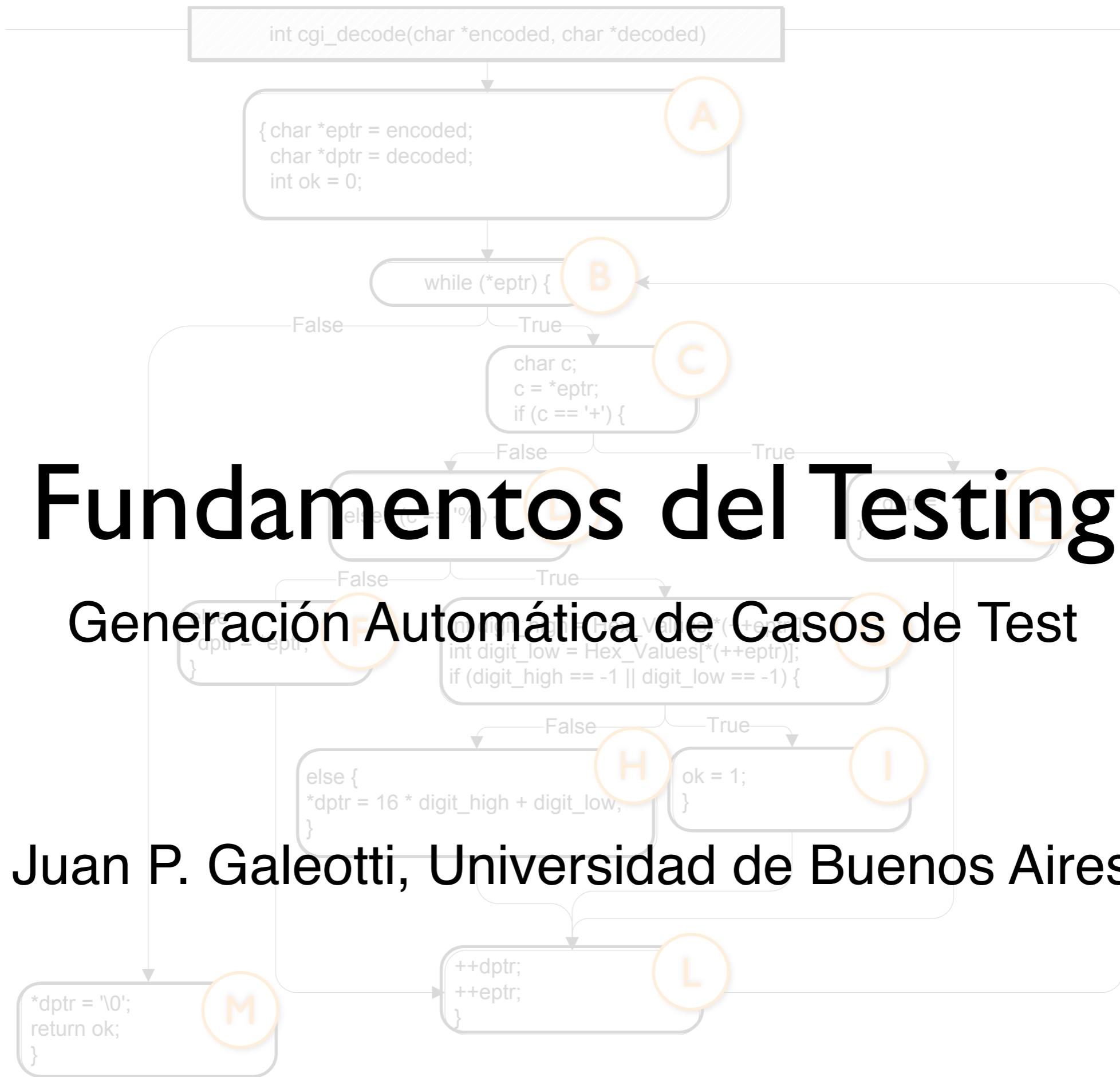


Fundamentos del Testing

Generación Automática de Casos de Test

Juan P. Galeotti, Universidad de Buenos Aires



Ariane V

- Último y más grande logro del programa espacial Europeo
- Vuelo inaugural: 4 de Junio de 1996



Le Bug in Python

```
def read_drift(vertical_bias, horizontal_bias,
               vertical_scale = 1.0, horizontal_scale = 1.0):
    """Compute drift as two 16-bit integers (horizontal_drift, vertical_drift)
       from vertical_bias/vertical_scale and horizontal_bias/horizontal_scale."""
    # Vertical drift
    vertical_drift_32 = convert_to_32_bit((1.0 / vertical_scale) * vertical_bias)

    if vertical_drift_32 > 32767:
        vertical_drift_16 = 32767 # 0x7fff
    elif vertical_drift_32 < -32768:
        vertical_drift_16 = -32768 # 0x8000
    else:
        vertical_drift_16 = convert_to_16_bit(vertical_drift_32)

    # Horizontal drift
    horizontal_drift_16 = \
        convert_to_16_bit((1.0 / horizontal_scale) * horizontal_bias)

    return (vertical_drift_16, horizontal_drift_16)
```

Testing read_drift()

```
#!/usr/bin/env python

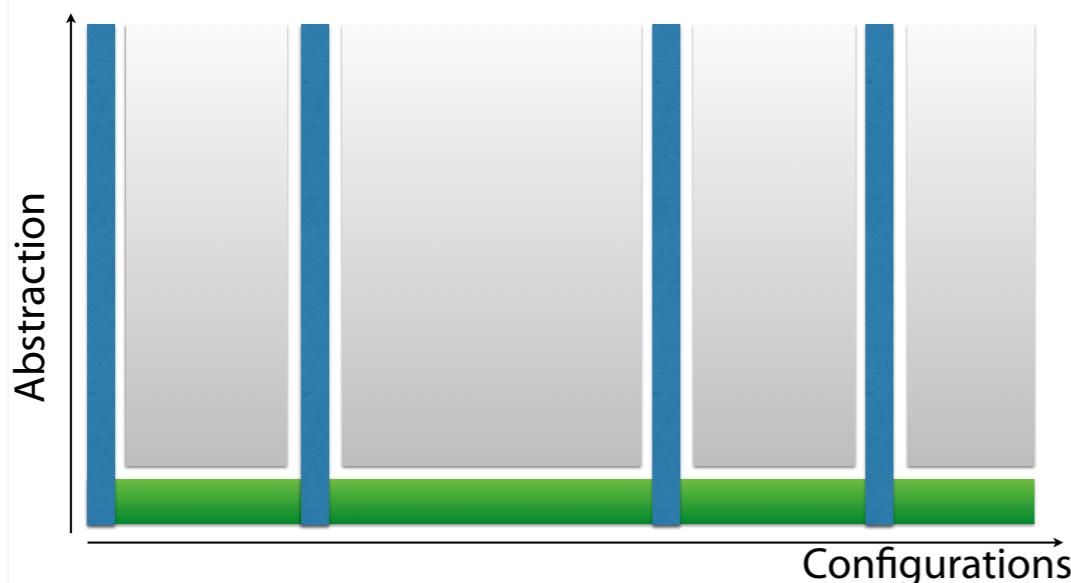
import Ariane

def test_neutral_scale():
    # Fixture
    vertical_bias = 100
    horizontal_bias = 200
    scale = 1.0

    # Operation
    (vertical_drift, horizontal_drift) = \
        Ariane.read_drift(vertical_bias,
                          horizontal_bias, scale, scale)

    # Check
    assert vertical_drift == vertical_bias
    assert horizontal_drift == horizontal_bias
```

Best of Two Worlds



Searching for Best Inputs

Random Testing

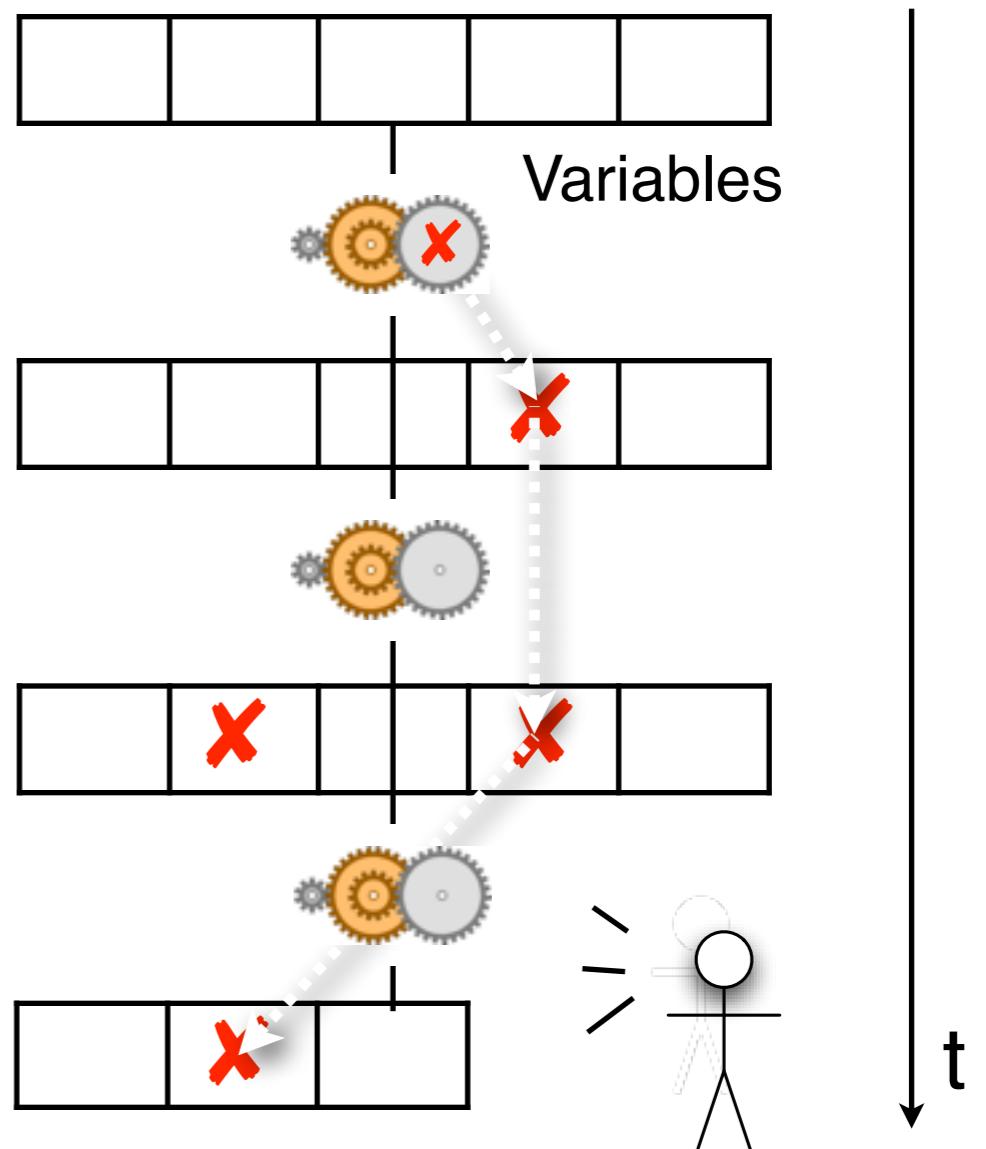
Exhaustive Testing

Symbolic Testing

Search-Based Testing

Del Defecto a la Falla

1. El programador crea un **defecto** en el código.
2. Cuando es ejecutado, el defecto crea una *infección*.
3. La infección se *propaga*.
4. La infección causa una *falla*.



Todos los Errores

- **Error**: Una desviación no buscada ni intencional de lo que es correcto, esperado o verdadero.
- **Defecto**: Un *error* en el código del programa, específicamente uno que puede crear un *infección* (y conducir a una *falla*)
- **Infección**: Un *error* en el estado del programa, específicamente uno que puede llevar a una *falla*
- **Falla**: Un error externamente visible en el comportamiento del programa.

Le Bug

```
procedure LIRE_DERIVE is
```

```
    -- lines and parameters omitted
```

– Compute vertical derivation

```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV) * G_M_INFO_DERIVE(T_ALG.E_BV));
```

– Convert into 16 bit

```
if L_M_BV_32 > 32767 then
```

```
    P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
```

```
elsif L_M_BV_32 < -32768 then
```

```
    P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
```

```
else
```

```
    P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M_BV_32));
```

```
end if;
```

– Same for horizontal

```
P_M_DERIVE(T_ALG.E_BH) :=
```

```
    UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH)));
```

```
end LIRE_DERIVE;
```

— El defecto

L'Effet

- *Hardware exception* halted Inertial Reference System as well as the backup system (same code), emitting crash data. — **La infección**
- Crash data would be misinterpreted by the autopilot as *actual flight data*
- Autopilot would correct assumed deviation by ordering *full nozzle deflections* of solid boosters and main engine. — **La falla**



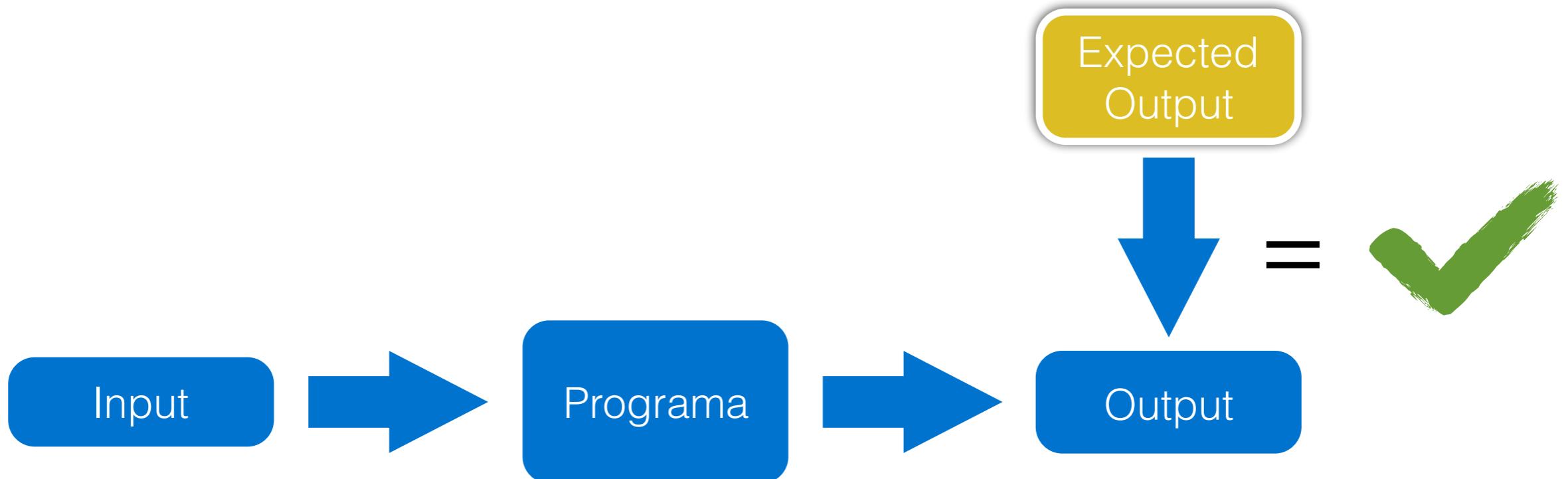
Testing

- **Problema:** Una propiedad o comportamiento del programa questionable.
- **Debugging:** La actividad de asociar un problema dado a un defecto que lo causa.
- **Testing:** La ejecución de un programa con la intención de producir algún problema, especialmente una falla

Objetivos del Testing

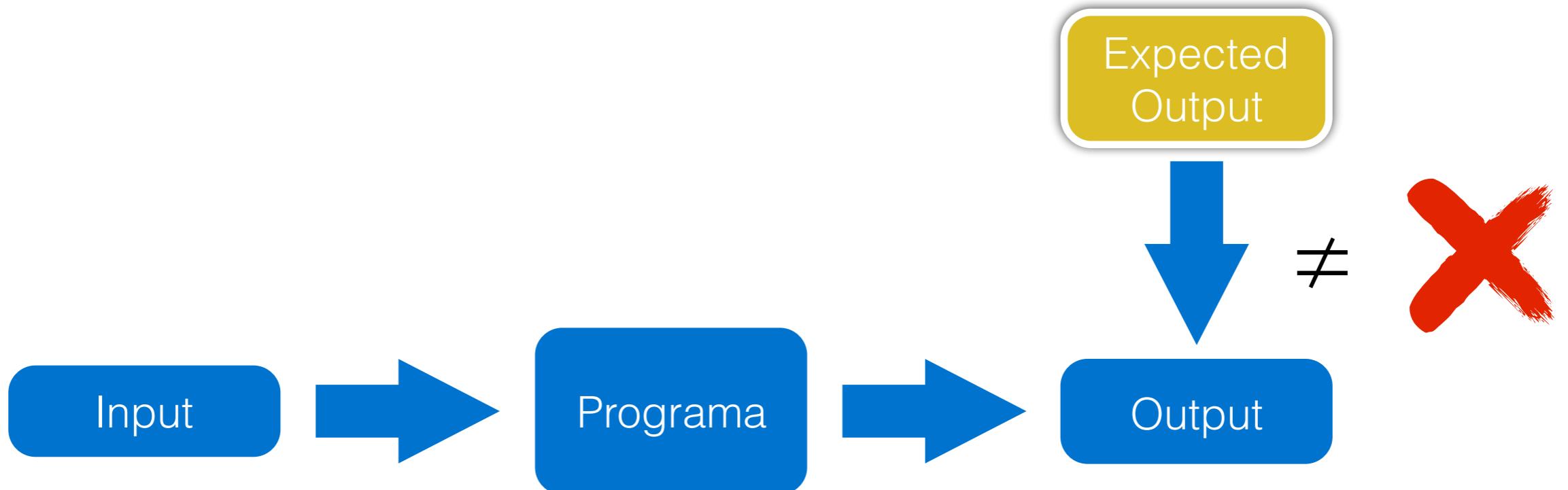
- Demonstración
 - mostrar que satisface la spec
- Destrucción
 - tratar de hacerlo fallar
- Evaluación
 - descubrir si y cómo falla
- Prevención
 - evitar futuras fallas

UNIT TESTING



1. Ejecutar el programa
Usando datos de test
2. Chequear el output del programa
Usando el oráculo de test

UNIT TESTING

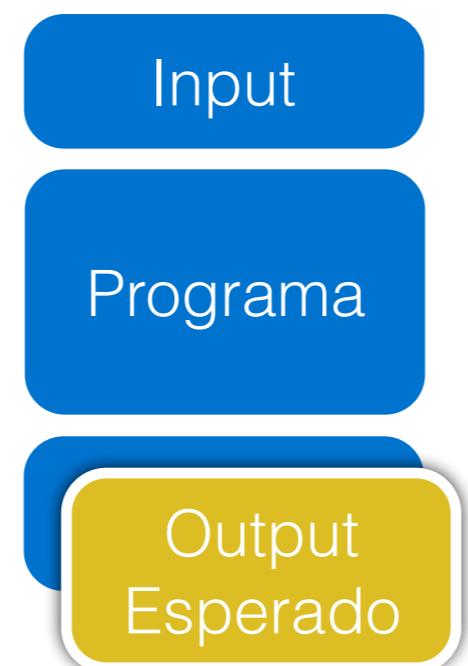


1. Ejecutar el programa
Usando datos de test
2. Chequear el output del programa
Usando el oráculo de test

JUNIT TESTING

@Test

```
public void test()  
{
```



```
}
```

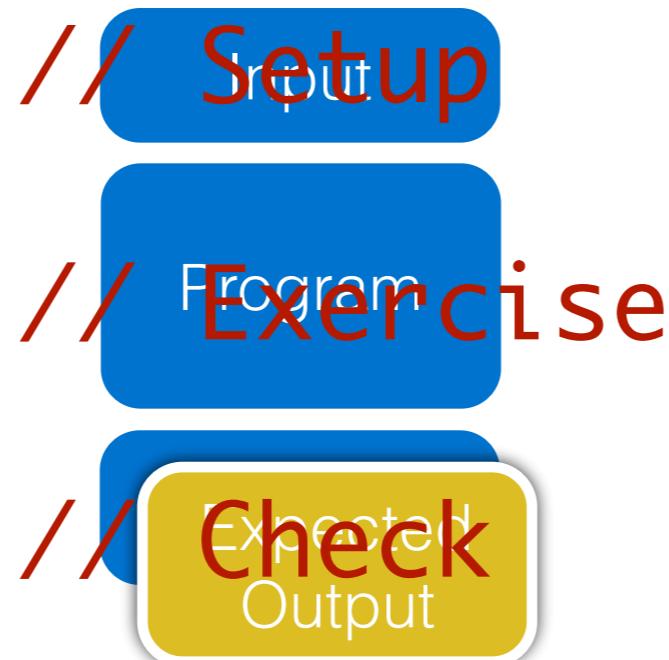
JUnit

<http://junit.org/>

JUNIT TESTING

@Test

```
public void test()  
{
```



```
}
```

JUnit

<http://junit.org/>

JUNIT TESTING

```
@Test
```

```
public void test()
```

```
{
```

```
    int x = 2;
```

```
    int y = 2;
```

```
    // Exercise
```

```
    int result = add(x,y);
```

```
    // Check
```

```
    assertEquals(4, result);
```

```
}
```

JUnit

<http://junit.org/>

JUNIT ANNOTATIONS

@Test (timeout=100)

A test case (possibly with timeout)

@Ignore ("Reason")

Do not execute test case

@Before

Execute before every test case

@After

Execute after every test case

@BeforeClass

Execute once before all test cases

@AfterClass

Execute once after all test cases

JUNIT ASSERTIONS

`assertTrue([message], condition);`

`assertFalse([message], condition);`

`assertNull([message], object);`

`assertNotNull([message], object);`

`assertEquals([message], expected , actual);`

`assertEquals([message], expected , actual , epsilon);`

`assert[Not]Same([message], object, object);`

ORGANISING TESTS

```
public class TestMyClass {  
    @Before public void setup() { ... }  
    @After public void tearDown() { ... }  
    @Test public void test1() { ... }  
    @Test public void test0() { ... }  
}
```

TESTING EXCEPTIONS

```
@Test  
public void test() {  
    try {  
        foo.bar();  
        fail("Expected exception!");  
    } catch(Exception e) {  
        // Expected exception  
    }  
}
```

TESTING EXCEPTIONS

```
@Test(expected = Exception.class)
public void test() {
    foo.bar();
}
```

DEMO



Java



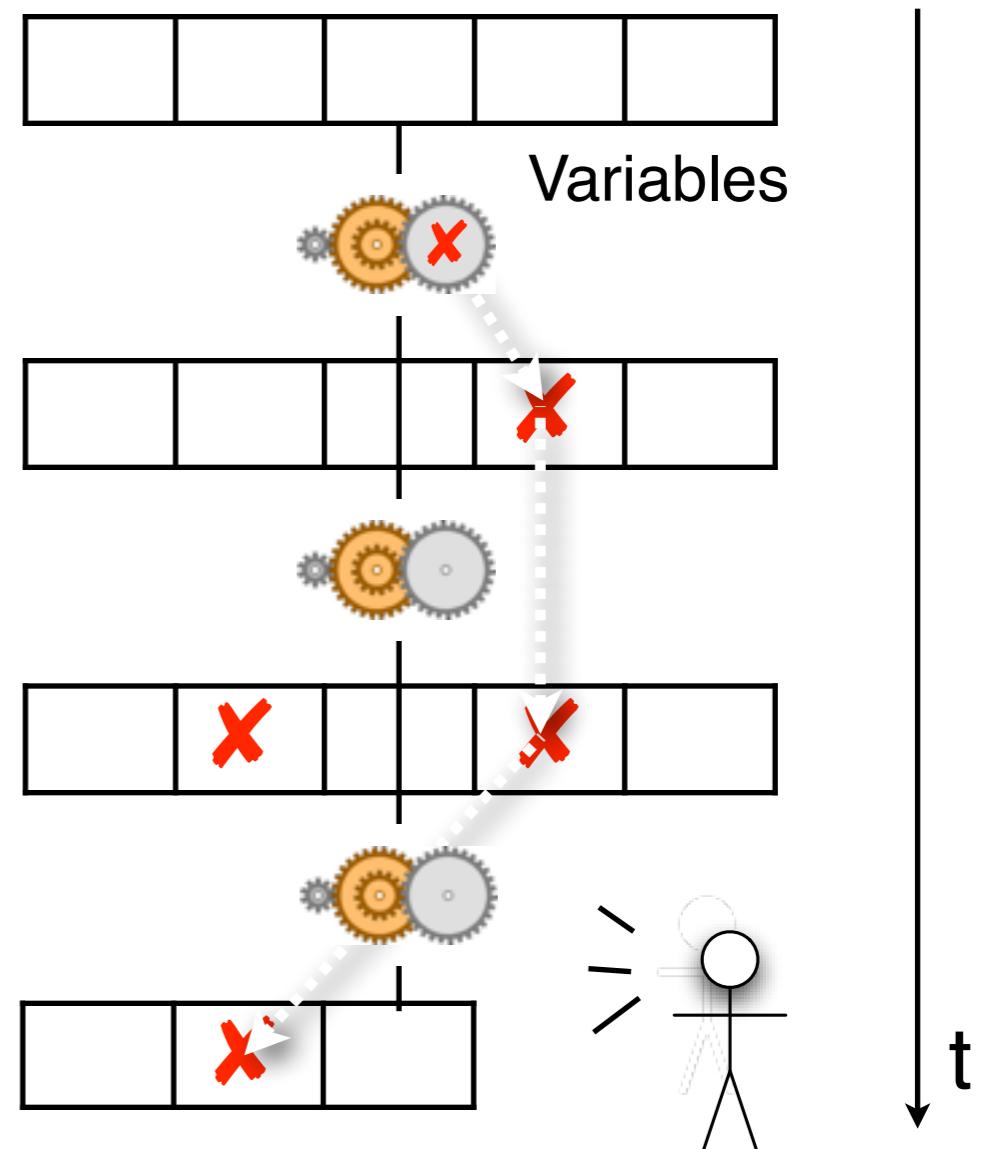
Eclipse IDE
<http://www.eclipse.org/>

JUnit

JUnit Framework

Los 3 Desafíos del Testing

- I. Tenemos que *disparar* el error en cuestión:
 - ejecutar el defecto
 - hacer que la infección se *propague*, y
 - resulte en una *falla*.
2. Debemos *reconocer* el error como tal – como una desviación de lo que es correcto, válido, o verdadero.
3. Debemos identificar *funcionalidad faltante*



Los 3 Desafíos del Testing

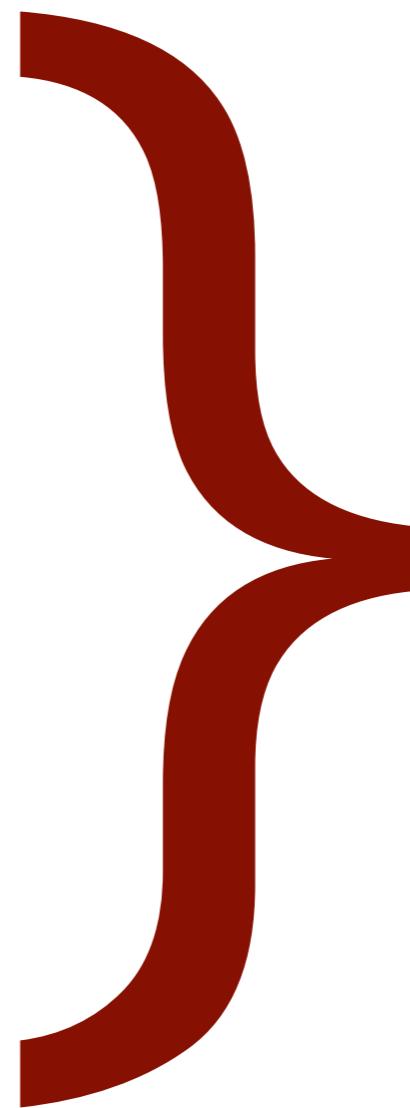
- I. Tenemos que *disparar* el error en cuestión:
 - ejecutar el defecto
 - hacer que la infección se *propague*, y
 - resulte en una *falla*.

— Cubrir tanto comportamiento como sea posible
2. Debemos *reconocer* el error como tal
 - como una desviación de lo que es correcto, válido, o verdadero.

— Proveer un oráculo
3. Debemos identificar *funcionalidad faltante*
 - Tener una especificación

Los 3 Desafíos del Testing

- I. Tenemos que *disparar* el error en cuestión:
 - ejecutar el defecto
 - hacer que la infección se *propague*, y
 - resulte en una *falla*.
2. Debemos *reconocer* el error como tal – como una desviación de lo que es correcto, válido, o verdadero.
3. Debemos identificar *funcionalidad faltante*



En una forma
eficiente
(realizable)

Los 3 Desafíos del Testing

I. Tenemos que *disparar* el error en cuestión:

- ejecutar el defecto
- hacer que la infección se *propague*, y
- resulte en una *falla*.

2. Debemos *reconocer* el error como tal – como una desviación de lo que es correcto, válido, o verdadero.

3. Debemos identificar *funcionalidad faltante*

– Cubrir tanto comportamiento como sea posible

– Proveer un oráculo

– Tener una especificación

Los 3 Desafíos del Testing

I. Tenemos que *disparar* el error en cuestión:

- ejecutar el defecto
- hacer que la infección se *propague*, y
- resulte en una *falla*.
- result in a *failure*.

— Cubrir tanto
comportamiento como
sea posible

Qué es exactamente
el
“comportamiento”?

Los 3 Desafíos del Testing

I. Tenemos que *disparar* el error en cuestión:

- ejecutar el defecto
- hacer que la infección se propague, y
- resulte en una *falla*.
- result in a *failure*

debemos alcanzar cada línea del programa

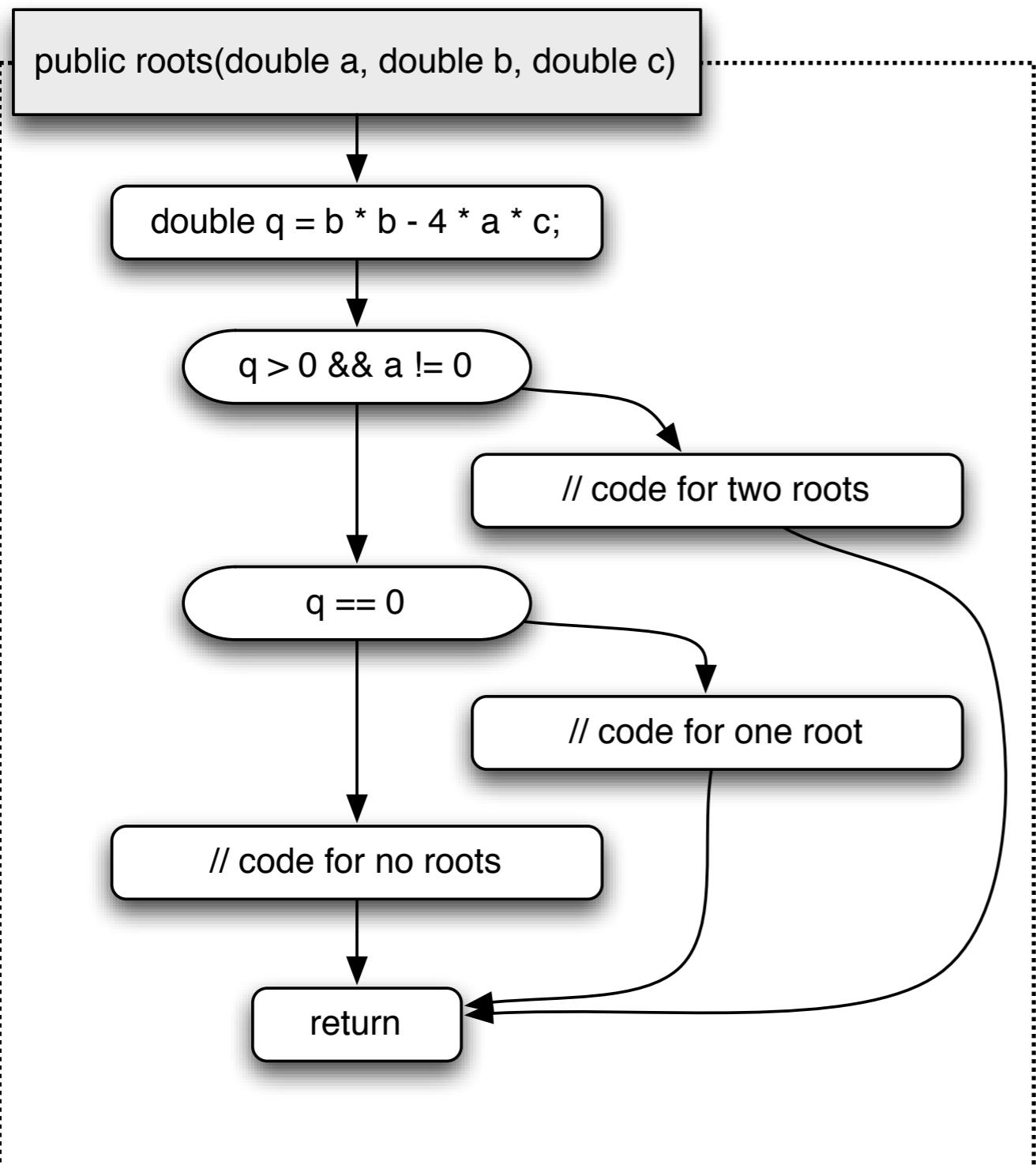
Test Cases vs. Test Suites

- **Test Case:** compuesto por código para *setup*, *exercise*, *check*
 - requiere de alguna noción de oráculo
- **Test Suite:** un conjunto de test cases
 - sin orden necesariamente

Criterios de Adecuación

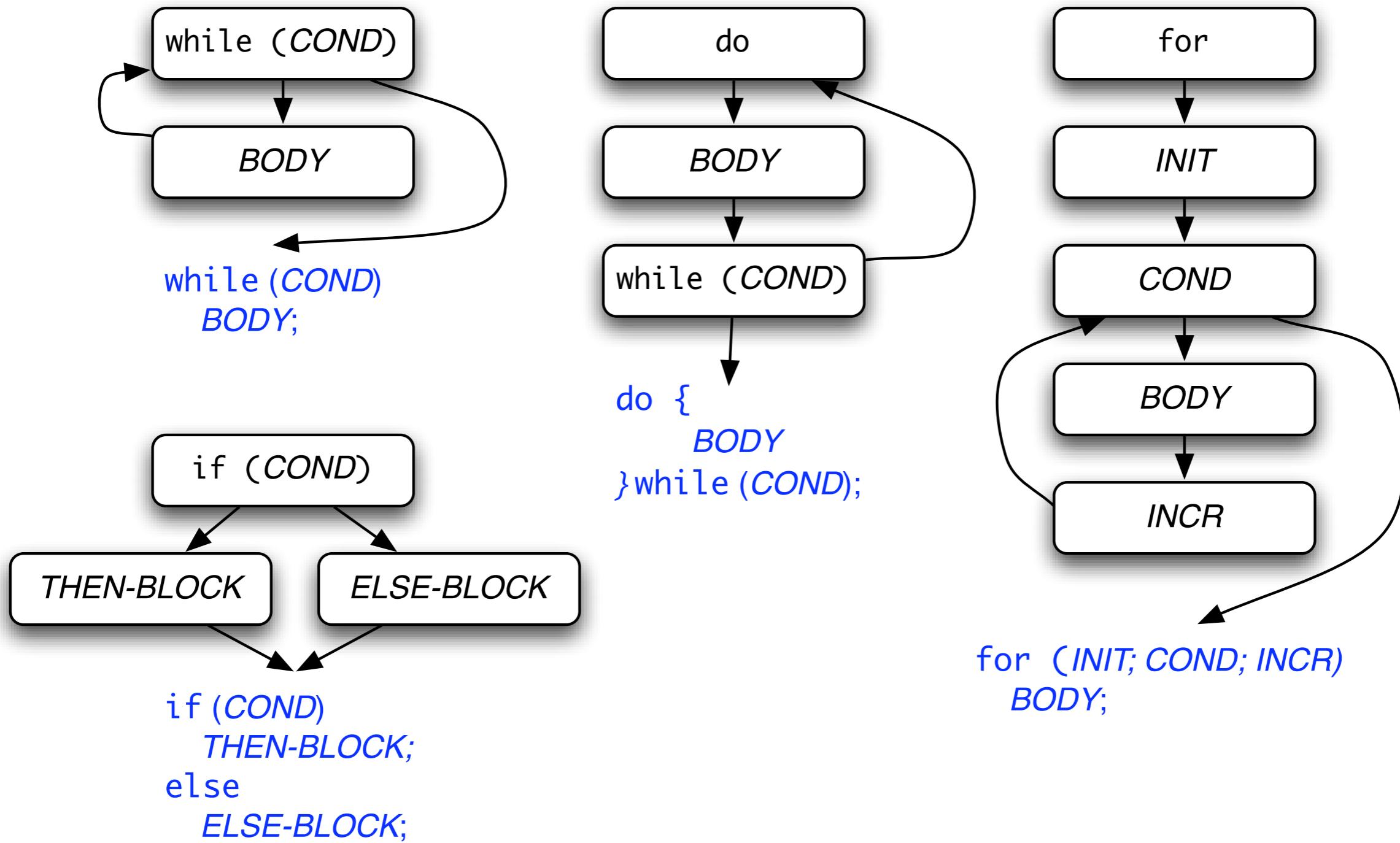
- ¿Cómo sabemos que una test suite es “suficientemente buena”?
- Un criterio de adecuación de test es un predicado que es verdadero o falso para un par $\langle \text{programa}, \text{test suite} \rangle$
- Usualmente expresado en forma de una regla – e.g., “todos los statements deben ser ejecutados”

Testing Estructural



- El *control flow graph* (CFG) puede servir como un criterio de adecuación para test cases
- Cuanto mas “partes” son ejecutadas (cubiertas), mayores las chances de un test de descubrir una falla
- “partes” pueden ser: nodos, ejes, caminos, decisiones...

Control Flow Patterns



Statement Testing

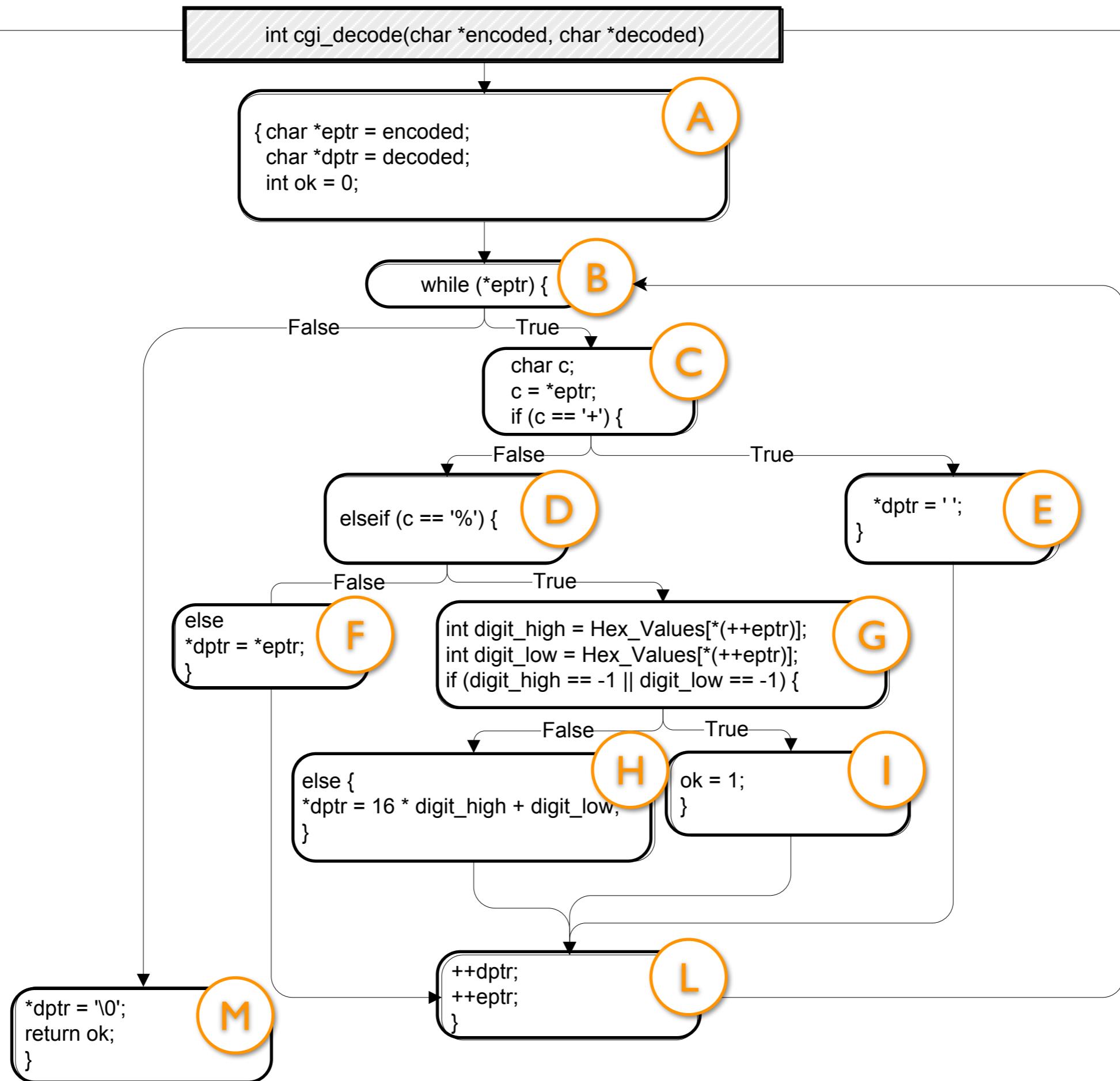
- Criterio de Adecuación: cada statement (o nodo en el CFG) *debe ser ejecutado al menos una vez*
- Idea: un defecto en un statement sólo puede ser revelado ejecutando el defecto
- Cobertura:
$$\frac{\text{\# statements ejecutados}}{\text{\# statements}}$$

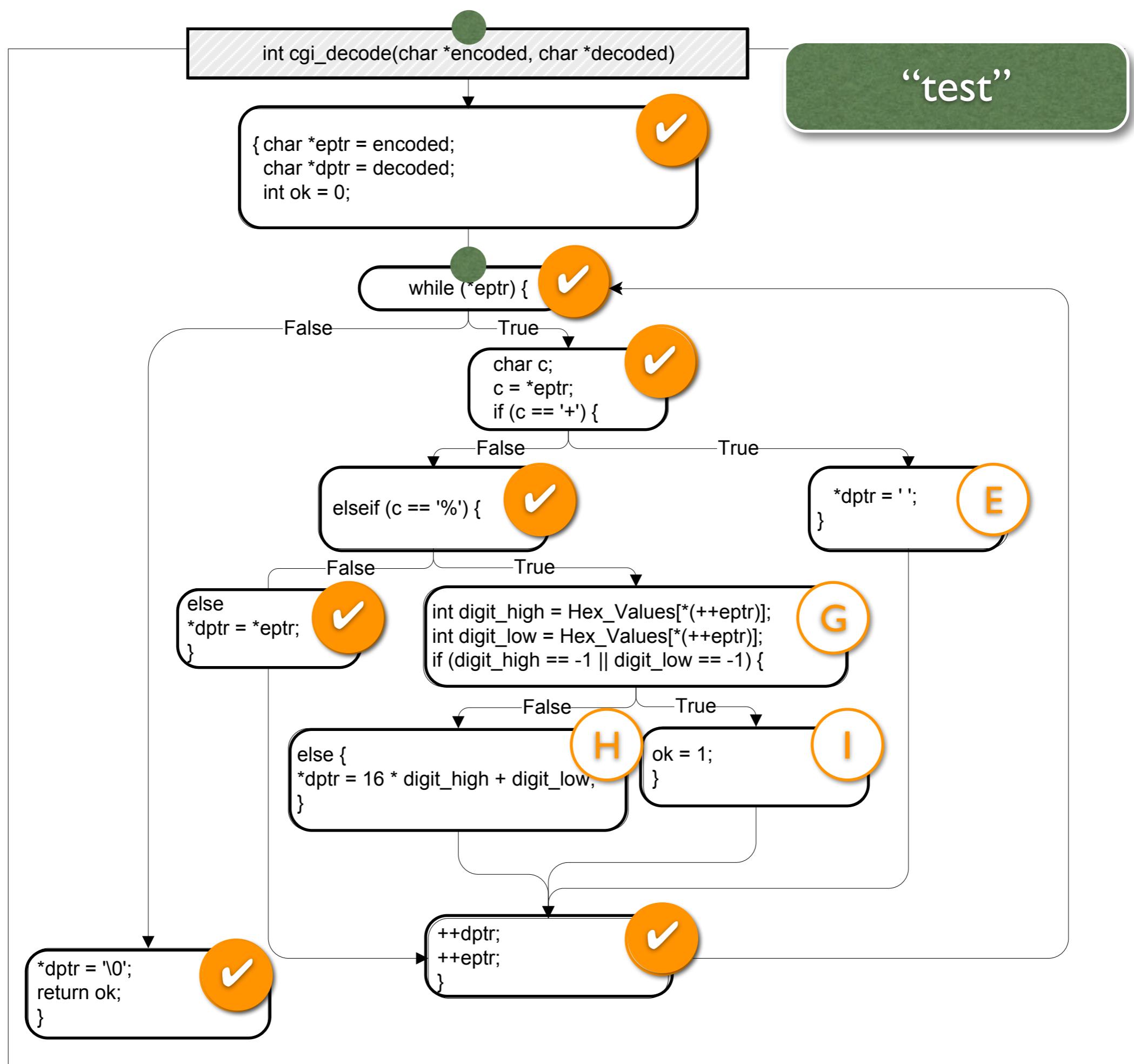
Ejemplo: cgi_decode

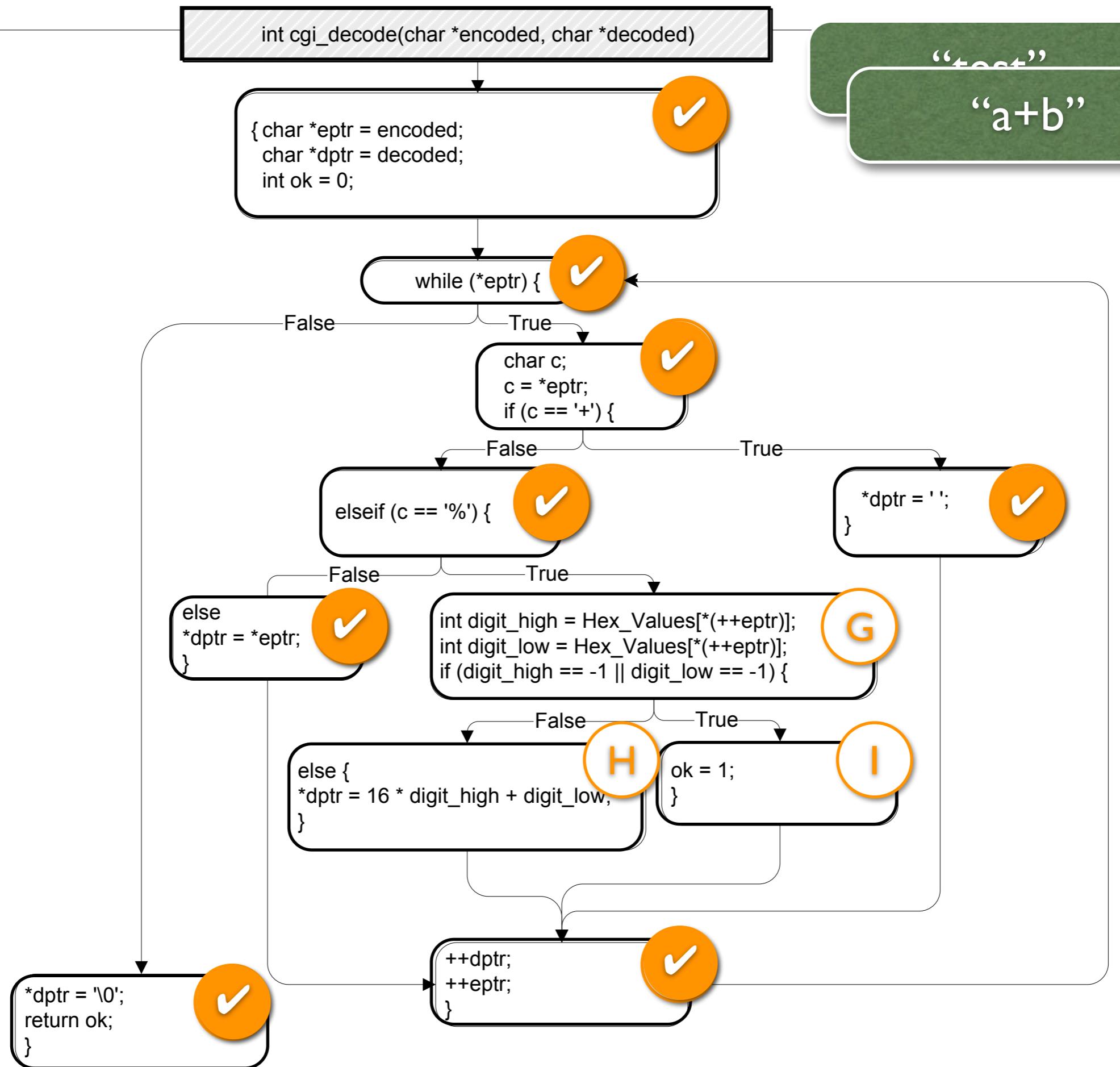
```
/**  
 * @title cgi_decode  
 * @desc  
 * Translate a string from the CGI encoding to plain ascii text  
 * '+' becomes space, %xx becomes byte with hex value xx,  
 * other alphanumeric characters map to themselves  
 *  
 * returns 0 for success, positive for erroneous input  
 * 1 = bad hexadecimal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

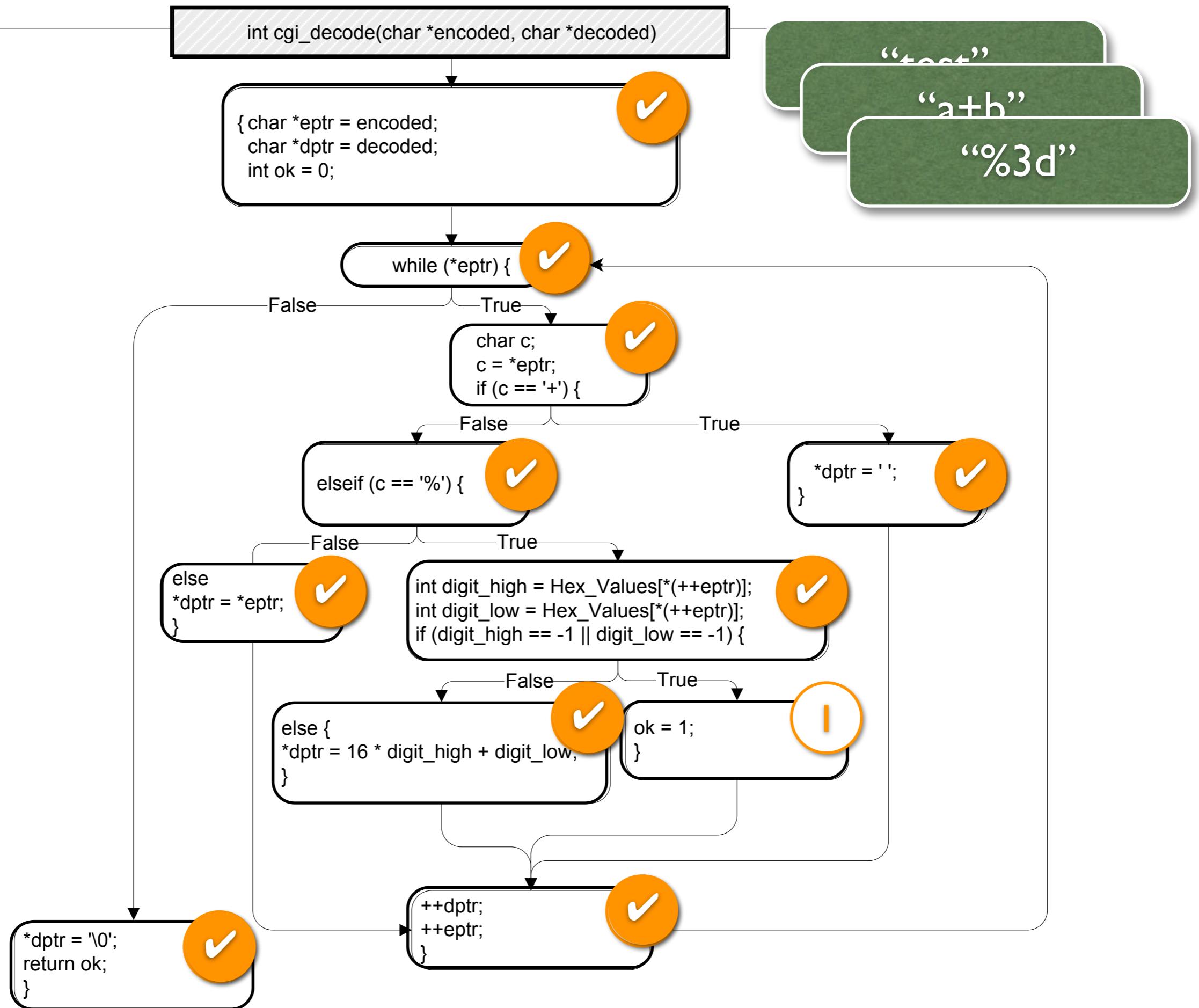
```
while (*eptr) /* loop to end of string ('\0' character) */ B
{
    char c; C
    c = *eptr;
    if (c == '+') { /* '+' maps to blank */
        *dptr = ' '; E
    } else if (c == '%') { /* '%xx' is hex for char xx */ D
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)]; 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
}

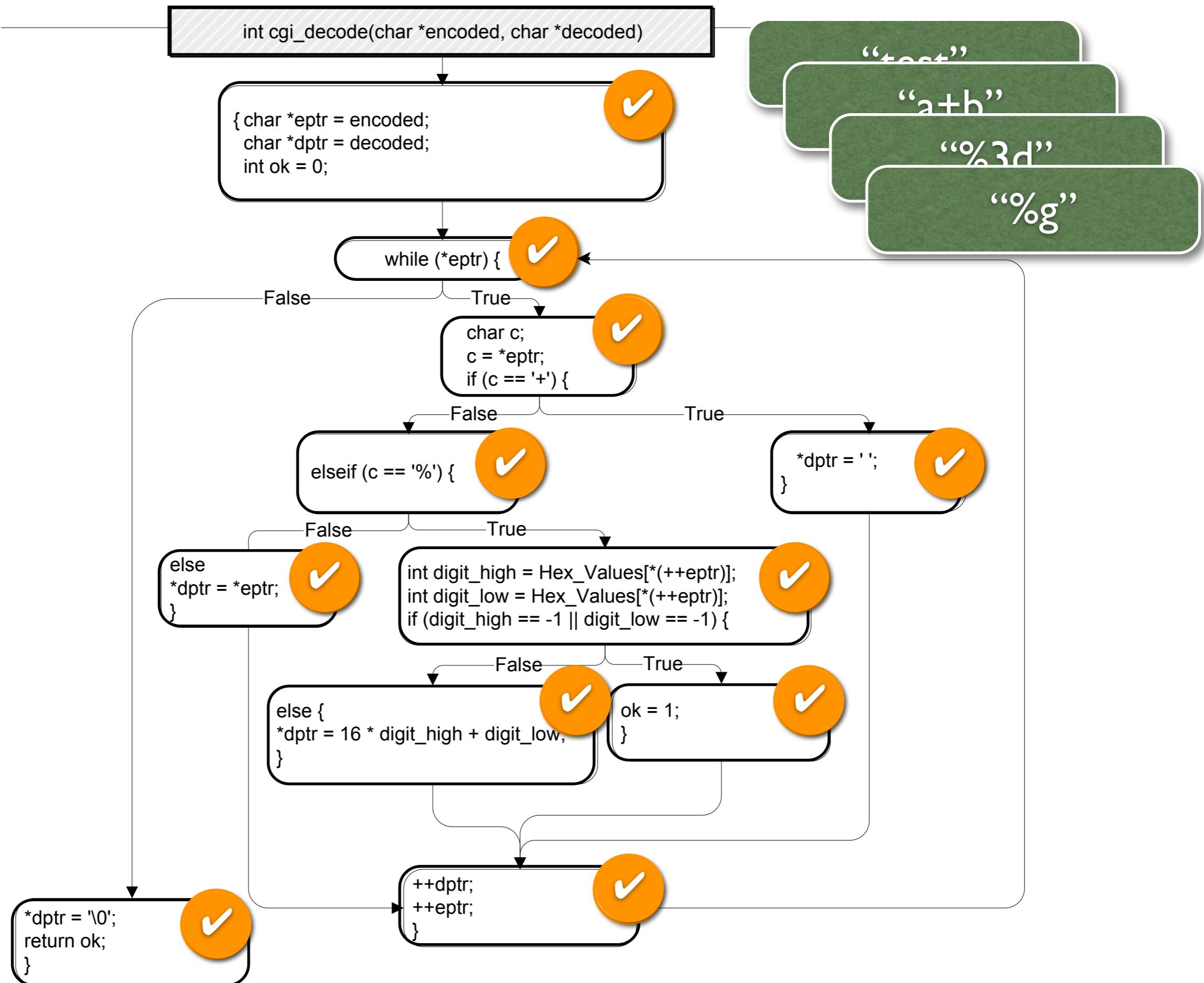
*dptr = '\0'; /* Null terminator for string */ M
return ok;
}
```





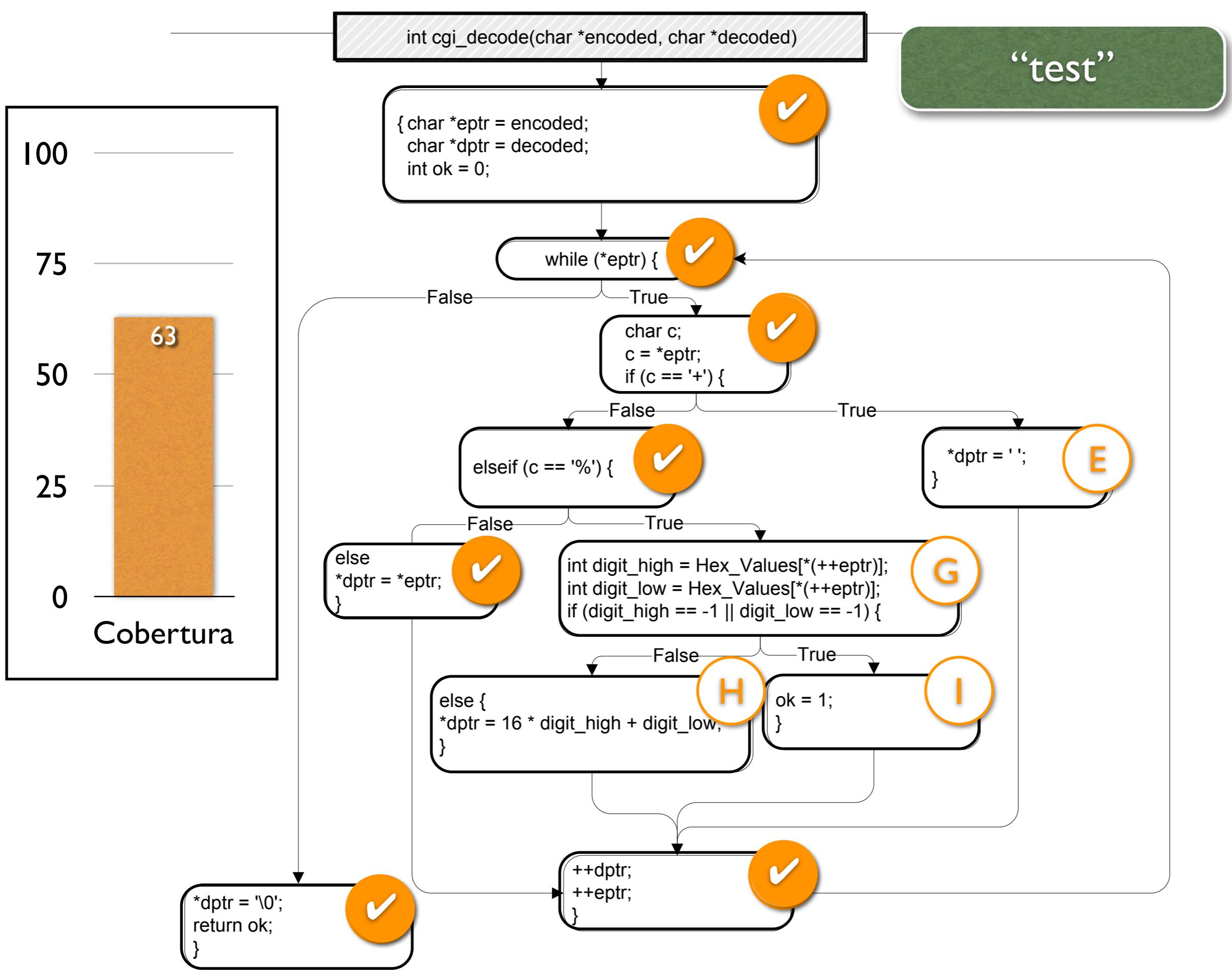






Statement Testing

- Cobertura: $\frac{\text{\# statements ejecutados}}{\text{\# statements}}$



100

75

50

25

0

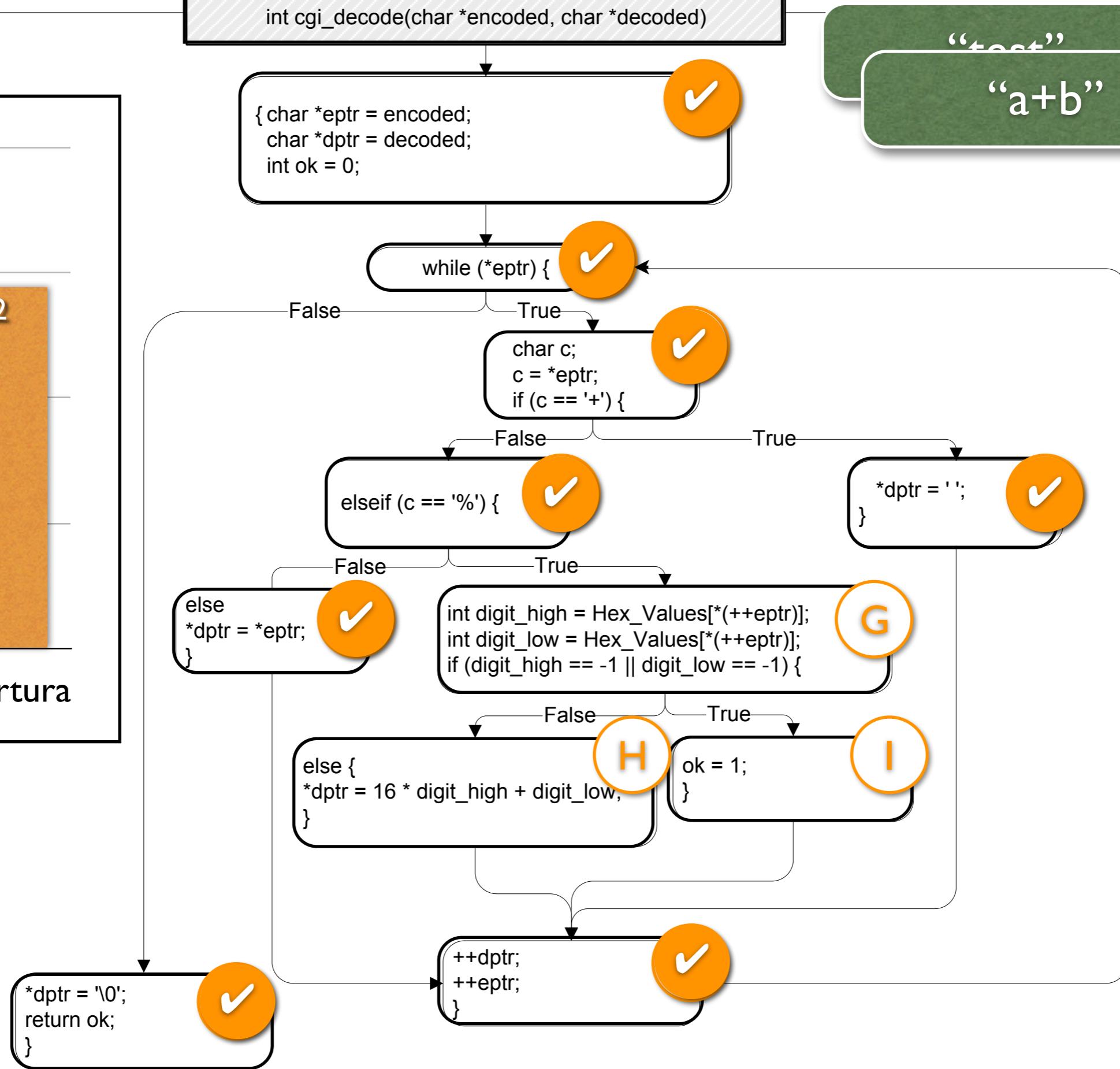
Cobertura

72

```
int cgi_decode(char *encoded, char *decoded)
```

“test”

“a+b”



100

75

50

25

0

Cobertura

91

`int cgi_decode(char *encoded, char *decoded)`

```
{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;
```

`"test"``"a+b"``"%3d"``while (*eptr) {`

```
char c;
c = *eptr;
if (c == '+') {
```

`elseif (c == '%') {``*dptr = ' ';`

```
else
*dptr = *eptr;
```



```
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
```

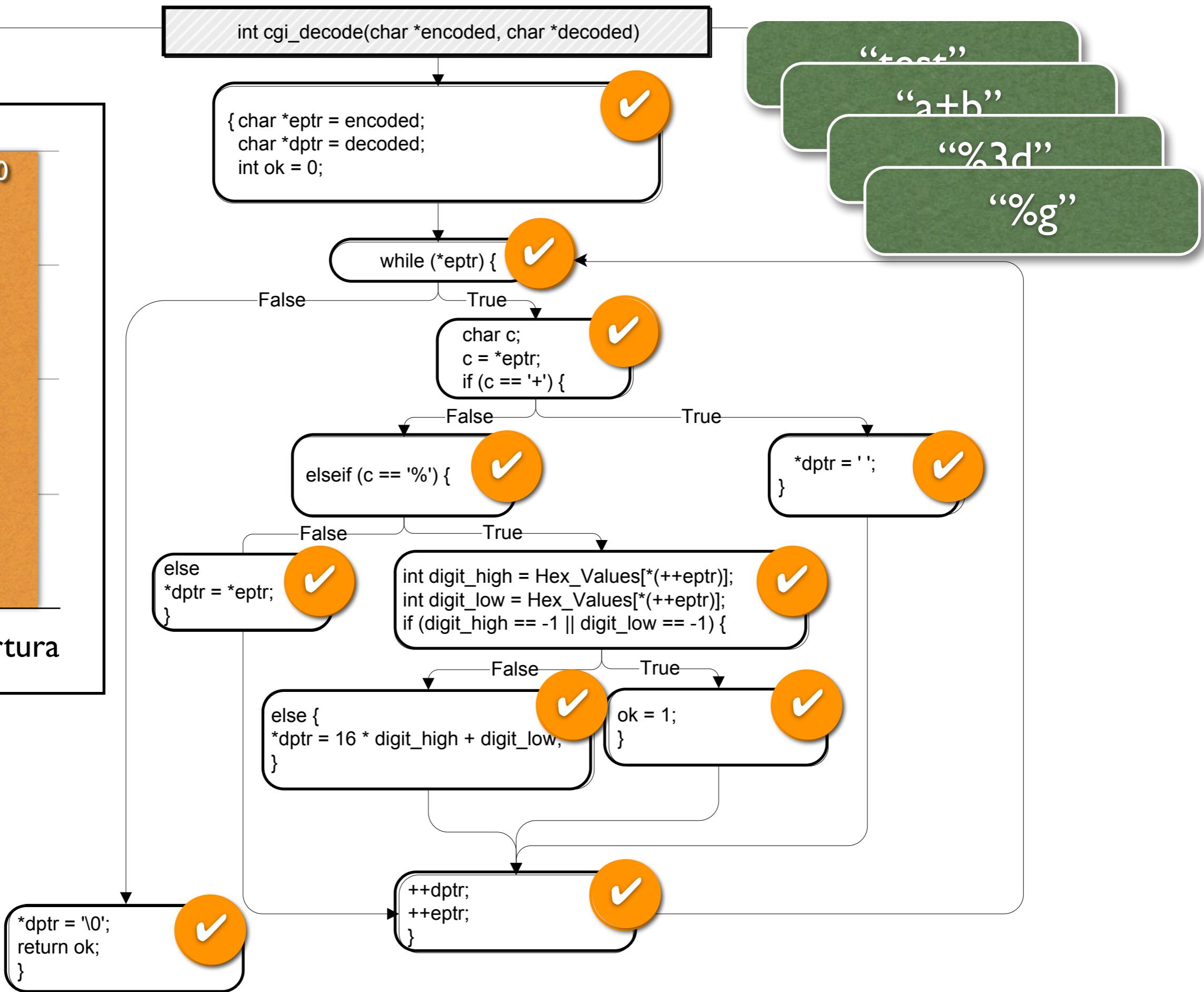
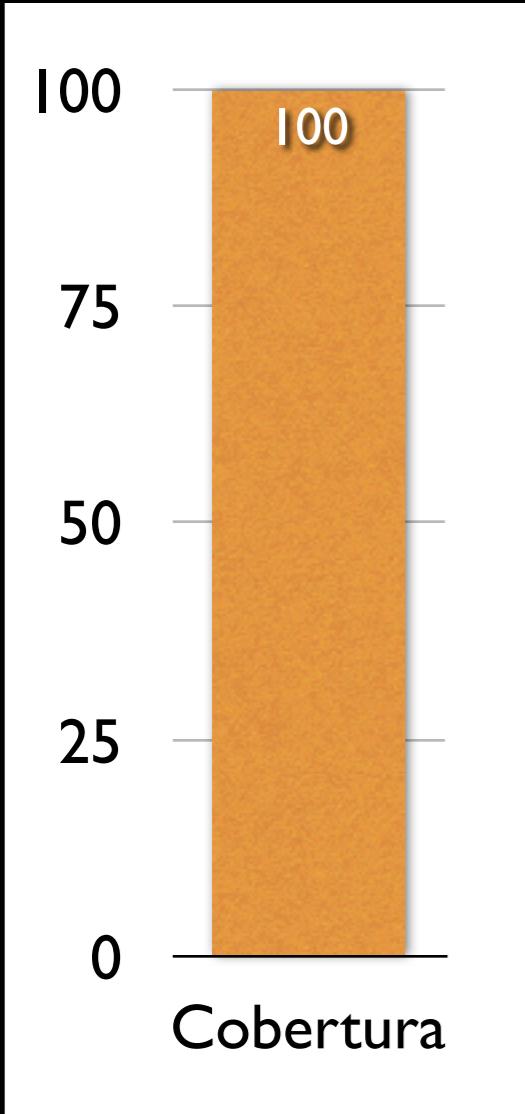


```
else {
*dptr = 16 * digit_high + digit_low,
}
```

`ok = 1;`

```
*dptr = '\0';
return ok;
}
```

`++dptr;
++eptr;
}``++dptr;
++eptr;
}`



Calculando la Cobertura

- La Cobertura es computada automáticamente mientras el programa es ejecutado
- Requiere la *instrumentación* en tiempo de compilación
- Luego de la ejecución, una herramienta de *cobertura* analiza y resume los resultados



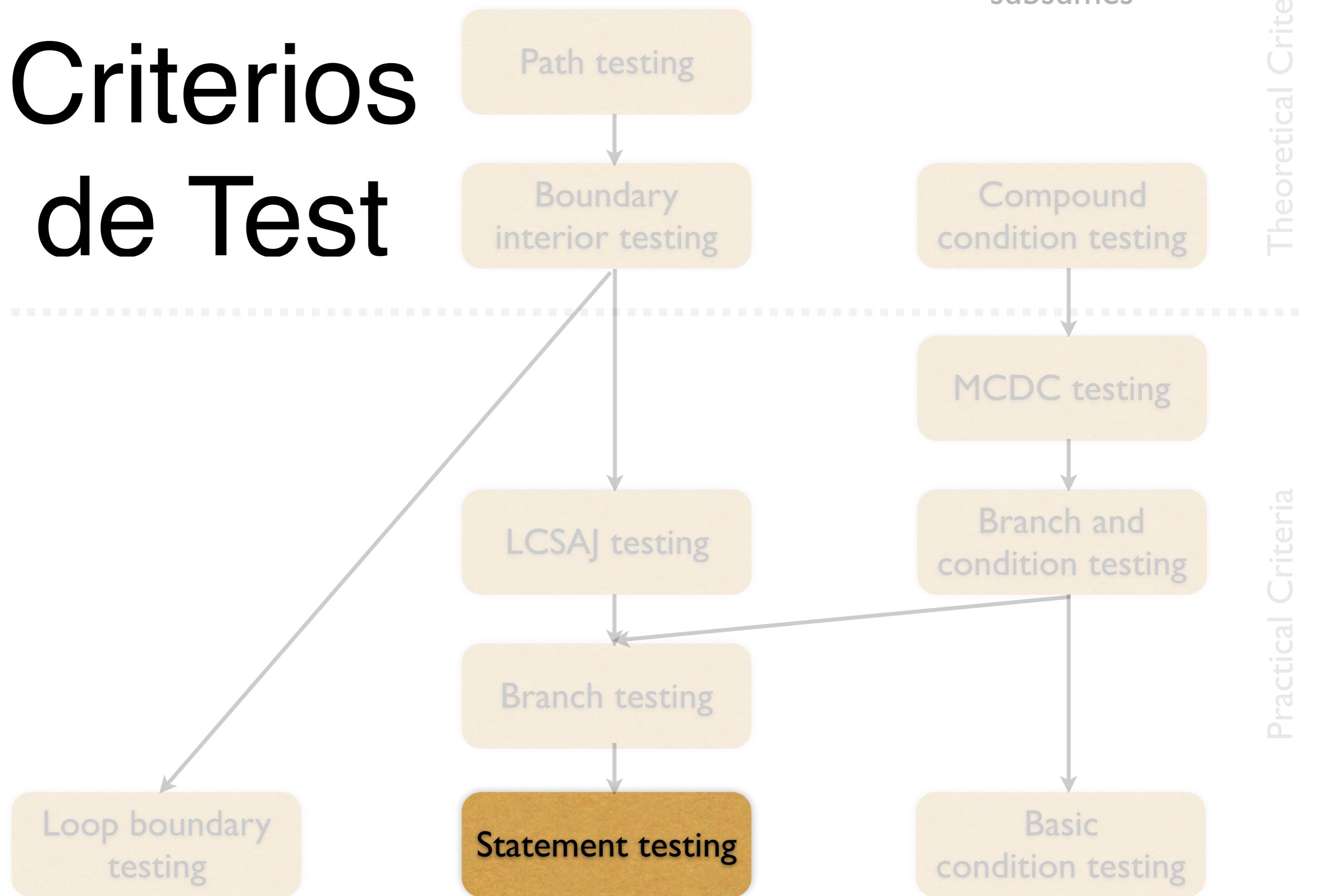
Pippin: cgi_encode — less — 80x24

```
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 */
-: 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:}
```

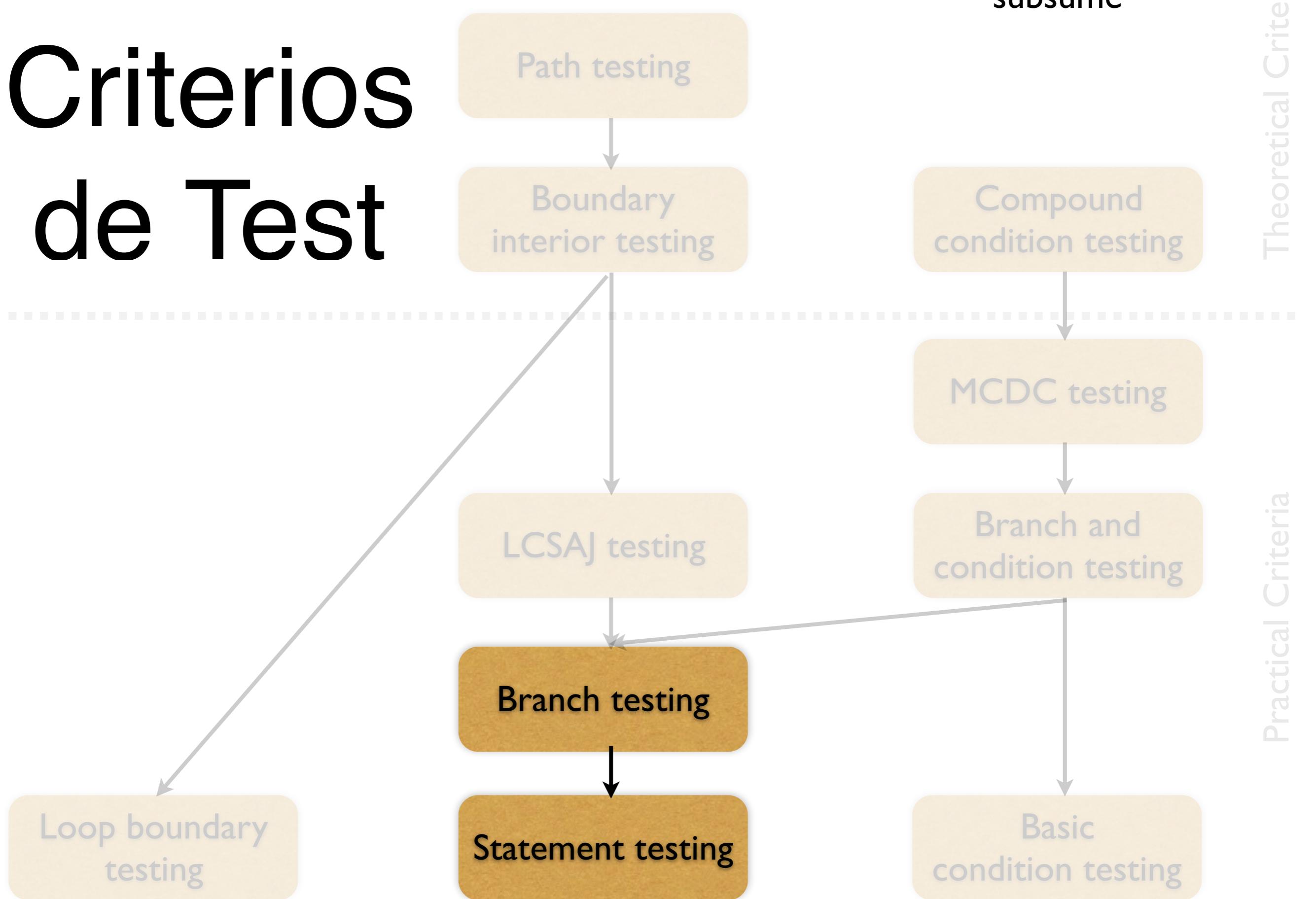
(END)



Criterios de Test

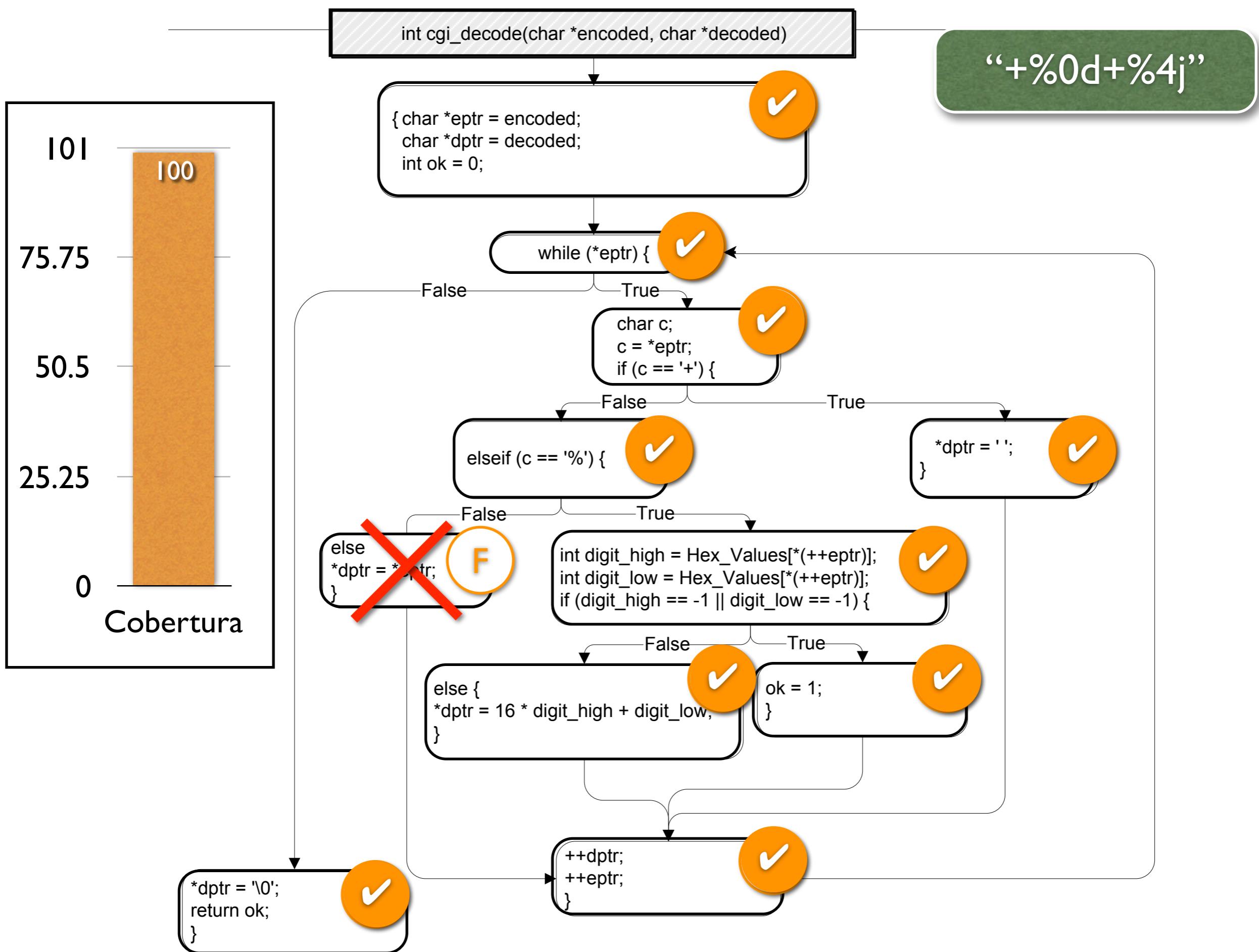


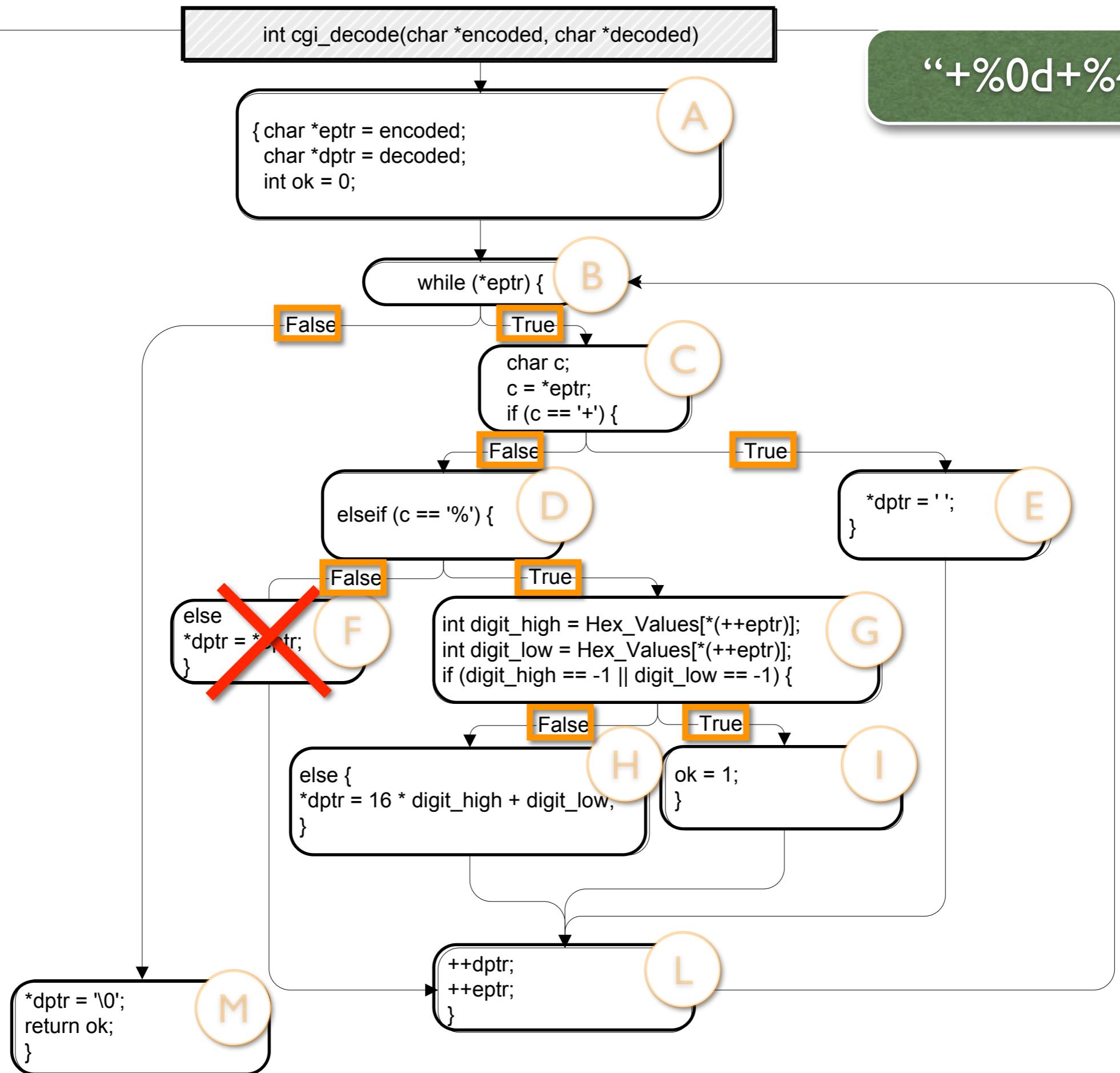
Criterios de Test

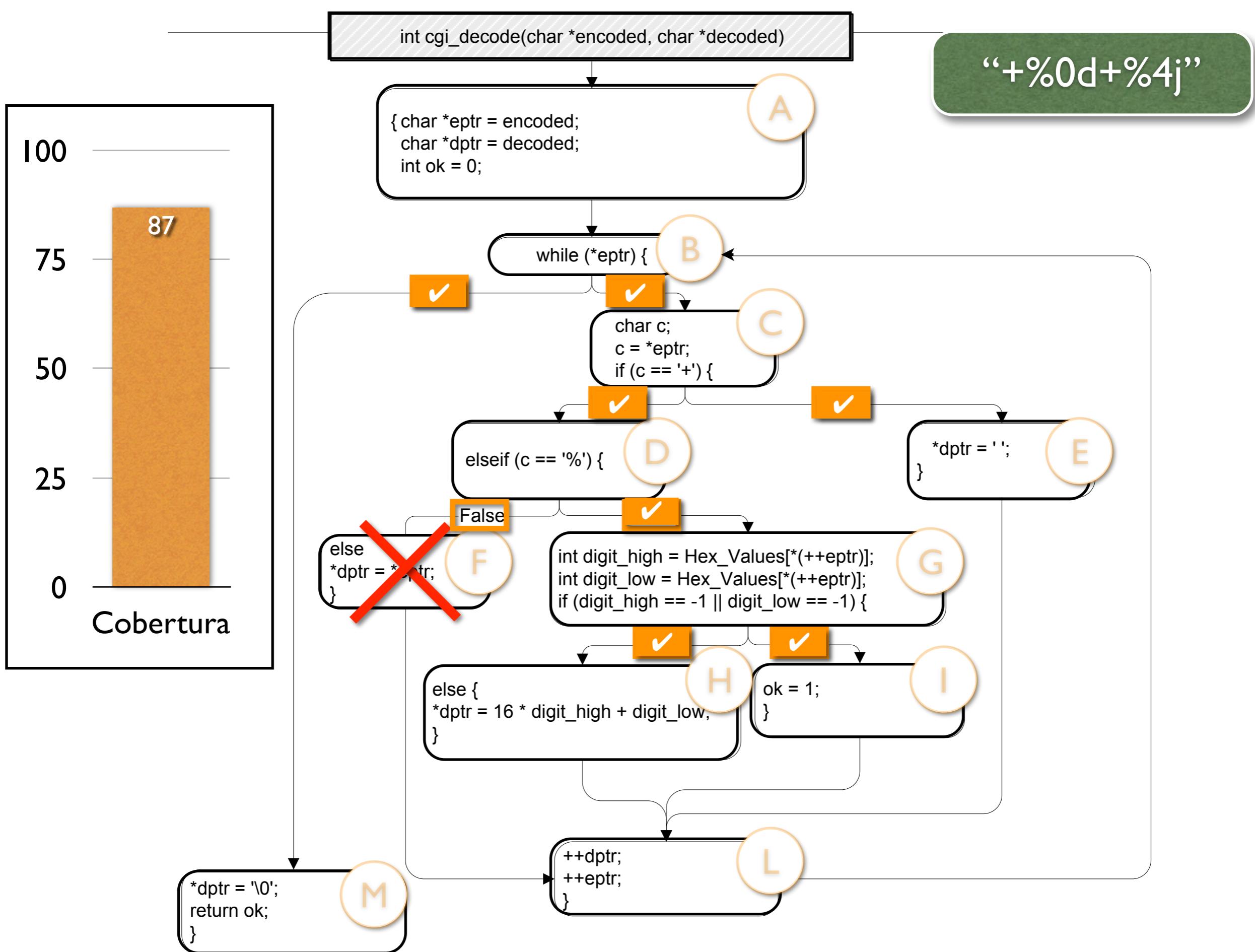


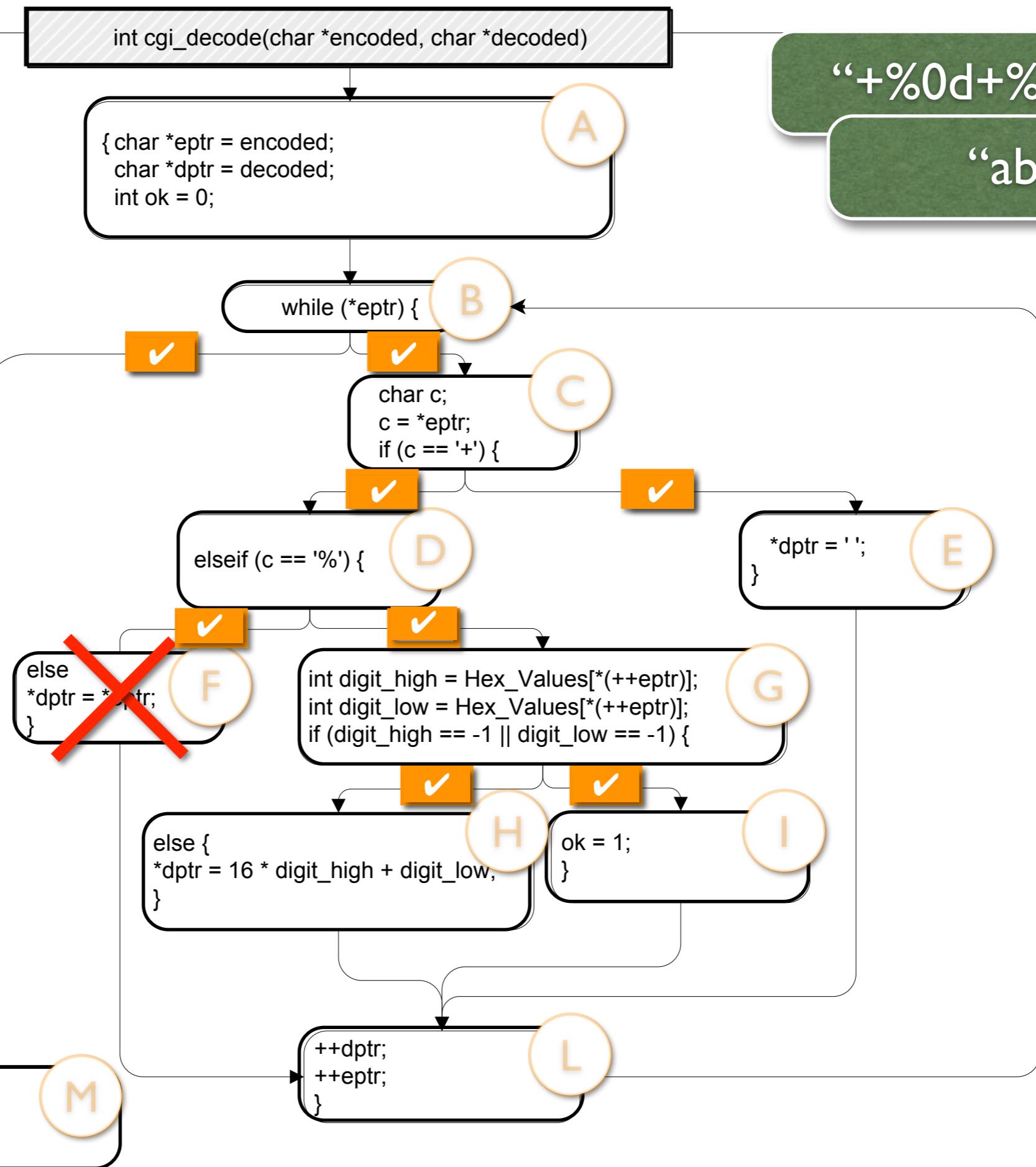
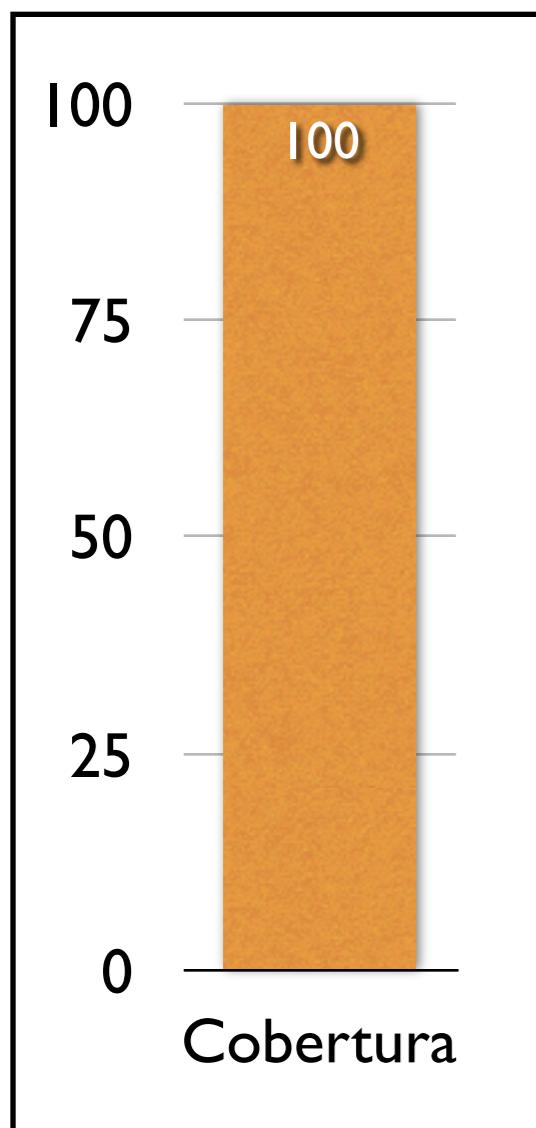
Theoretical Criteria

Practical Criteria









“+%0d+%4j”

“abc”

Branch Testing

- Criterio de adecuación: cada branch en el CFG debe ser ejecutado al menos una vez
- Cobertura :
$$\frac{\text{\# branches ejecutados}}{\text{\# branches}}$$
- Subsume Statement Testing
ya que al recorrer todos los ejes recorremos todos los nodos
- Most widely used criterion in industry

Taller

- Ejercicio #1: Java/JUnit/Cobertura
 - Escribir JUnit Test Suites 100% branch/ statemente coverage en las clases StackAr

Y ahora...

**¡Vamos a construir
nuestra propia
herramienta de
cobertura!**

A person is wearing a brown horse head mask and an orange hooded garment. A cat's head is visible behind the person's shoulders, looking towards the camera. The background is a blurred outdoor scene.

Wat

cgi_decode.py

```
def cgi_decode(s):
    t = "" A
    i = 0
    while i < len(s):
        c = s[i] C
        if c == '+': B
            t = t + ' '
        elif c == '%': D
            digit_high = s[i + 1] E
            digit_low = s[i + 2] G
            i = i + 2
            if (hex_values.has_key(digit_high) and
                hex_values.has_key(digit_low)):
                v = (hex_values[digit_high] * 16 +
                      hex_values[digit_low]) H
                t = t + chr(v)
        else:
            raise Exception I
        else:
            t = t + c F
            i = i + 1 L
    return t M
```

Tracing en Python

- En Python, trazar ejecuciones es mucho más simple que en los lenguajes compilados.
- La función `sys.settrace(f)` define `f()` como una función de tracing que es invocada (llamada) por cada línea ejecutada
- `f()` tiene acceso al estado completo del *intérprete*

Tracing en Python

frame actual (PC + variables)

```
import sys
```

```
def traceit(frame, event, arg):
    if event == "line":
        lineno = frame.f_lineno
        print "Line", lineno, frame.f_locals
    return traceit
```

```
sys.settrace(traceit)
```

"line", "call", "return", ...

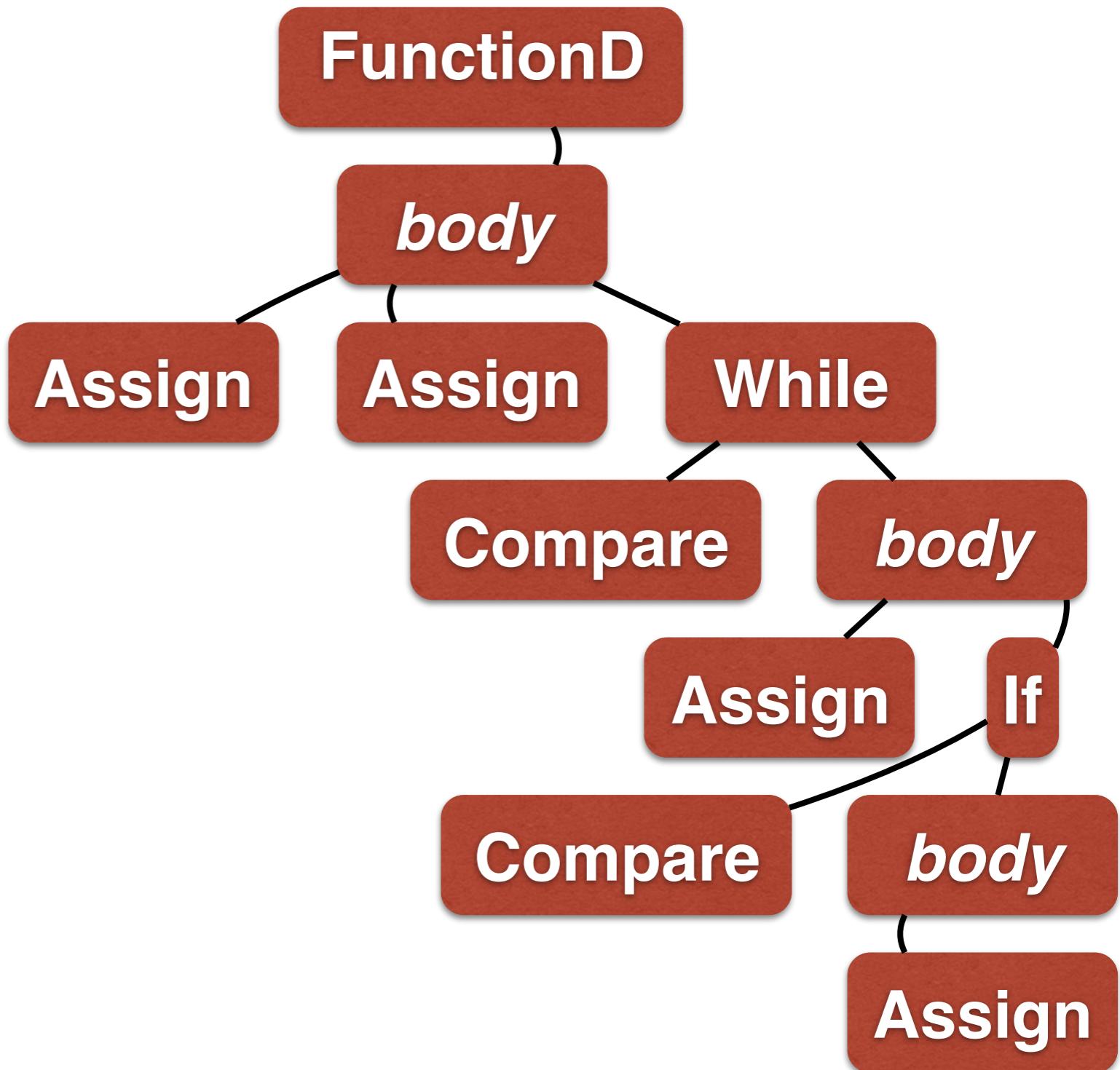
tracer a ser usado
en este scope (este mismo)

Tracing de Branches

- Tracer los *branches tomados* es fácil – sólo tracea todos los *pares* de líneas ejecutados secuencialmente
- Pero cómo obtenemos *todos los branches posibles*?
- Se necesita analizar el programa *estáticamente* (sin ejecutarlo)

Abstract Syntax Trees

```
def cgi_decode(s):
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t = t + ' '
        elif c == '%':
            ...
        else:
            t = t + c
        i = i + 1
    return t
```



Python AST

- El módulo de AST de Python convierte un archivo de programas Python en su *abstract syntax tree (AST)*
- El árbol puede ser visitado usando el design-pattern “visitor”

Python AST

root del AST

`import ast`

`root = ast.parse('x = 1')`
`print ast.dump(root)`

input de Python

|

AST como string
(para debugging)

<https://docs.python.org/2/library/ast.html#ast.AST>

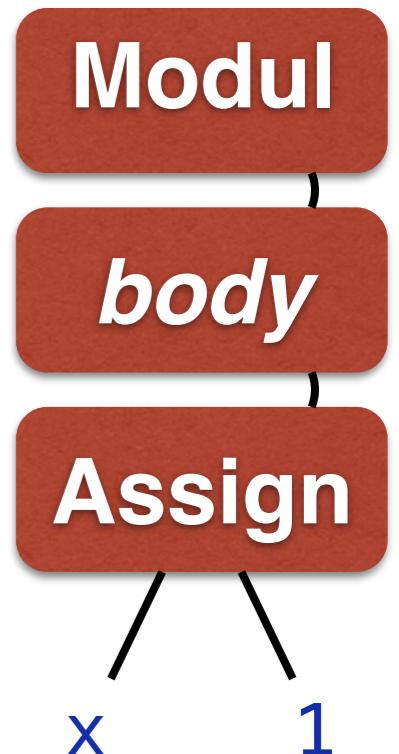
Python AST

```
import ast
```

```
root = ast.parse('x = 1')
print ast.dump(root)
```

→

```
Module(
    body = [
        Assign(
            targets = [
                Name(id = 'x', ctx = Store())
            ],
            value = Num(n=1)
        )
    ]
)
```



AST Visitor

- La clase `ast.NodeVisitor` provee un método `visit(n)` que recorre todos los subnodos de *n*
- Debe ser subclaseada para ser extendida
- En cada nodo *n* de tipo *TYPE*, el método `visit_TYPE(n)` es llamado si este exists
- Si no existe `visit_TYPE(n)`, el método `generic_visit()` recorre todos los hijos

AST Visitor

```
class IfVisitor(ast.NodeVisitor):  
    def visit_If(self, node):  
        print "if", node.lineno, ":"  
        for n in node.body:  
            print "    ", n.lineno  
        print "else:"  
        for n in node.orelse:  
            print "    ", n.lineno  
    self.generic_visit(node)
```

muestra cuerpo
y parte “else”

número de línea

recorre los hijos

AST Visitor

```
root = ast.parse(open('cgi_decode.py').read())
v = IfVisitor()
v.visit(root)
```

Lee source Python

Visita todos los nodos IF

```
→ if 34 :
35
else:
36 if 36 :
37
38
39
40
else:
47 if 40 :
42
43
else:
45 if 81 :
82
83
else:
```

AST Visitor

```
def cgi_decode(s):
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t = t + ' '
        elif c == '%':
            digit_high = s[i + 1]
            digit_low = s[i + 2]
            i = i + 2
            if (hex_values.has_key(digit_high) and
                hex_values.has_key(digit_low)):
                v = (hex_values[digit_high] * 16 +
                      hex_values[digit_low])
                t = t + chr(v)
            else:
                raise Exception
        else:
            t = t + c
            i = i + 1
    return t
```

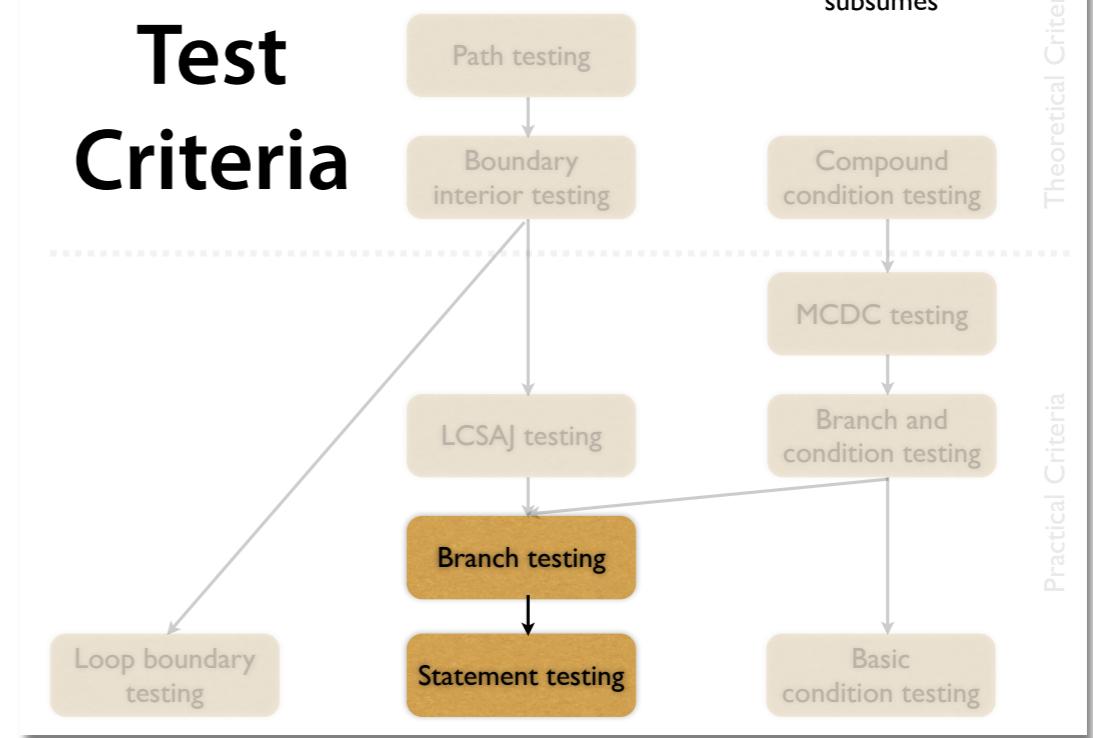
```
→ if 34 :
   35
else:
   36
if 36 :
   37
   38
   39
   40
else:
   41
      42
         43
else:
   44
      45
         46
            47
if 81 :
   82
   83
else:
```

Three Challenges of Testing

1. We must *trigger* the error in question:
 - execute the defect
 - have the infection propagate, and
 - result in a *failure*.
2. We must *recognize* the error as such – as a deviation from what is correct, right, or true.
3. We must identify *missing functionality*

– *Cover as many behaviors as possible*
 – *Provide an oracle*
 – *Have a spec*

Test Criteria



AST Visitor

```

class IfVisitor(ast.NodeVisitor):
    def visit_If(self, node):
        print "if", node.lineno, ":"
        for n in node.body:
            print "  ", n.lineno
        print "else:"
        for n in node.orelse:
            print "  ", n.lineno
    self.generic_visit(node)
  
```

line number
show body
and "else" part
traverse children

Python Tracing and AST

```

import sys

def traceit(frame, event, arg):
    if event == "line":
        lineno = frame.f_lineno
        print "Line", lineno, frame.f_locals
    return traceit

sys.settrace(traceit)
  
```

```

import ast

root = ast.parse('x = 1')
print ast.dump(root)
  
```

