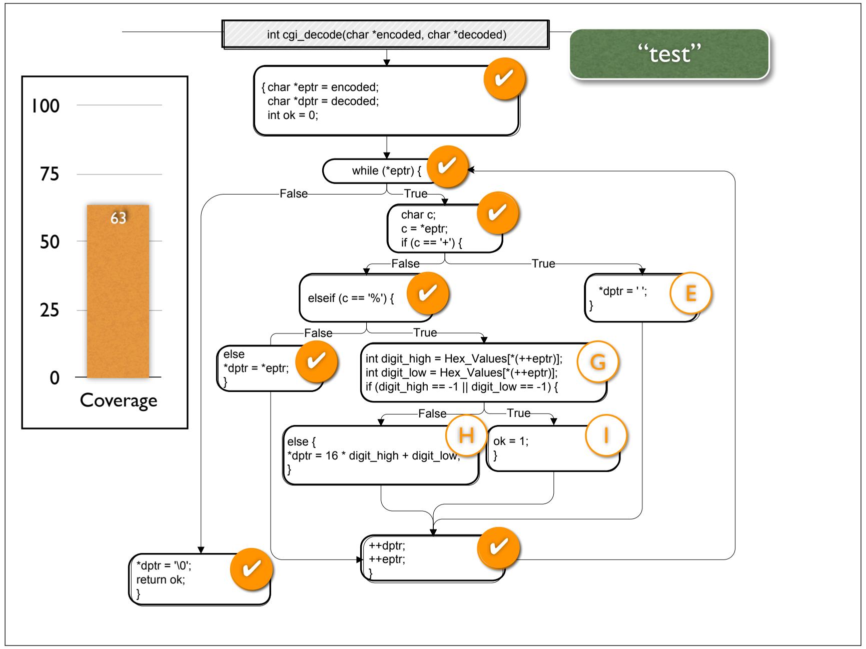



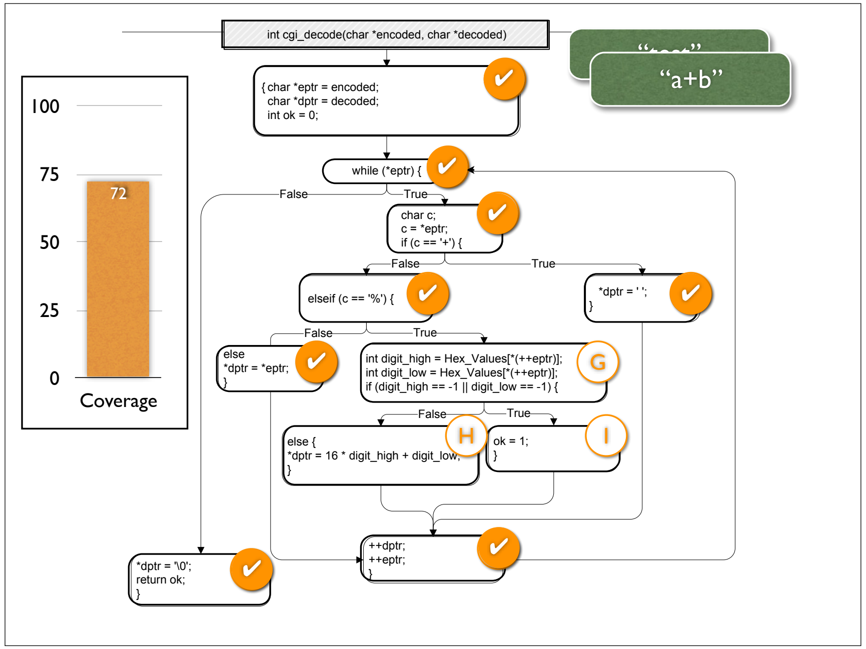

# Test Adequacy Criteria

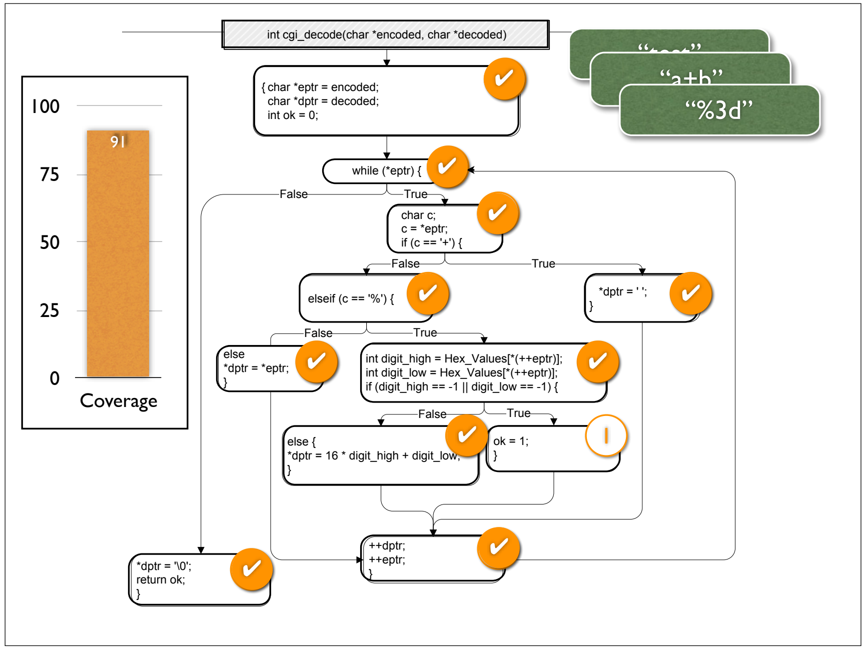
- How do we know a test suite is "good enough"?
- A *test adequacy criterion* is a predicate that is true or false for a pair ⟨*program, test suite*⟩
- Usually expressed in form of a rule – e.g., "all statements must be covered"

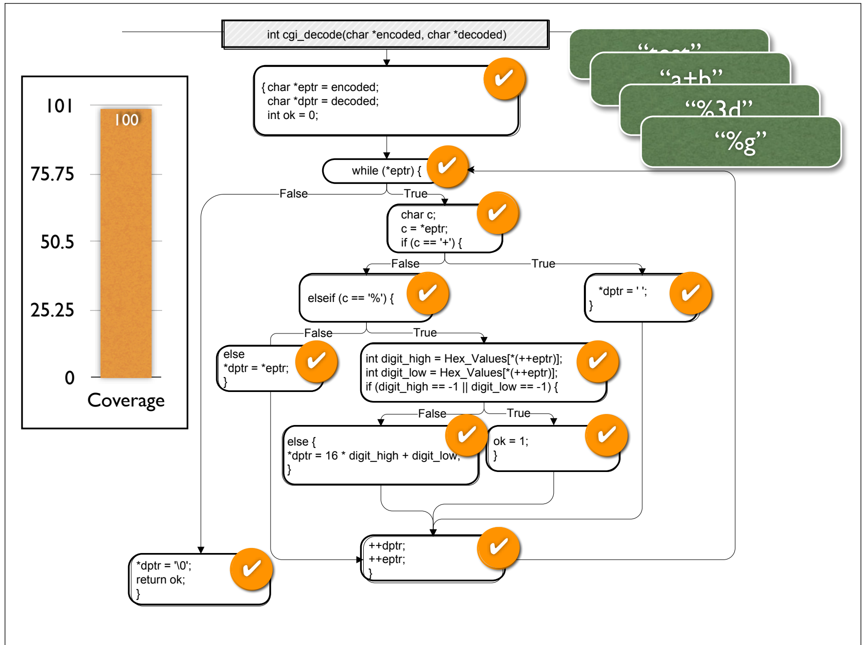
# Statement Testing

- 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: <u># executed statements</u>
  # statements





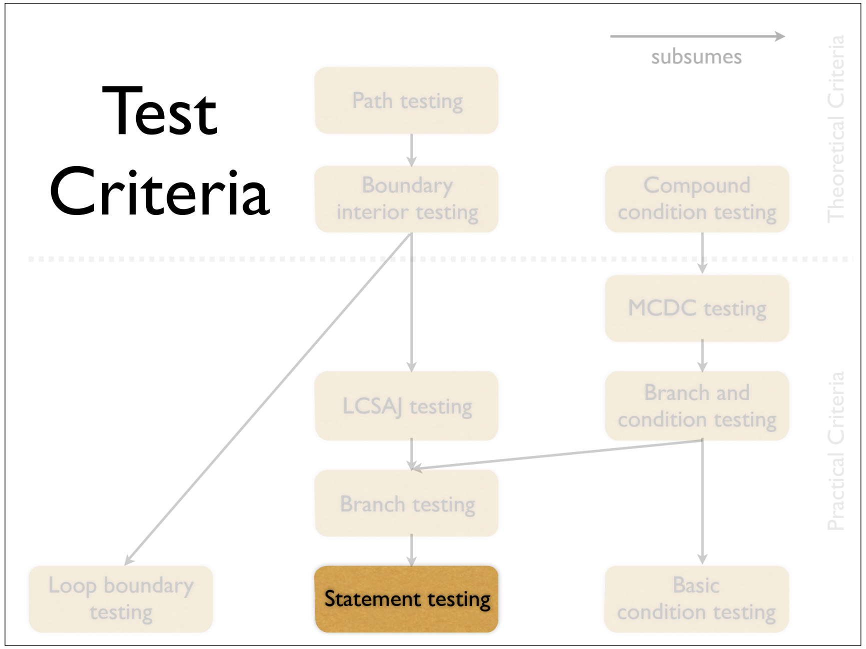

# Computing Coverage

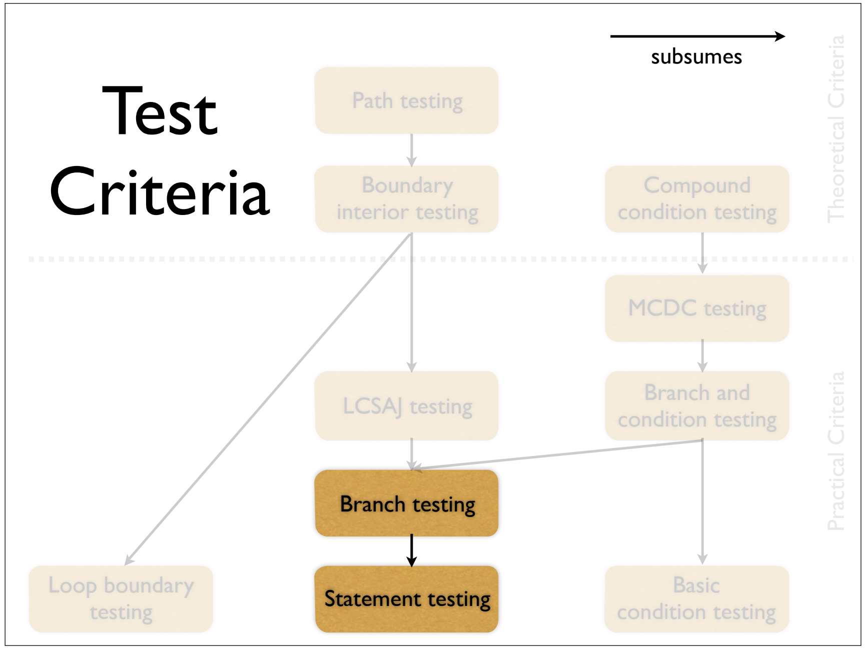
- Coverage is computed automatically while the program executes
- Requires *instrumentation* at compile time
  With GCC, for instance, use options -ftest-coverage -fprofile-arcs
- After execution, *coverage tool* assesses and summarizes results
  With GCC, use "gcov *source-file*" to obtain readable .gcov file

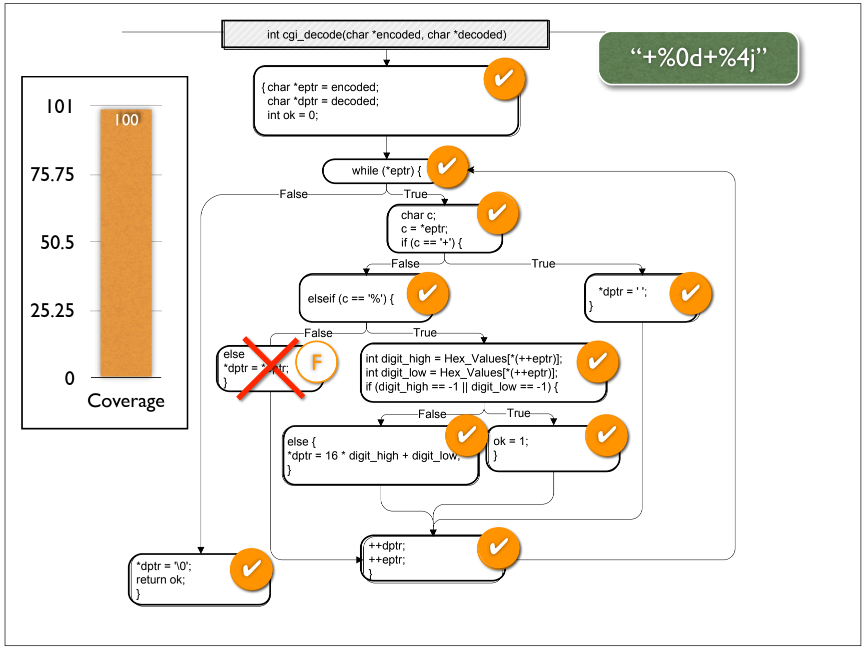
Pippin: cgi_encode — less — 80×24

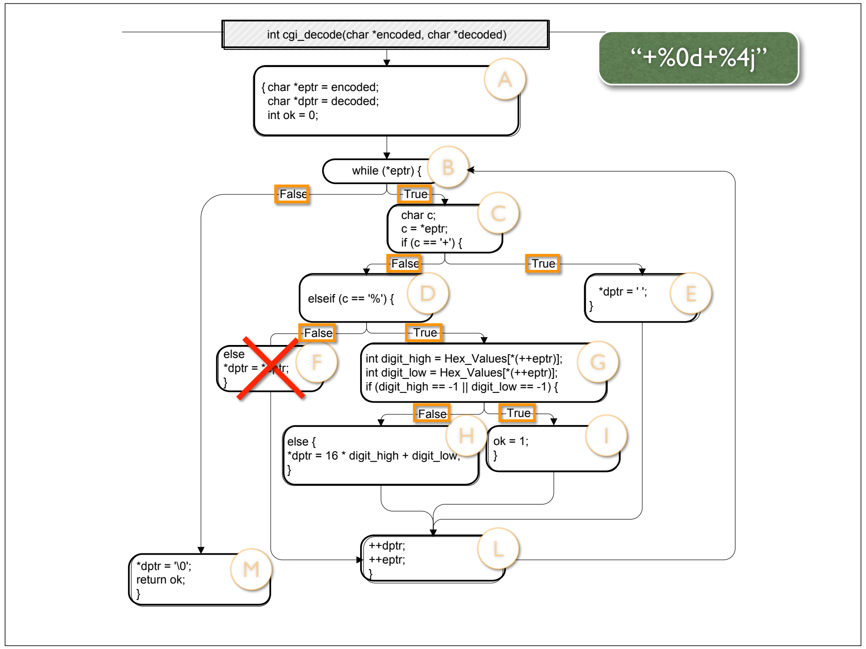
```
        4:   18:    int ok = 0;
        -:   19:
       38:   20:    while (*eptr)  /* loop to end of string ('\0' character) */
        -:   21:    {
        -:   22:        char c;
       30:   23:        c = *eptr;
       30:   24:        if (c == '+') {  /* '+' maps to blank */
        1:   25:            *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)
```

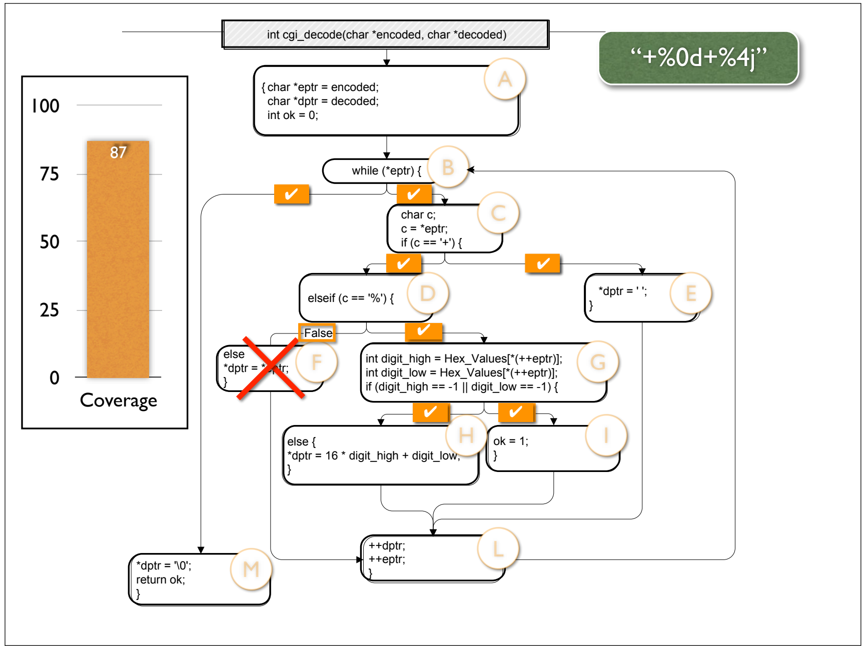
*Demo*

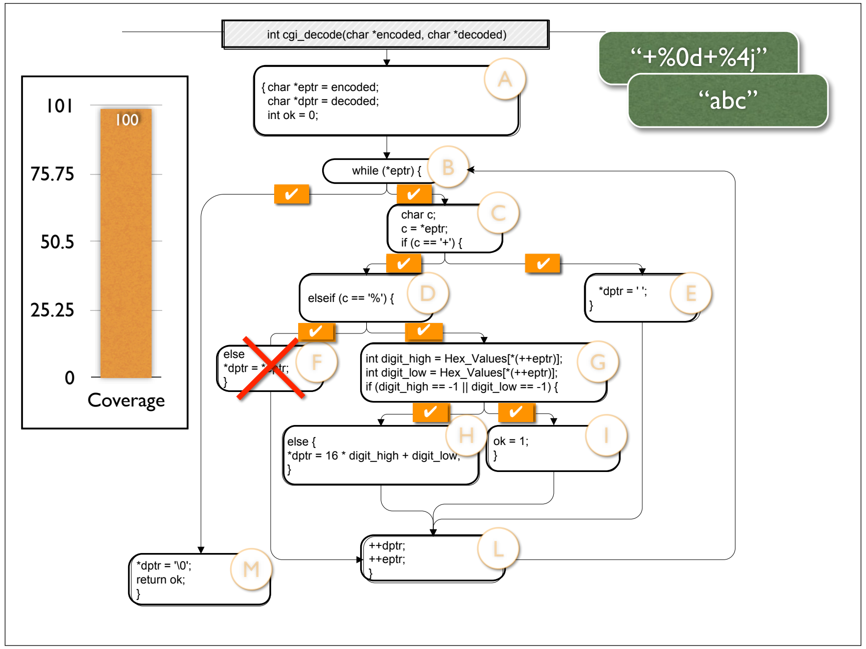

subsumes
Test Criteria
Path testing
Boundary interior testing
Compound condition testing
Theoretical Criteria
MCDC testing
LCSAJ testing
Branch and condition testing
Branch testing
Practical Criteria
Loop boundary testing
Statement testing
Basic condition testing


subsumes
Test Criteria
Path testing
Boundary interior testing
Compound condition testing
Theoretical Criteria
MCDC testing
LCSAJ testing
Branch and condition testing
Branch testing
Practical Criteria
Loop boundary testing
Statement testing
Basic condition testing


int cgi_decode(char *encoded, char *decoded)
"+%0d+%4j"
101
75.75
50.5
25.25
0
100
Coverage
{ 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;
}
F
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;
}


int cgi_decode(char *encoded, char *decoded)
"+%0d+%4j"
{ 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
else
*dptr = *eptr;
}
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




# Branch Testing

- Adequacy criterion: each branch in the CFG must be executed at least once
- Coverage: $\frac{\text{\# executed branches}}{\text{\# branches}}$
- Subsumes statement testing criterion
  because traversing all edges implies traversing all nodes
- Most widely used criterion in industry

# Condition Testing

- Consider the defect
  `(digit_high == 1 || digit_low == -1)`
  `// should be -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

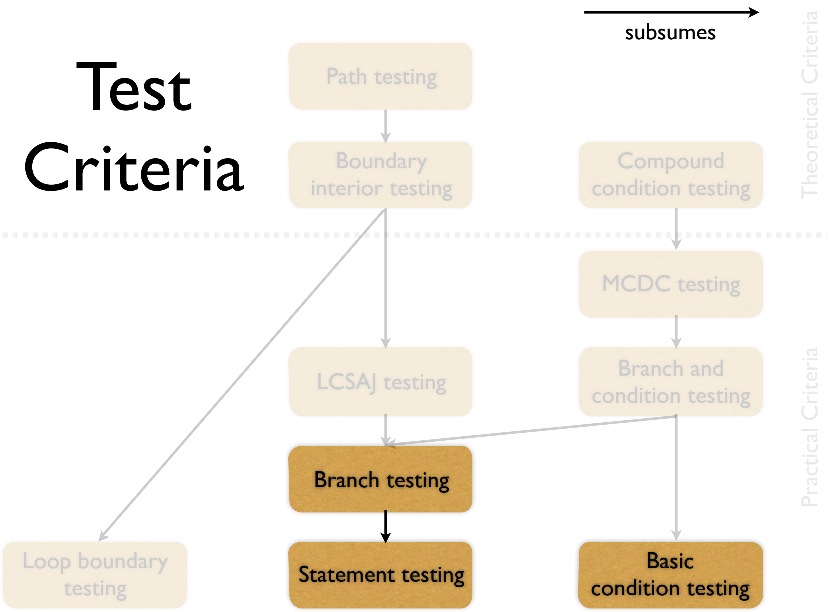
# Condition Testing

- Key idea: also cover *individual conditions* in compound boolean expression
  e.g., both parts of `digit_high == 1 || digit_low == -1`

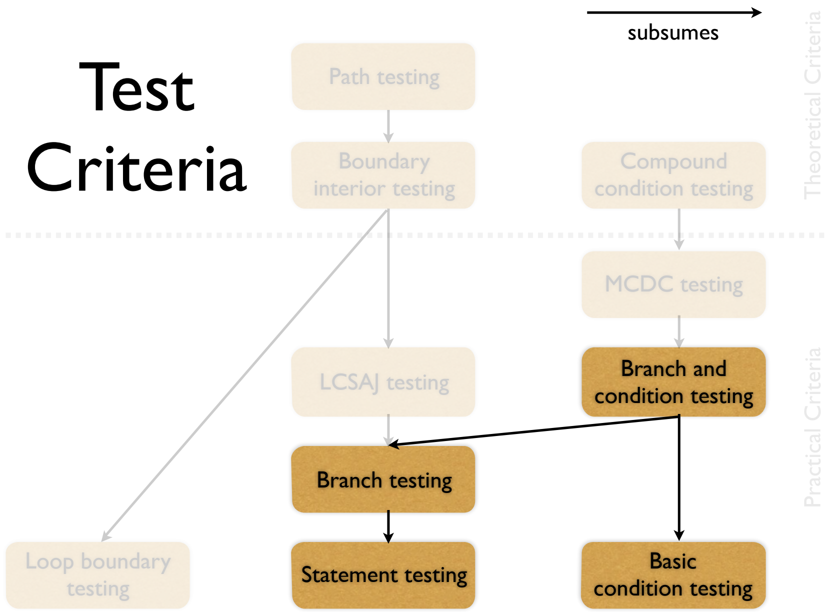
# Condition Testing

- Adequacy criterion: each basic condition must be evaluated at least once
- Coverage:
  $$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$
- Example: "test+%9k%k9"
  100% basic condition coverage
  but only 87% branch coverage

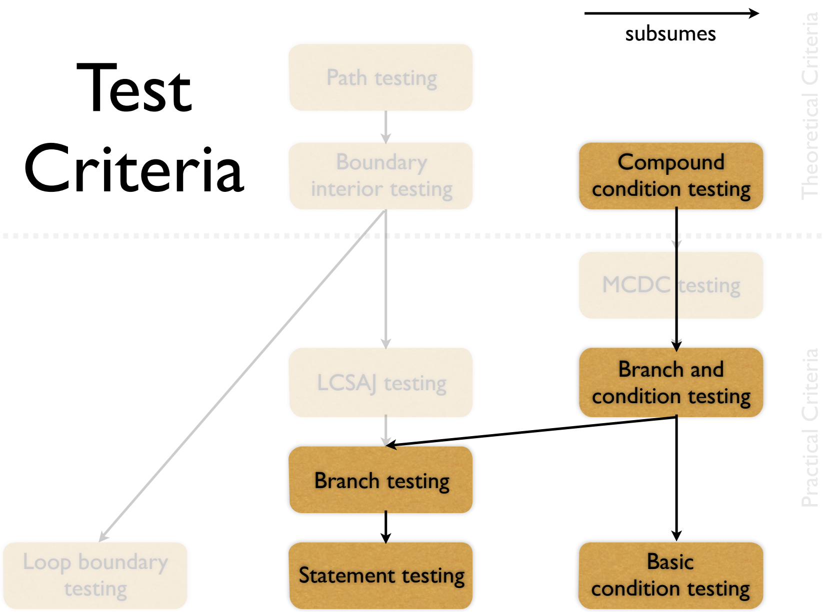
# Test Criteria




# Test Criteria



# Test Criteria

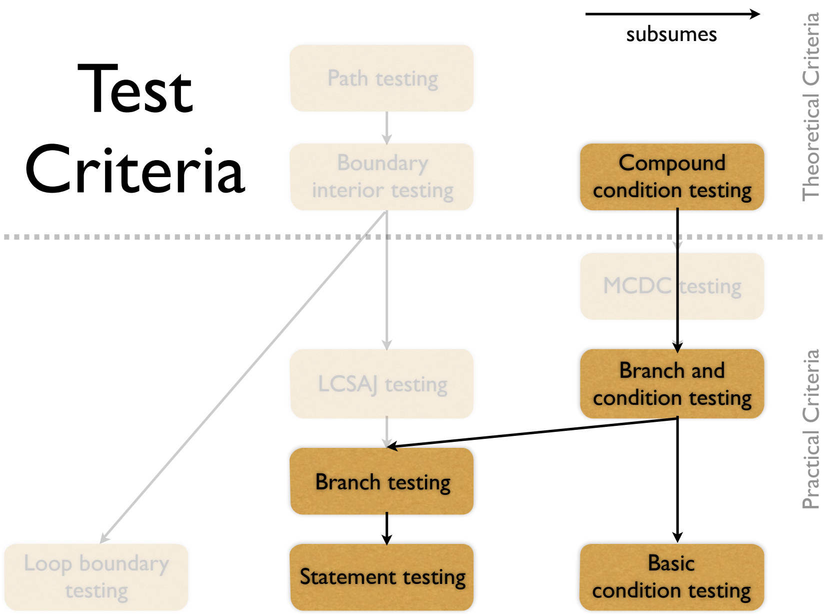



# Compound Conditions

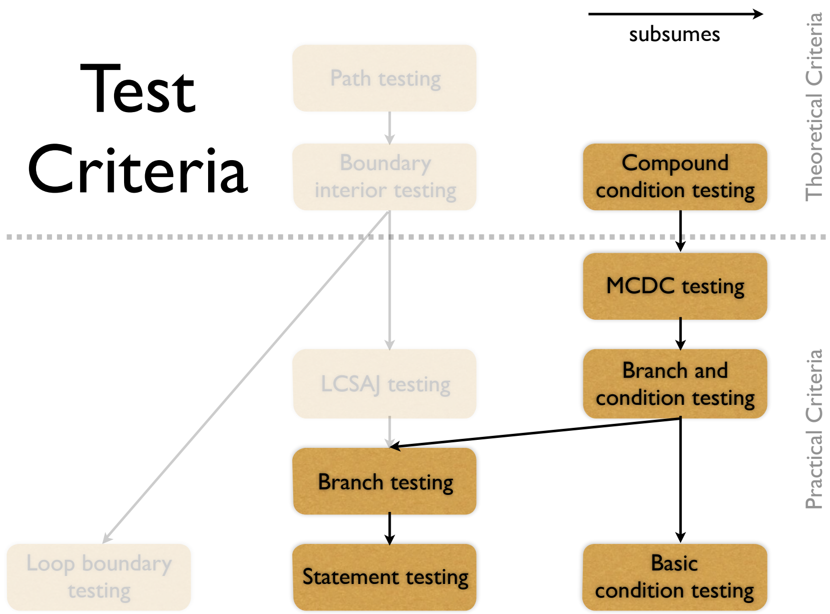
- Assume `(((a ∨ b) ∧ c) ∨ d) ∧ e)`
- We need 13 tests to cover all possible combinations
- In general case, we get a combinatorial explosion

| 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




# Test Criteria



# 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

# MCDC 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)

# MCDC Testing

Modified Condition Decision Coverage

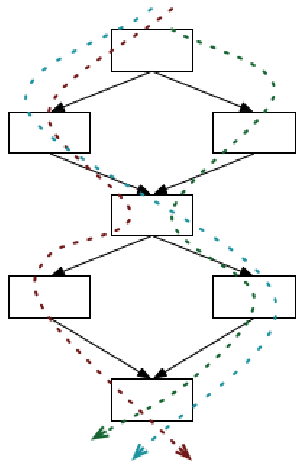
- Assume `(((a ∨ b) ∧ c) ∨ d) ∧ e)`
- We need six tests to cover MCDC combinations

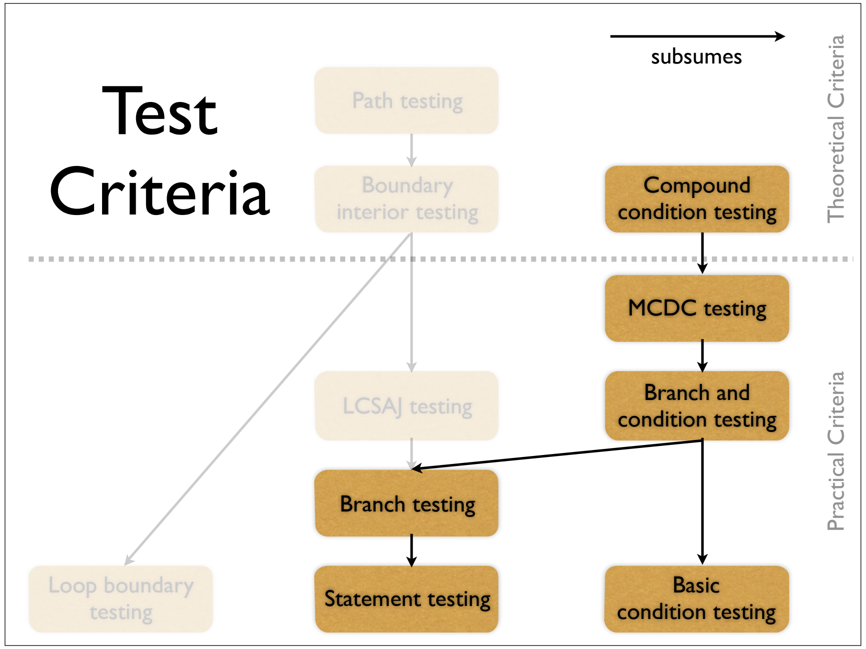
| | a | b | c | d | e | Decision |
|---|---|---|---|---|---|---|
| (1) | **True** | – | **True** | – | **True** | True |
| (2) | False | **True** | True | – | True | True |
| (3) | True | – | False | **True** | True | True |
| (6) | True | – | True | – | **False** | False |
| (11) | True | – | **False**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

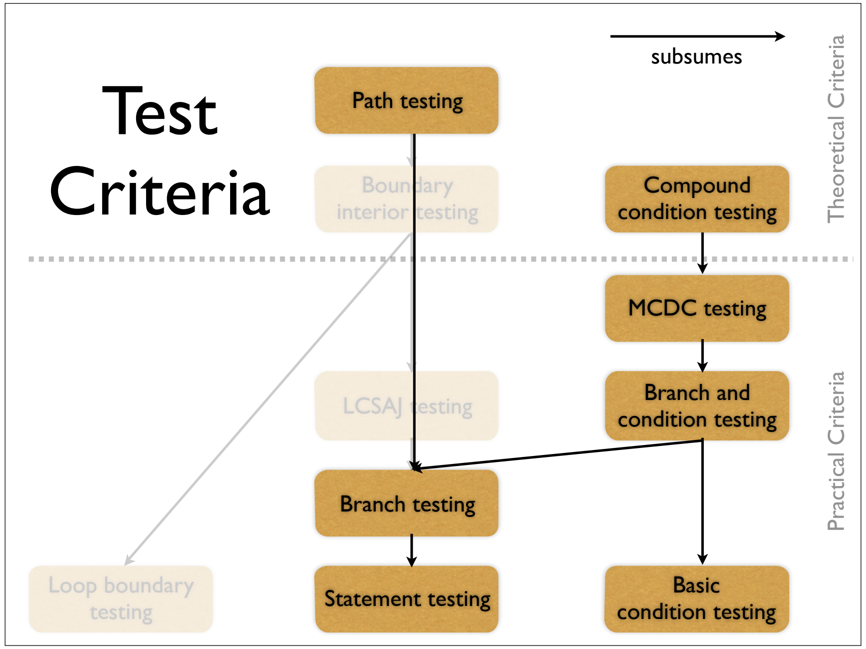
# Path Testing

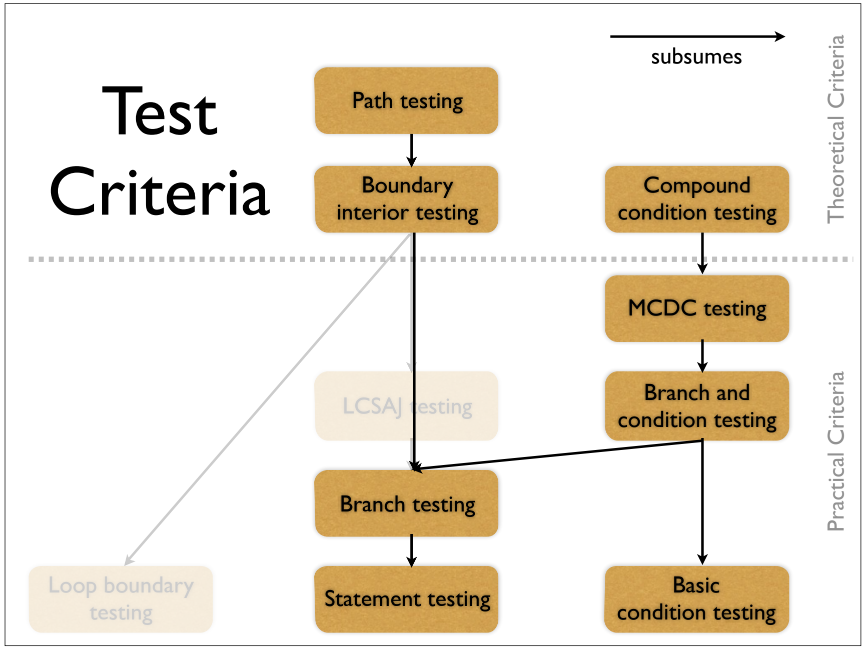
beyond individual branches

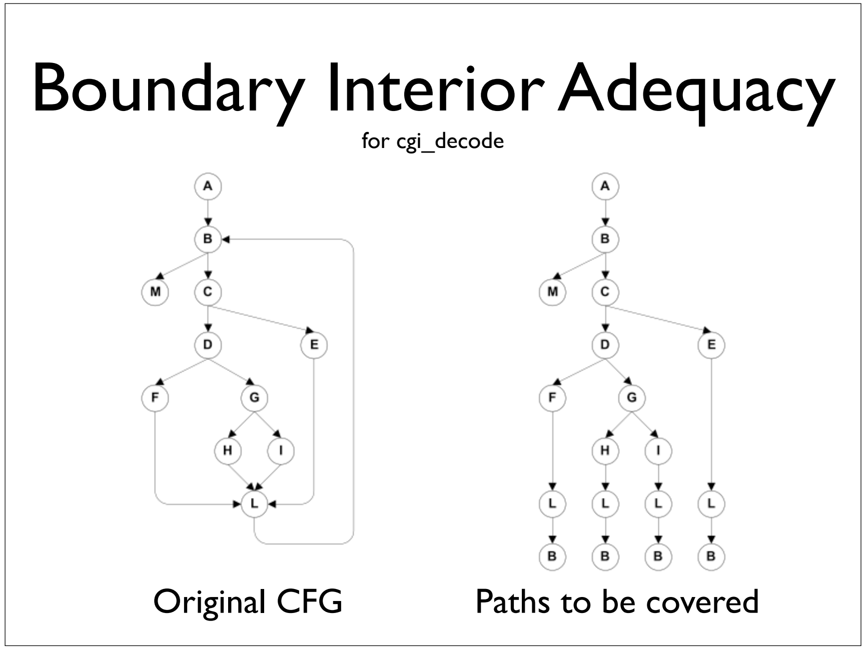
- Key idea: explore *sequences of branches* in control flow
- Many more paths than branches
  calls for compromises



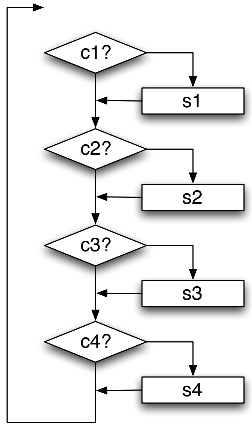

subsumes
Test Criteria
Path testing
Boundary interior testing
Compound condition testing
Theoretical Criteria
MCDC testing
LCSAJ testing
Branch and condition testing
Practical Criteria
Branch testing
Loop boundary testing
Statement testing
Basic condition testing


subsumes
Test Criteria
Path testing
Boundary interior testing
Compound condition testing
Theoretical Criteria
MCDC testing
LCSAJ testing
Branch and condition testing
Practical Criteria
Branch testing
Loop boundary testing
Statement testing
Basic condition testing


subsumes
Test Criteria
Path testing
Boundary interior testing
Compound condition testing
Theoretical Criteria
MCDC testing
LCSAJ testing
Branch and condition testing
Practical Criteria
Branch testing
Loop boundary testing
Statement testing
Basic condition testing

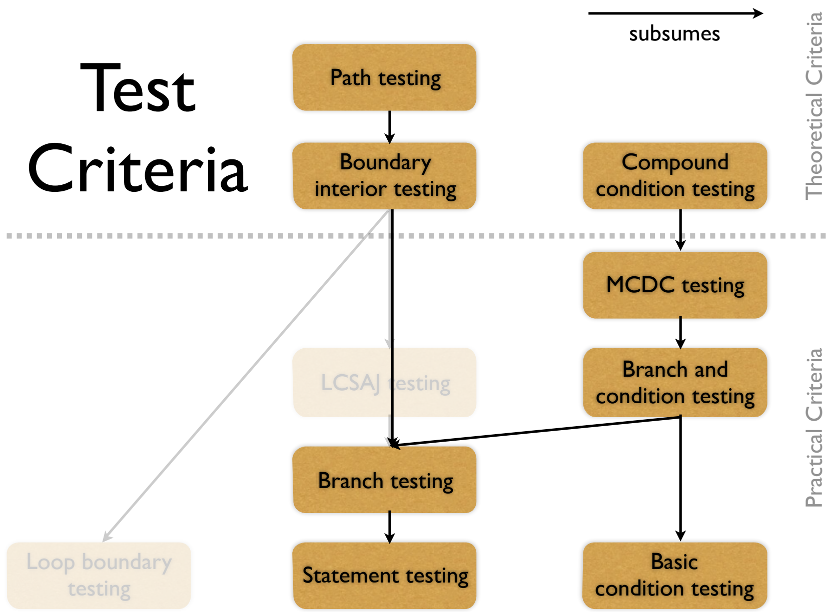

Boundary Interior Adequacy
for cgi_decode
Original CFG
Paths to be covered

# Issues

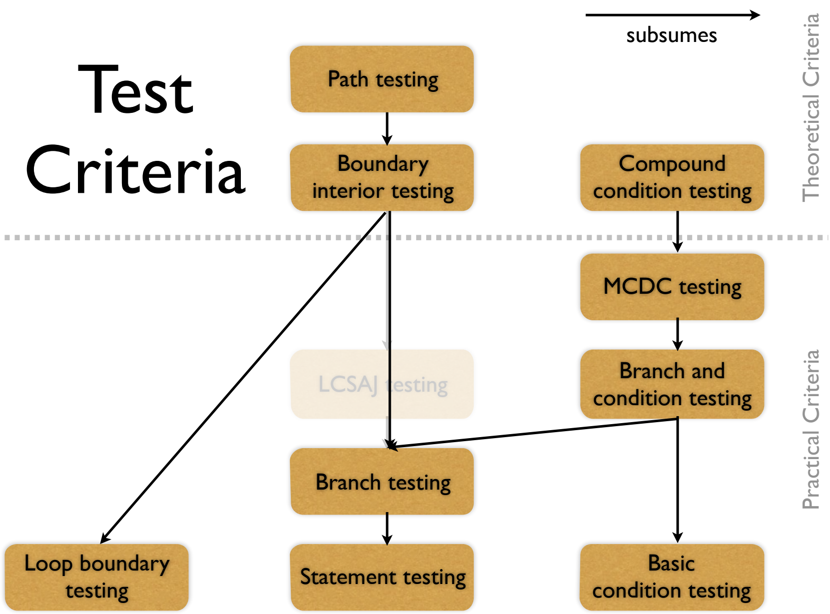



- The number of paths may still grow exponentially
  In this example, there are $2^4$ = 16 paths
- Forcing paths may be *infeasible*
  or even *impossible* if conditions are not independent

# Test Criteria




# Test Criteria

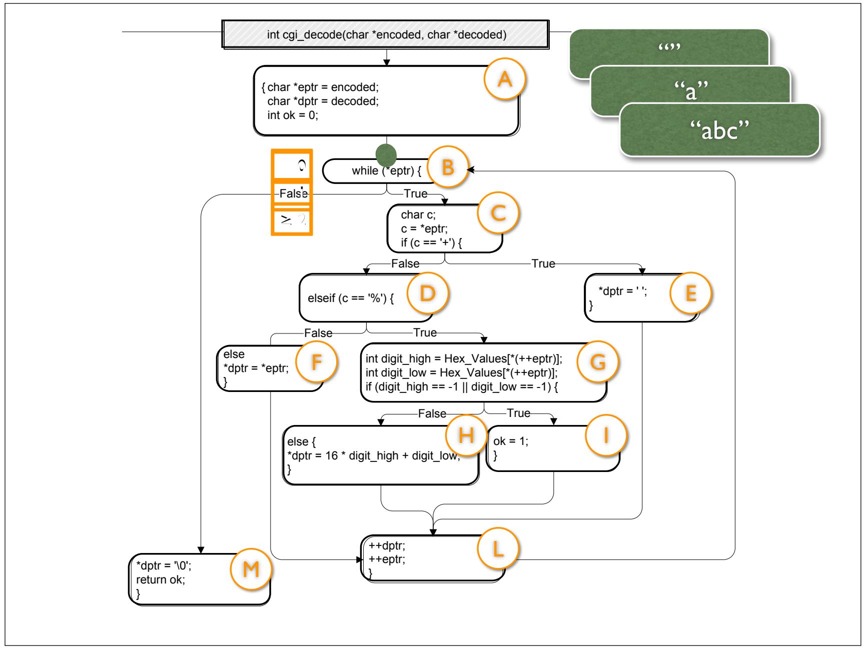


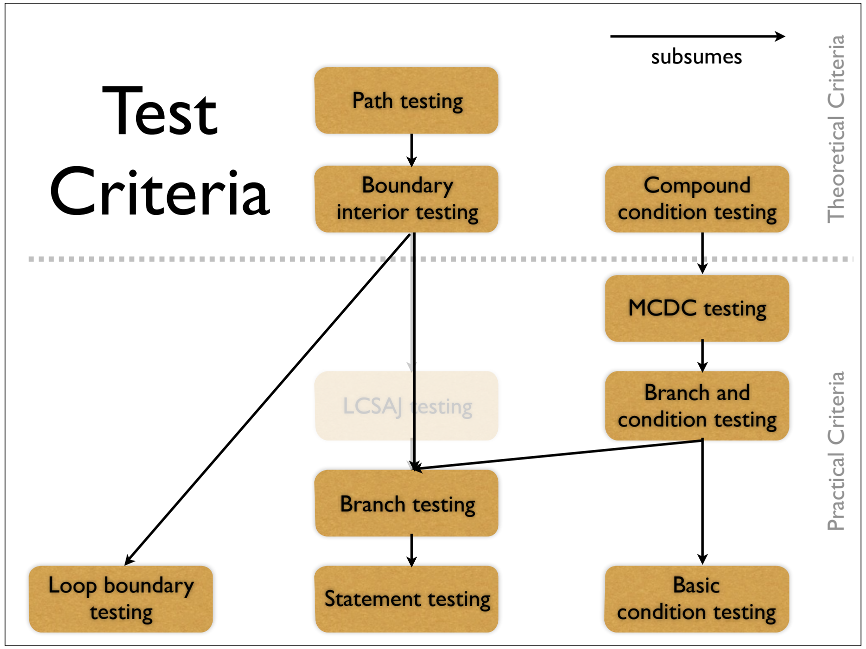

# Loop Boundary Adequacy

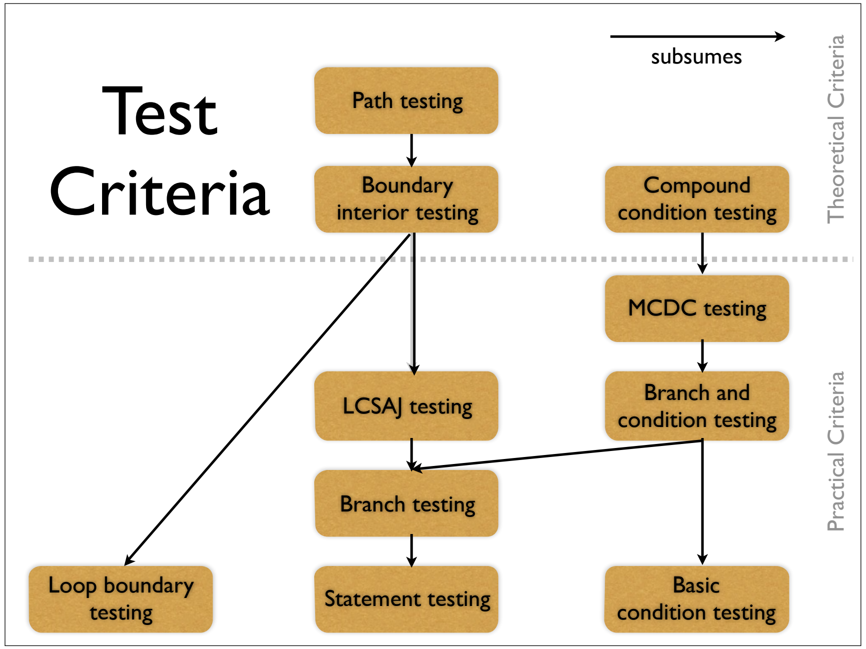
A test suite satisfies the loop boundary adequacy criterion if for every loop *L*:

- *There is a test case which iterates L zero times*
- *There is a test case which iterates L once*
- *There is a test case which iterates L more than once*

Typically combined with other adequacy criteria such as MCDC



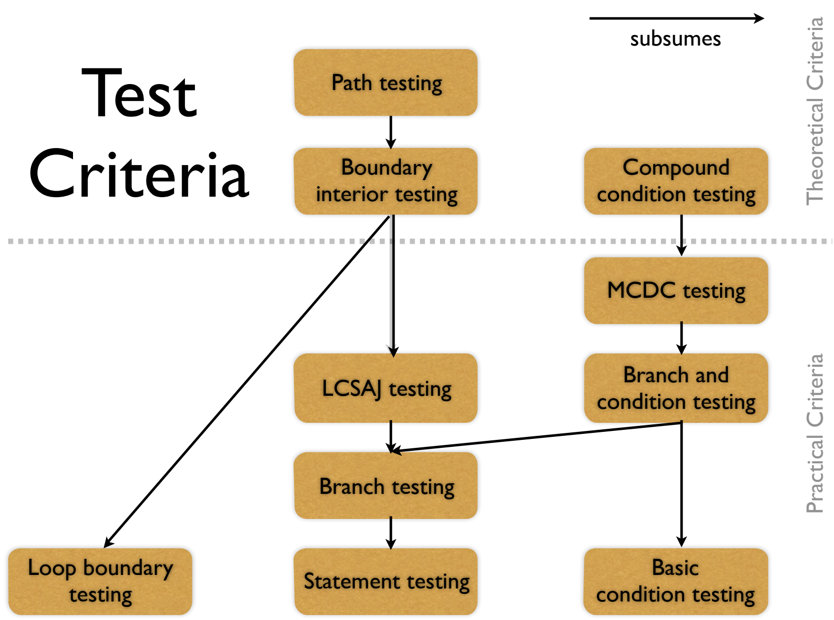

# LCSAJ Adequacy

Testing all paths up to a fixed length

- LCSAJ = Linear Code Sequence And Jump
- 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 |

# Test Criteria




# Weyuker's Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.

# Satisfying Criteria

Sometimes criteria may not be satisfiable:

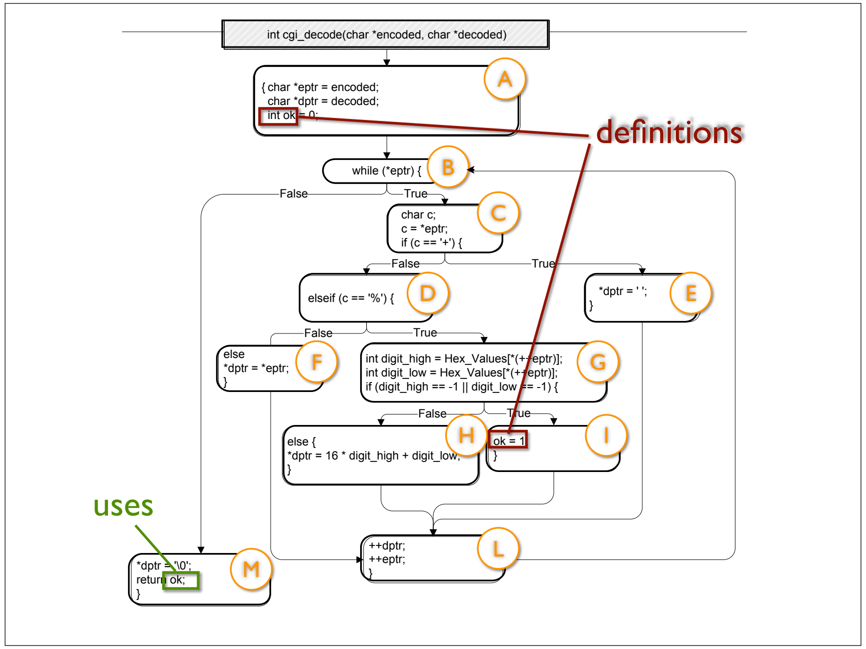
- *Statements* may not be executed because of *defensive programming* or *code reuse*
- *Conditions* may not be satisfiable because of *interdependent conditions*
- *Paths* may not be executable because of *interdependent decisions*

# 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*
- Solutions: allow coverage less than 100%, or require justification for exceptions

# More Testing Criteria

- Object-oriented testing
  e,g,"Every transition in the object's FSM must be covered" or "Every method pair in the object's FSM must be covered"
- Interclass testing
  e.g,"Every interaction between two objects must be covered"
- Data flow testing
  e.g.,"Every definition-use pair of a variable must be covered"




# Summary