

Introducción al Testing Automático

Generación Automática de Tests - 2018

Ariane V

- Ultima innovación del Programa Espacial Europeo
- Vuelo Inaugural: 4 de Junio 1996





MS Zune - 31/12/08



```
year = ORIGINYEAR; /* = 1980 */

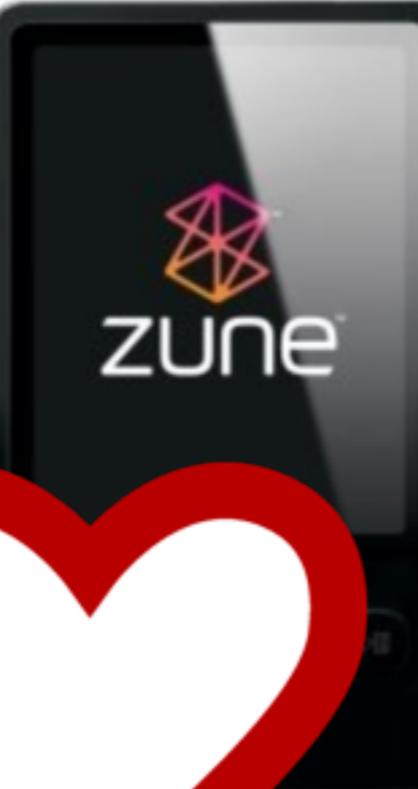
while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

MS Zune - 31/12/08





Trágicas



Molestas



Ecónomicas



Descubriendo Errores



Verificación de Programas

Programa

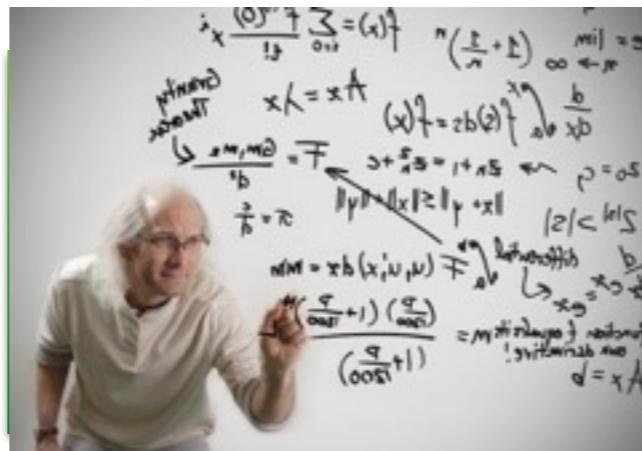


Spec



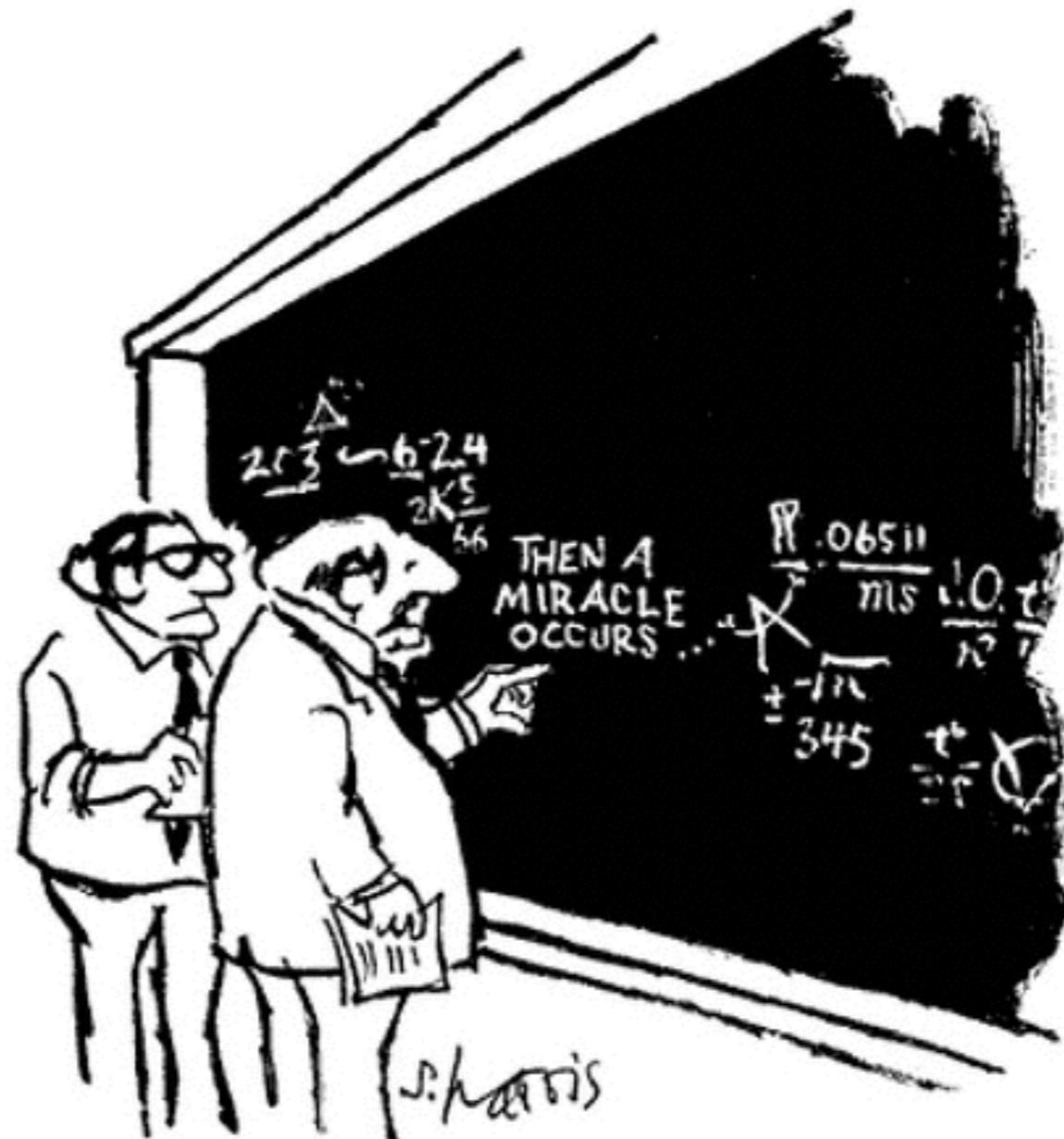
Verificación Manual de Programas

Programa



Spec

Verificación Manual de Programas



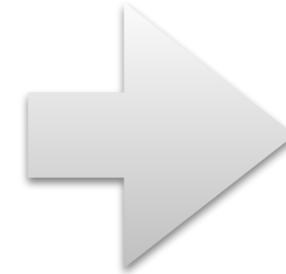
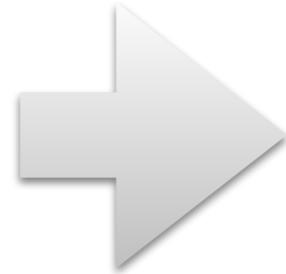
"I think you should be more explicit here in step two."

Verificación Automática de Programas

Programa



Spec

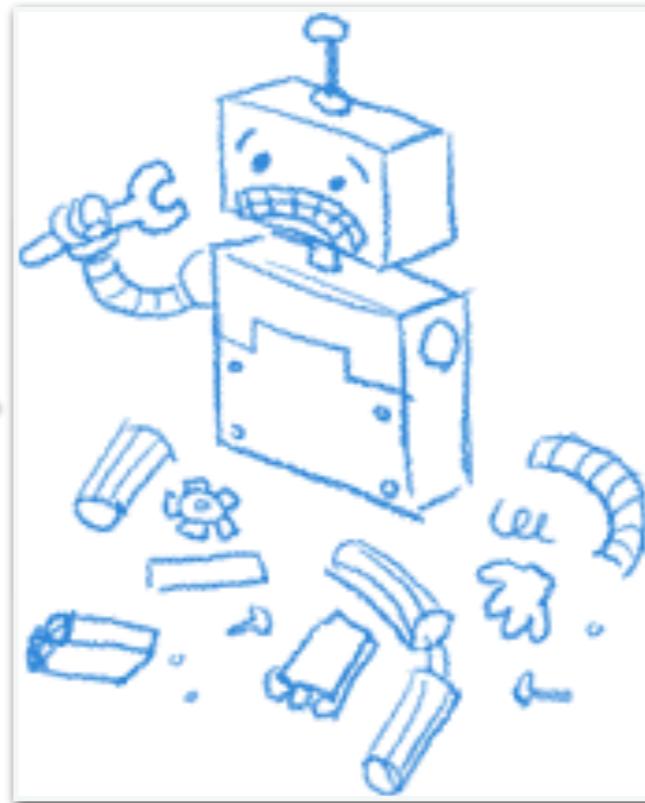
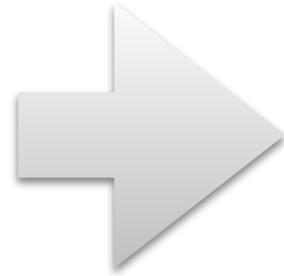


Verificación Automática de Programas

Programa



Spec



Testing



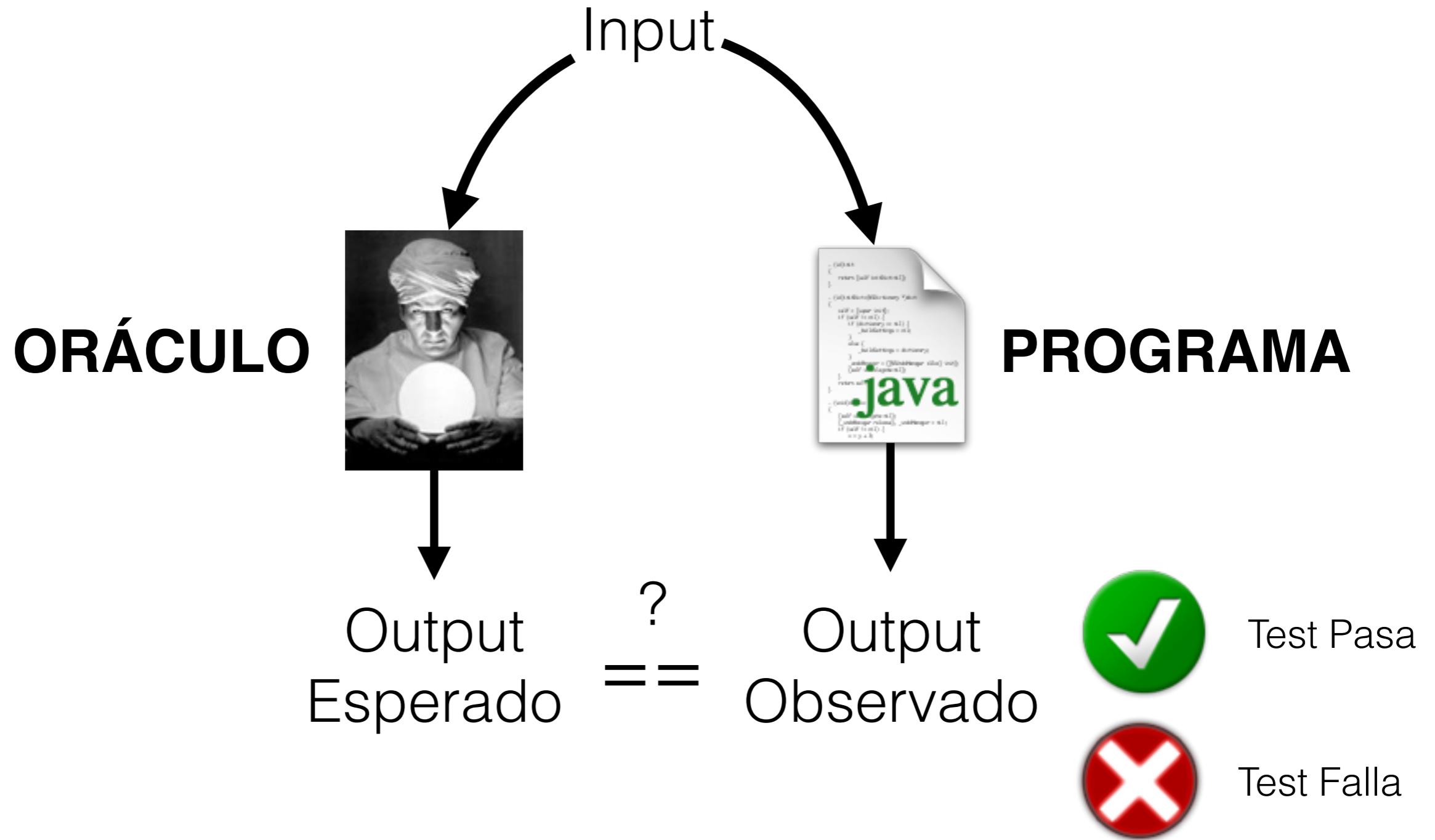
¿Qué significa hacer Testing?

Proceso de ejecutar un producto para:

- Verificar que satisface los requerimientos
- Identificar diferencias entre comportamiento **real** y comportamiento **esperado**

(IEEE Standard for Software Test Documentation, 1983)

¿Cómo hacemos Testing?



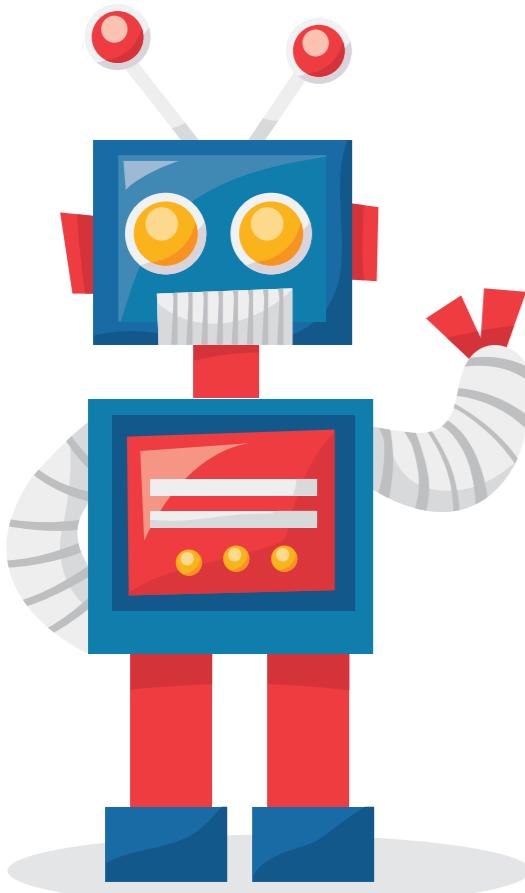
Límites del Testing



"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger Dijkstra
1930–2002

Ejecución Automática de Casos de Tests



JUnit



Selenium

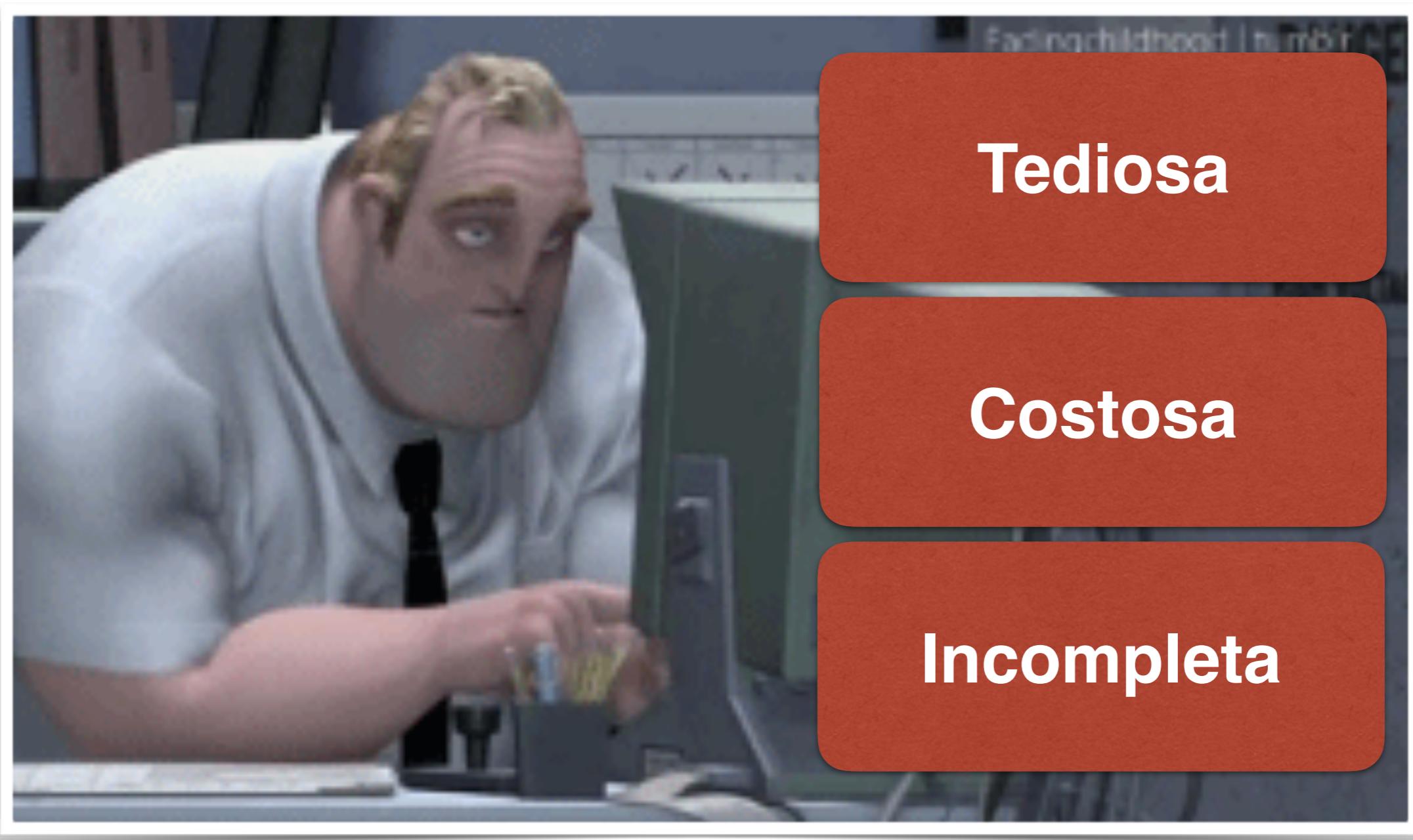


Jenkins

```
@Test  
public void newArrayListsHaveNoElements() {  
    assertThat(new ArrayList().size(), is(0));  
}
```

```
@Test  
public void sizeReturnsNumberOfElements() {  
    List instance = new ArrayList();  
    instance.add(new Object());  
    instance.add(new Object());  
    assertThat(instance.size(), is(2));  
}
```

Creación Manual de Casos de Tests



Visión

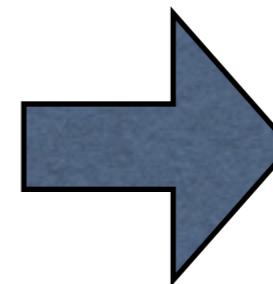
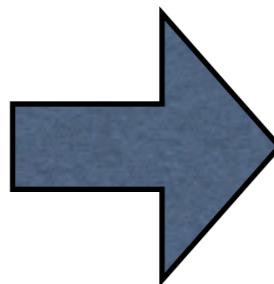
- Aprovechar el actual **poder de cómputo** para
 - Mejorar la **eficiencia** de los programadores
 - Aumentar la **calidad** de los programas
- ¿Cómo?

Automatizando la creación de los Casos de Tests

Creación Automática de Casos de Tests

Programa

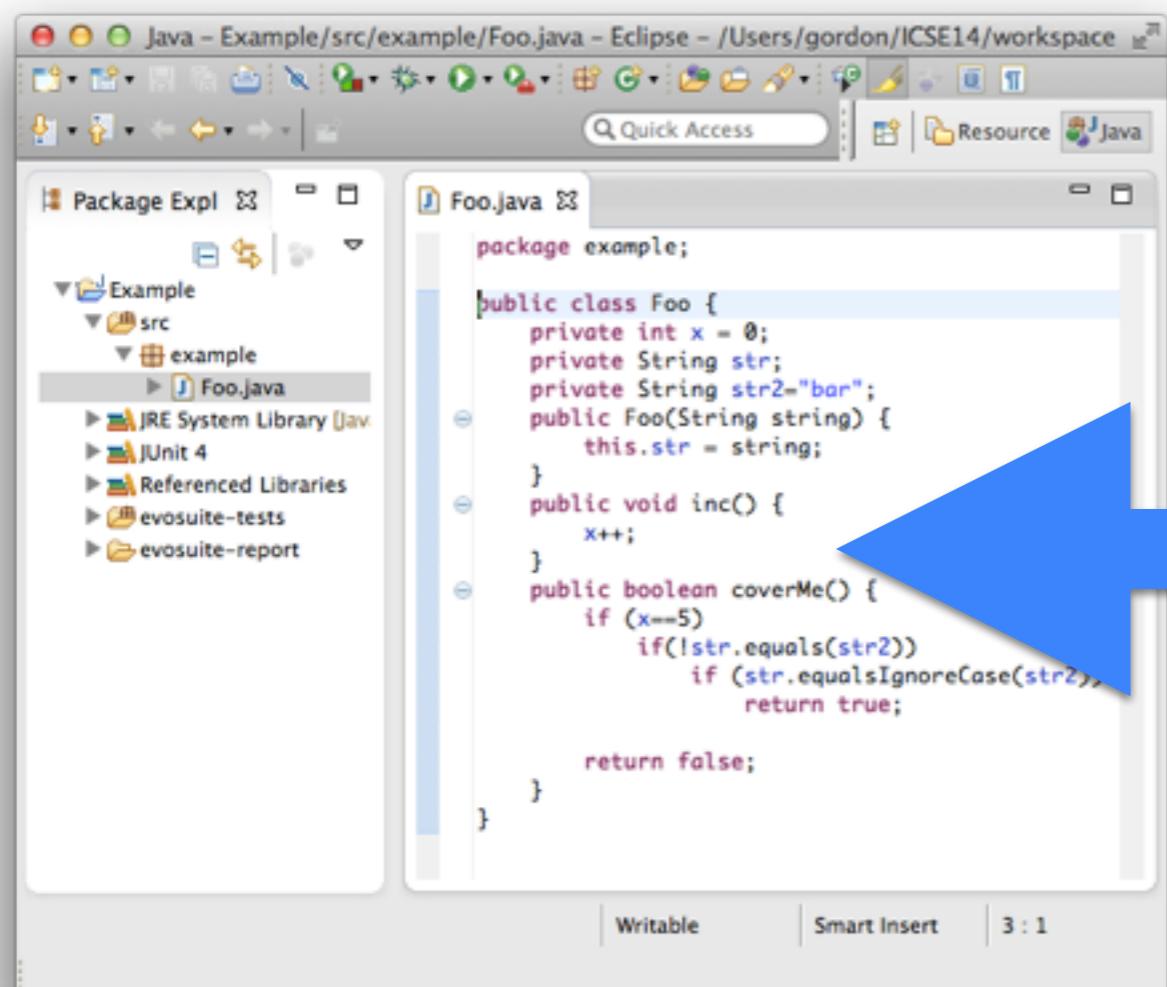
```
public void start() {  
    timer = new Timer();  
    pictures = new Image[10];  
    MediaTracker tracker = new MediaTracker(this);  
    for (int a = 0; a < lastCount; a++)  
        pictures[a] = getImage("image" + a);  
    tracker.addImage(pictures[0]);  
    tracker.waitForCompletion();  
    tracker.checkAll(true);  
}  
  
public void start() {  
    if (timer == null) {  
        timer = new ThreadedImageTimer();  
        timer.start();  
    }  
}  
  
public void paint(Graphics g) {  
    g.drawImage(picture[0]);  
    if (count == lastCount)  
        run();  
}
```



Casos de Test
(Test Suite)

```
@Test  
public void test0() throws Throwable {  
    Foo foo0 = new Foo();  
    Bar bar0 = new Bar("baz3");  
    bar0.coverMe(foo0);  
    assertEquals(0, foo0.getX());  
}
```



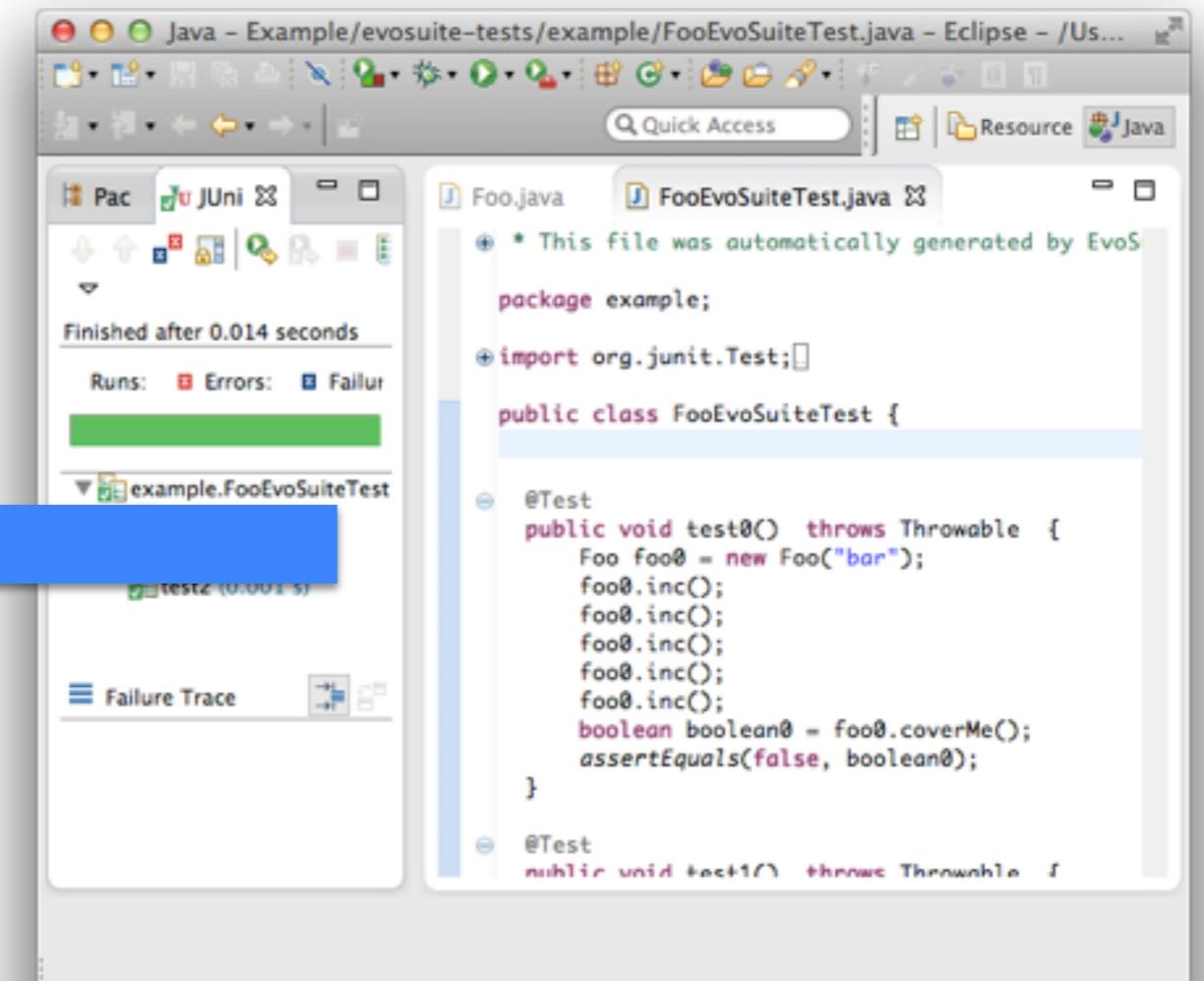


Java - Example/src/example/Foo.java - Eclipse - /Users/gordon/ICSE14/workspace

Package Expl. Foo.java

```
package example;

public class Foo {
    private int x = 0;
    private String str;
    private String str2="bar";
    public Foo(String string) {
        this.str = string;
    }
    public void inc() {
        x++;
    }
    public boolean coverMe() {
        if (x==5)
            if(!str.equals(str2))
                if (str.equalsIgnoreCase(str2))
                    return true;
        return false;
    }
}
```



Java - Example/evosuite-tests/example/FooEvoSuiteTest.java - Eclipse - /Us...

Pac JUni

Finished after 0.014 seconds

Runs: Errors: Failures:

example.FooEvoSuiteTest

```
* This file was automatically generated by EvoSuite

package example;
import org.junit.Test;

public class FooEvoSuiteTest {

    @Test
    public void test0() throws Throwable {
        Foo foo0 = new Foo("bar");
        foo0.inc();
        foo0.inc();
        foo0.inc();
        foo0.inc();
        foo0.inc();
        boolean boolean0 = foo0.coverMe();
        assertEquals(false, boolean0);
    }

    @Test
    public void test1() throws Throwable {
    }
}
```

Source code

Tests

Automated test generation

```
public class Foo {  
    private int x = 0;  
    private String str;  
    private String str2="bar";  
    public Foo(String string) {  
        this.str = string;  
    }  
    public void inc() {  
        x++;  
    }  
    public boolean coverMe() {  
        if (x==5)  
            if(!str.equals(str2))  
                if (str.equalsIgnoreCase(str2))  
                    return true;  
  
        return false;  
    }  
}
```

Java - Example/src/example/Foo.java - Eclipse - /Users/gordon/ICSE14/workspace

Quick Access | Resource Java

Package Explorer JUnit

Example

src

example

Foo.java

GenericParameter.java

Stack.java

JRE System Library [JavaSE-1.6]

JUnit 4

Referenced Libraries

Foo.java

```
package example;

public class Foo {
    private int x = 0;
    private String str;
    private String str2="bar";
    public Foo(String string) {
        this.str = string;
    }
    public void inc() {
        x++;
    }
    public boolean coverMe() {
        if (x==5)
            if(!str.equals(str2))
                if (str.equalsIgnoreCase(str2))
                    return true;
        return false;
    }
}
```

example.Foo.java - Example/src

Java - Example/test/randoop/RandoopTest0.java - Eclipse - /Users/gordon/ICSE14/workspace

Quick Access Resource Java

Package Explorer JUnit

Example src example Foo.java GenericParameter.java Stack.java JRE System Library [JavaSE-1.6] JUnit 4 Referenced Libraries test randoop RandoopTest.java RandoopTest0.java

Foo.java RandoopTest.java RandoopTest0.java

```
example.Foo var1 = new example.Foo("hi!");
boolean var2 = var1.coverMe();
var1.inc();
var1.inc();
var1.inc();
boolean var6 = var1.coverMe();
var1.inc();
boolean var8 = var1.coverMe();
var1.inc();
var1.inc();
boolean var11 = var1.coverMe();
boolean var12 = var1.coverMe();
var1.inc();

// Regression assertion (captures the current behavior of the code)
assertTrue(var2 == false);
```

Randoop

Tests generated: 1389 Failures: 0

Writable Smart Insert 1 : 1

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer (left):** Shows the project structure under "Example". It includes a "src" folder containing "example" with files "Foo.java", "GenericParameter.java", and "Stack.java". It also contains "JRE System Library [JavaSE-1.6]", "JUnit 4", "Referenced Libraries", and a "test" folder with "randoop" containing "RandoopTest.java" and "RandoopTest0.java".
- Editor Area (center):** Displays the content of "RandoopTest.java". The code creates a "Foo" object, calls its "coverMe()" method, and then repeatedly calls "inc()". It then checks the value of "var2" using a regression assertion.
- Randoop View (bottom):** Shows the results of the Randoop analysis. It indicates 1389 tests generated and 0 failures. A green progress bar at the bottom of the view is nearly full.

Generación Automática de Casos de Test

Primer Cuatrimestre 2018

Juan P. Galeotti
Universidad de Buenos Aires

Temario del Curso

1. Fundamentos Básicos
2. Mutation Testing
3. Random Testing
4. Concolic Execution
5. Search-Based Testing
6. Exhaustive Testing
7. Fuzzing / Event-Driven Testing

La Materia

- Una Clase por semana (Lunes 10am-3pm)
- (Casi) cada clase una nueva Práctica de Laboratorio auto-evaluable
- 2 Trabajos Prácticos
- 1 Parcial (escrito) + Final (oral)

La Materia

- <https://campus.exactas.uba.ar/course/view.php?id=1033>
- Aula 13 - Pabellón 1
- Laboratorio 3 - Pabellón 1

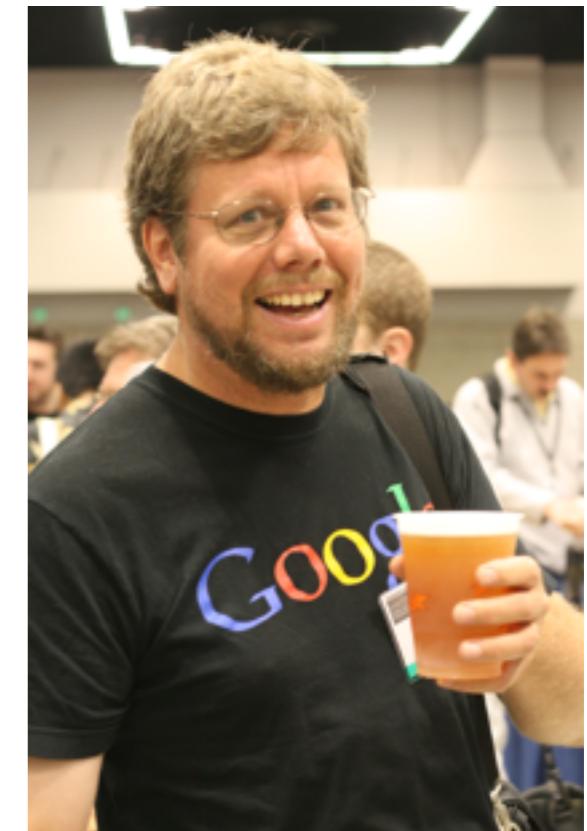


- Lenguaje de los Talleres y las Prácticas
- creador: James Gosling Sun Microsystems / Oracle
- OO / Interpretado





- Pseudo-código
- Fácil de leer, Fácil de entender
- Interactivo e Interpretado
- Especialmente útil para análisis dinámicos



Guido van Rossum,
Inventor de Python

About Me



www.dc.uba.ar/~jgaleotti
Oficina #13
(Entrepiso)

- *Doctorado:* Verificación de Software (Alloy/Java) en UBA
- *Post-doc:* Verificación, Inferencia y Testing en Universität Saarlandes (Alemania)
- CONICET+UBA: Ingeniería del Software (Tool EvoSuite)

Automatizando la Ejecución de los Tests

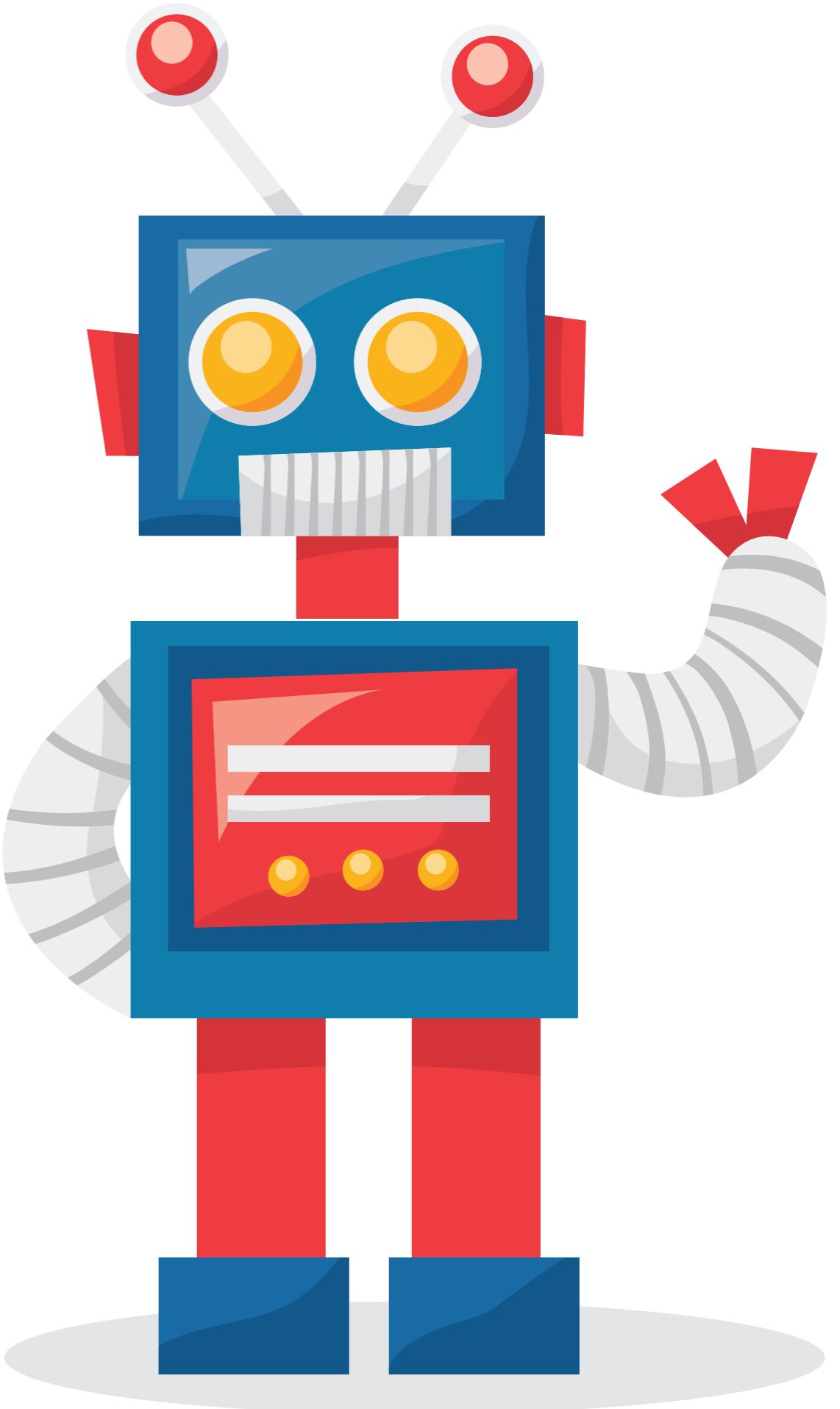


Software Moderno

Features

Nuestra capacidad de
Testear las Features

Generación de Tests



Tests por Niveles

Un test de unidad (unit test) consiste de:

1. **Test de Sistema:** probar el resultado de usar un sistema. Todo el equipo.
2. **Test de Integración:** probar funcionamiento entre unidades/módulos y programadores
3. **Test de Unidad:** chequear comportamiento de una unidad. Mocks. Único programador.

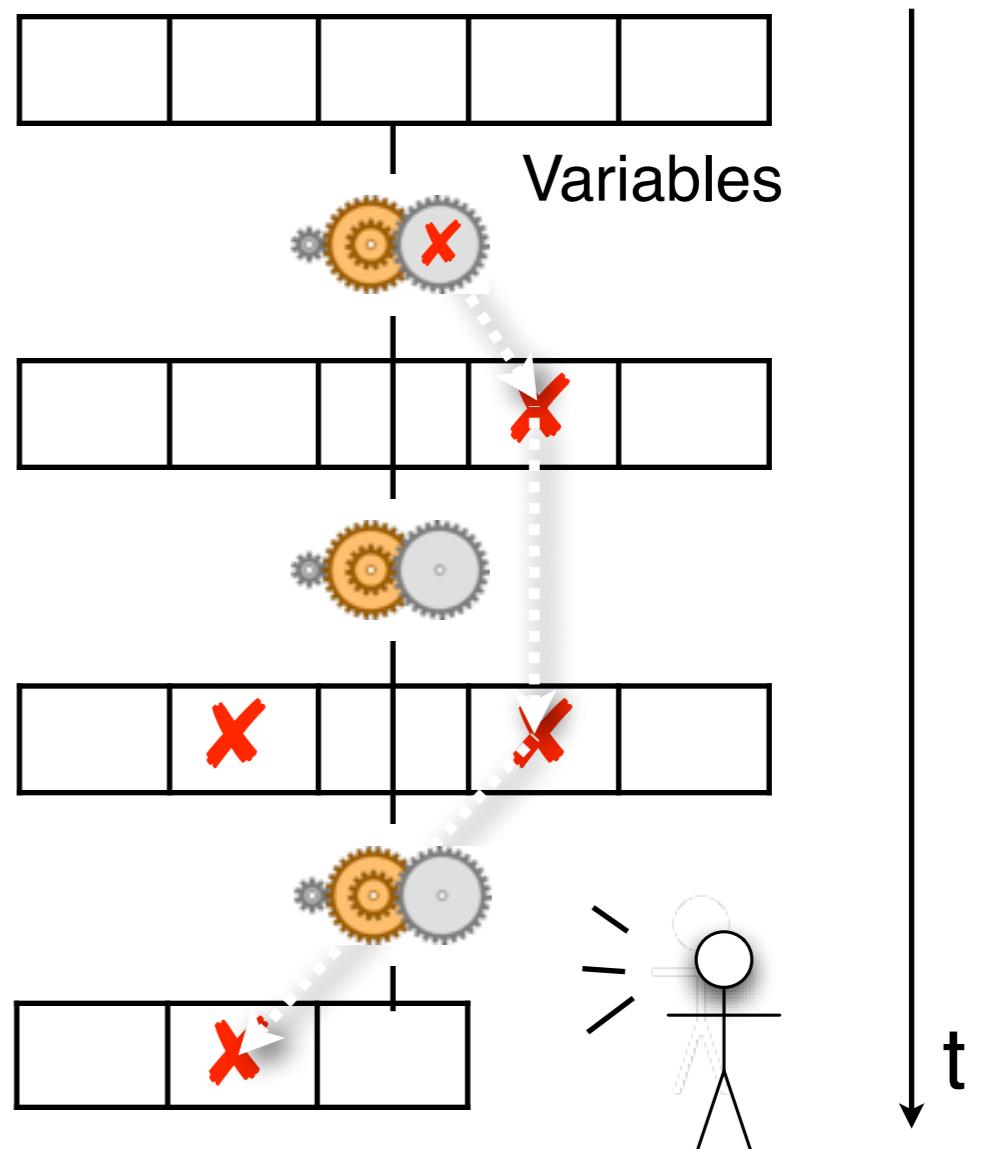
Unit Tests

Un test de unidad (unit test) consiste de:

1. el ***setup*** – una etapa de preparación requerida antes de ejercitar el test
2. la ***ejercitación*** – el código que ejercita la funcionalidad bajo test
3. el ***chequeo*** – código que chequea la respuesta contra el resultado esperado (oráculo)

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**.

El error



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

- Una excepción de hardware paralizó el sistema de inercia referencial así como el sistema backup (usaban el mismo código), emitiendo información inválida.
 - Esa información inválida fue interpretada por el navegador como **información válida (no estaba sanitizada)**
 - El Autopilot asumió que se estaba desviando del curso original, al intentar corregir el cohete salió del curso previsto y se activó la autodestrucción
- *La infección*
- *La falla*

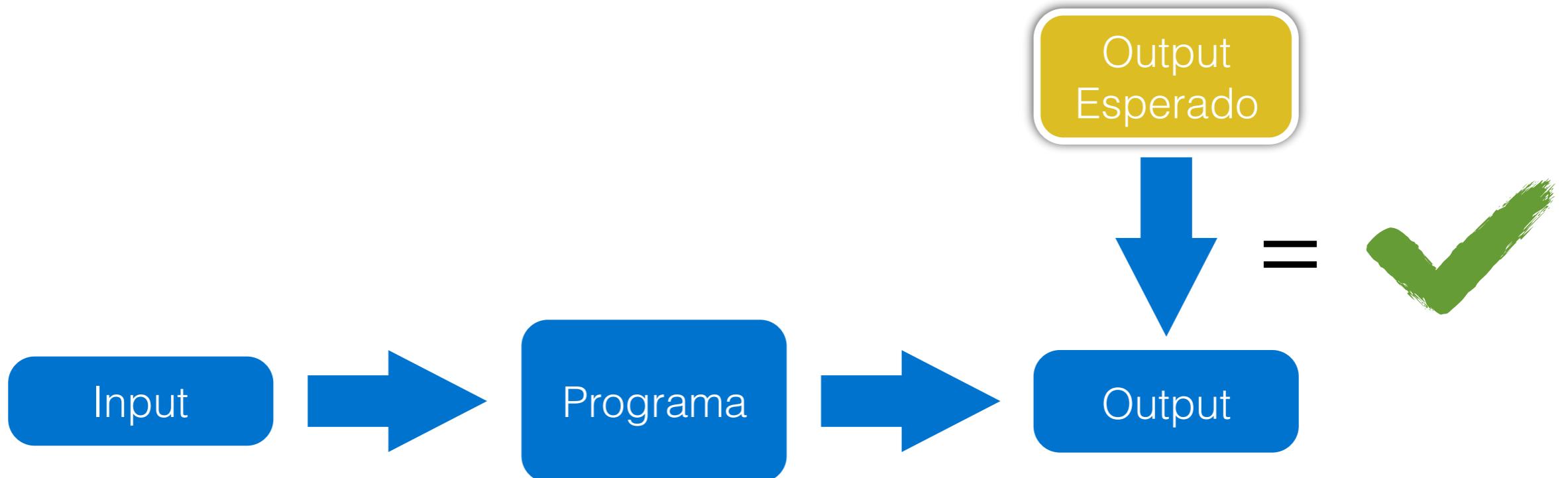
Testing

- **Problema:** Una propiedad o comportamiento del programa cuestionable.
- **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

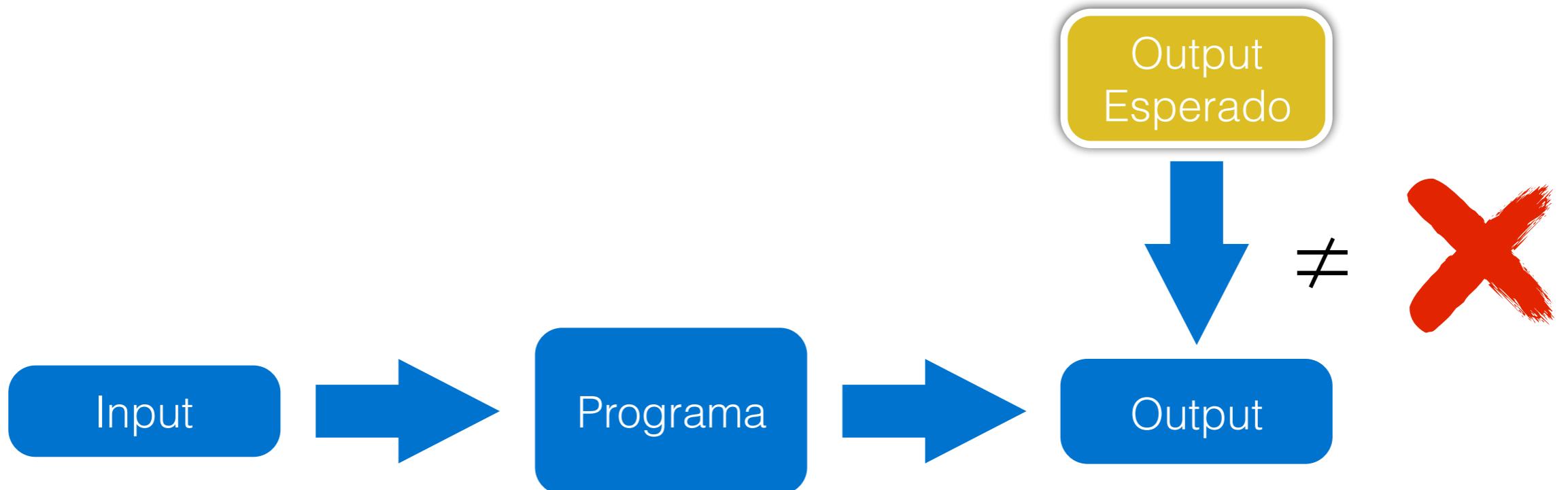
- Demonstración
 - mostrar que satisface la spec
- Destrucción
 - tratar de hacerlo fallar
- Evaluación
 - descubrir si y cómo falla
- Prevención
 - evitar futuras fallas

UNIT TESTING



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

UNIT TESTING

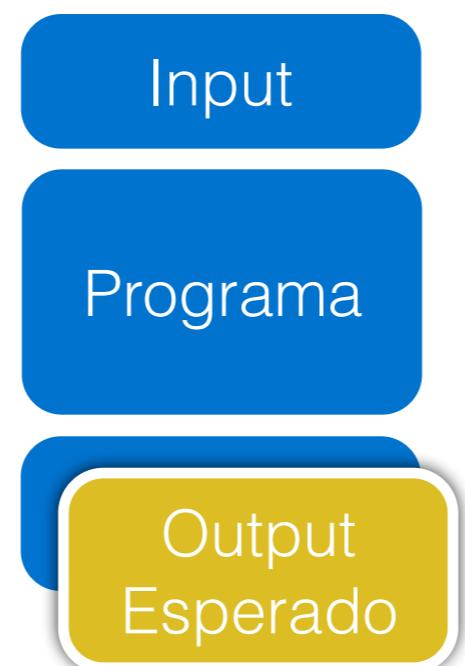


1. Ejecutar el programa
Usando datos de test
2. Chequear el output del programa
Usando el oráculo de test

JUNIT TESTING

@Test

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



```
}
```

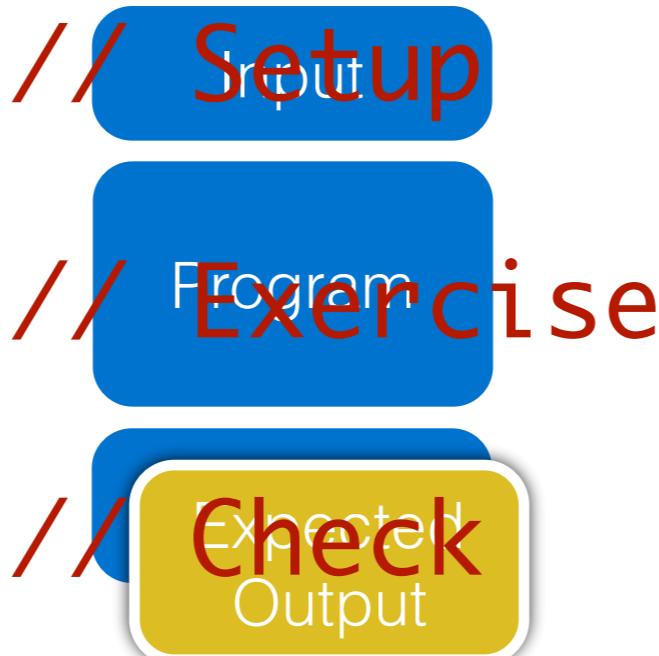
JUnit

<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/>

ANOTACIONES JUNIT

@Test (timeout=100)

Un test case (posiblemente con un timeout)

@Ignore ("Reason")

No ejecutar este test case

@Before

Ejecutar antes de cada test case

@After

Ejecutar luego de cada test case

@BeforeClass

Ejecutar una única vez antes de todos los test cases

@AfterClass

Ejecutar una única vez luego de todos los test cases

ASERCIIONES JUNIT

`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);`

ORGANIZANDO TESTS

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

TESTING DE EXCEPCIONES

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

TESTING DE EXCEPCIONES

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

DEMO



Java



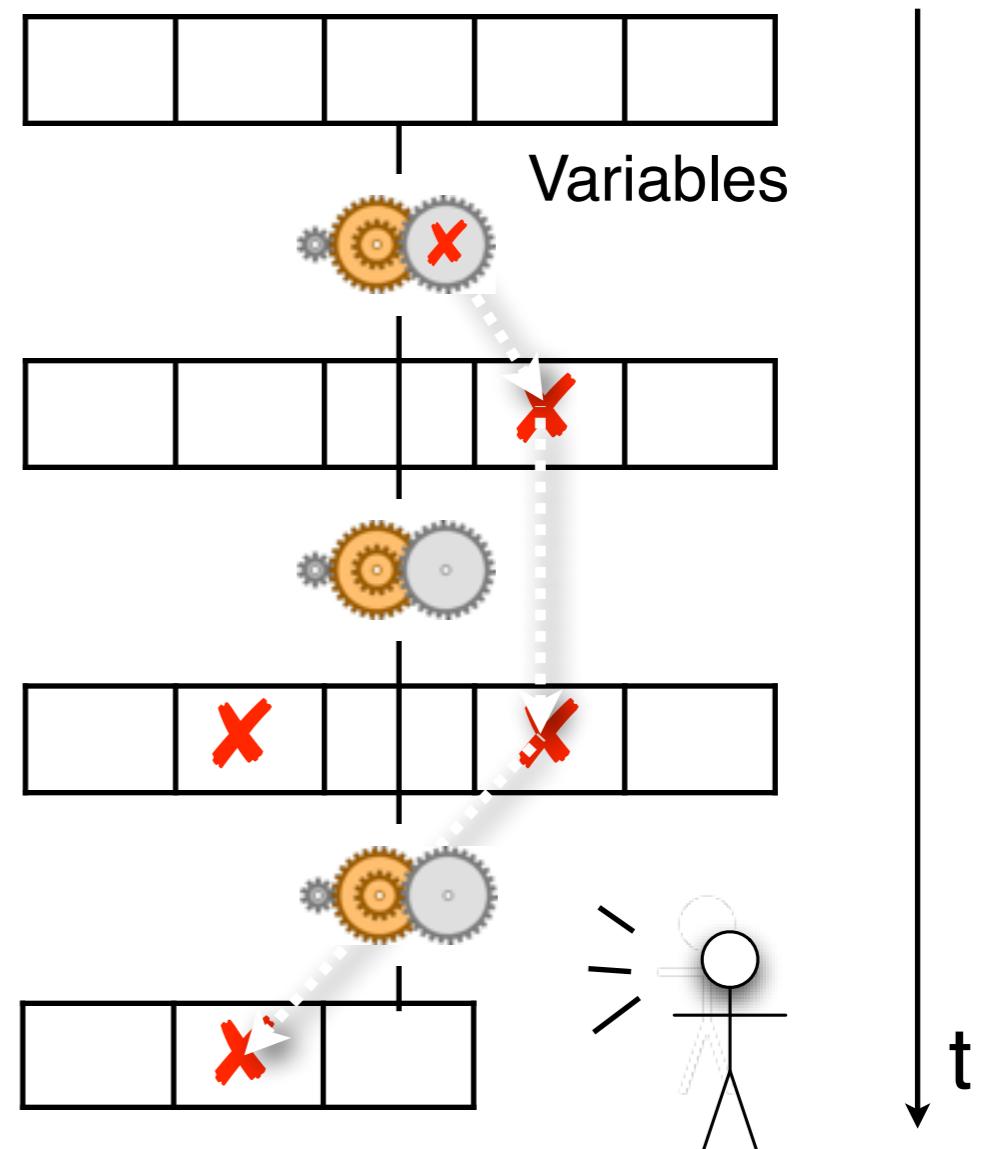
Eclipse IDE
<http://www.eclipse.org/>

JUnit

JUnit Framework

Los 3 Desafíos del Testing

1. 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*.

2. Debemos *reconocer* el error como tal

- como una desviación de lo que es correcto, válido, o verdadero.

3. Debemos identificar *funcionalidad faltante*

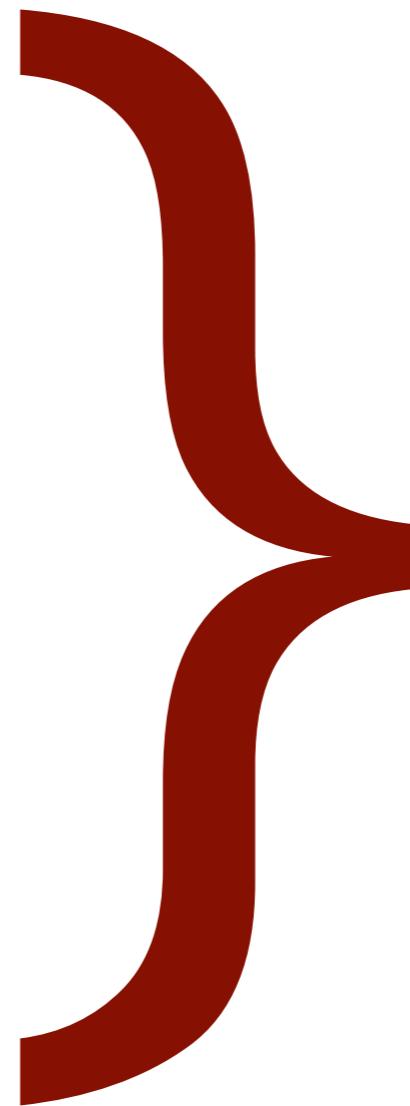
– *Cubrir tanto comportamiento como sea posible*

– *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*.
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*

– 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*.

debemos alcanzar cada línea del programa

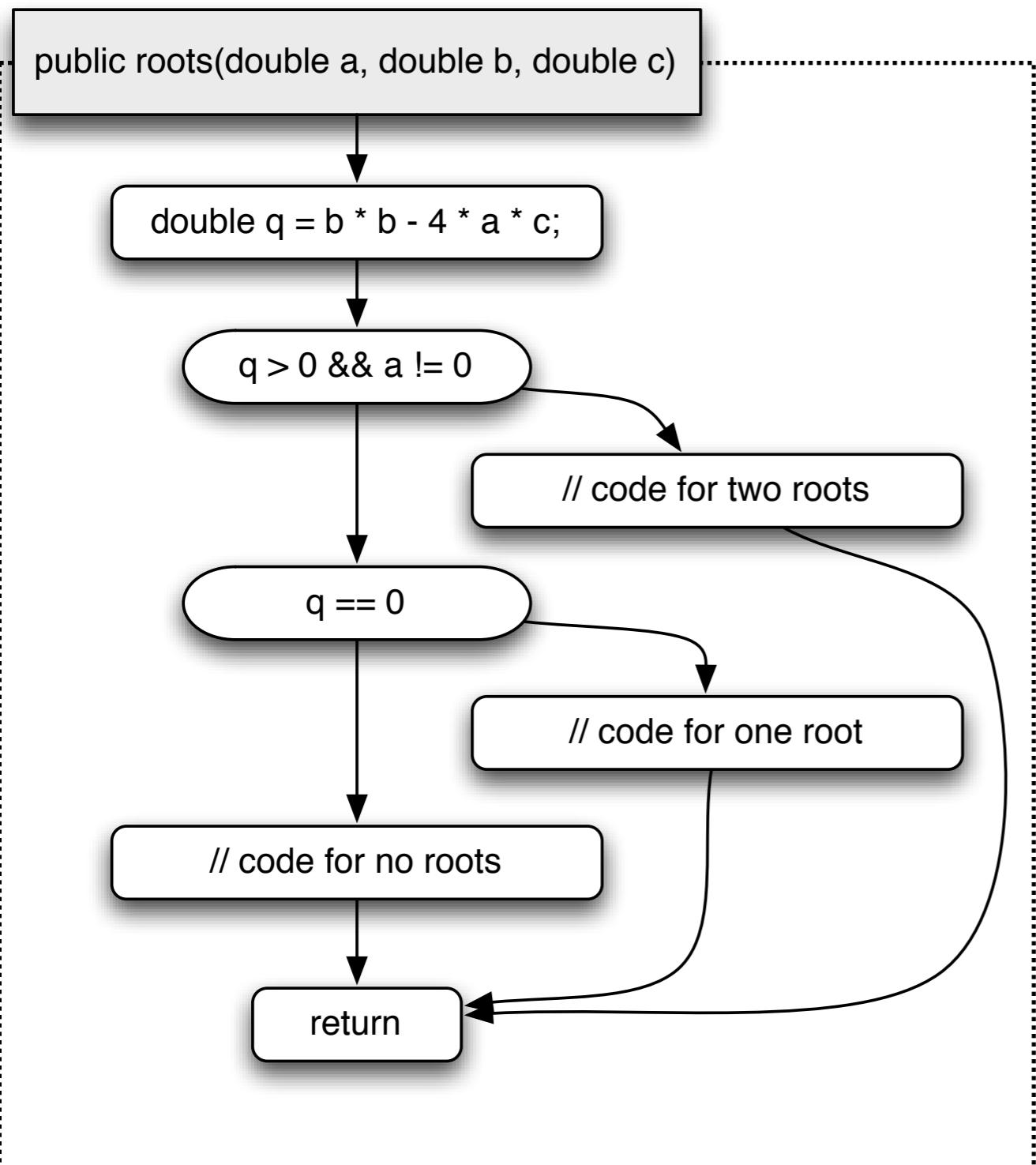
Test Cases vs. Test Suites

- **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

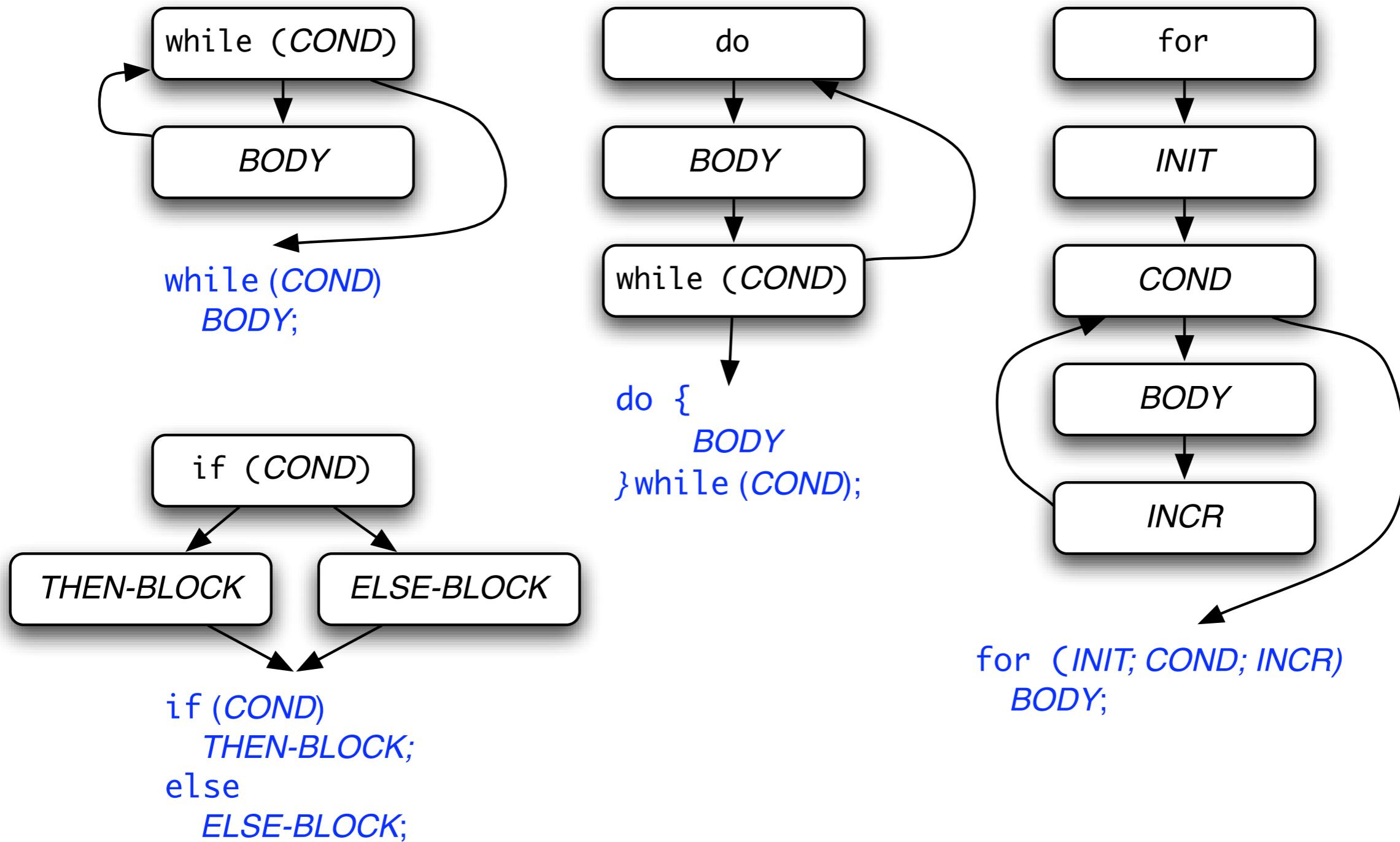
- ¿Cómo sabemos que una test suite es “suficientemente buena”?
- Un criterio de adecuación de test es un predicado que es verdadero o falso para un par $\langle \text{programa}, \text{test suite} \rangle$
- Usualmente expresado en forma de una regla – e.g., “todos los statements deben ser ejecutados”

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

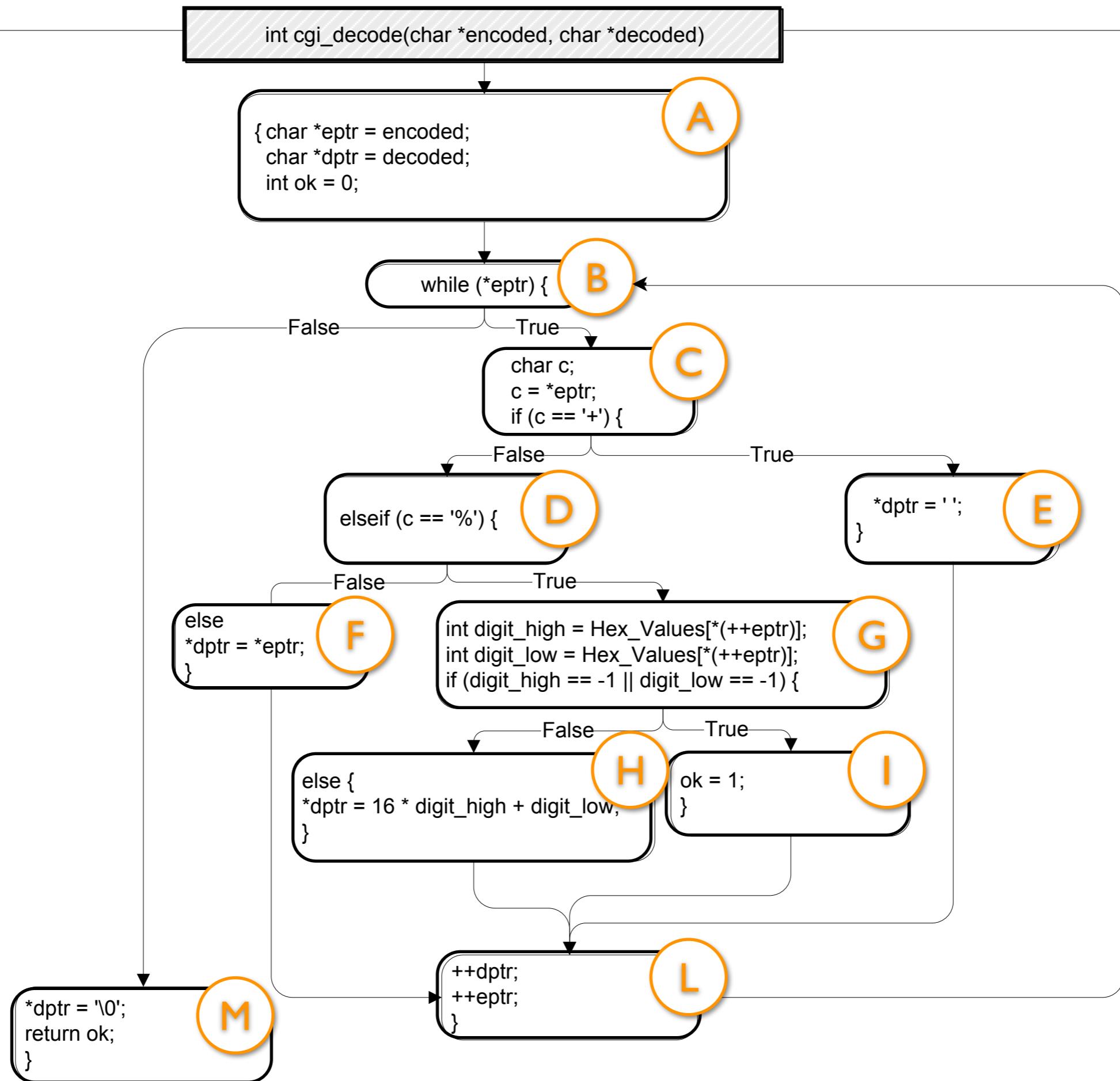
- Criterio de Adecuación: cada statement (o nodo en el CFG) *debe ser ejecutado al menos una vez*
- Idea: un defecto en un statement sólo puede ser revelado ejecutando el defecto
- Cobertura:
$$\frac{\text{\# statements ejecutados}}{\text{\# statements}}$$

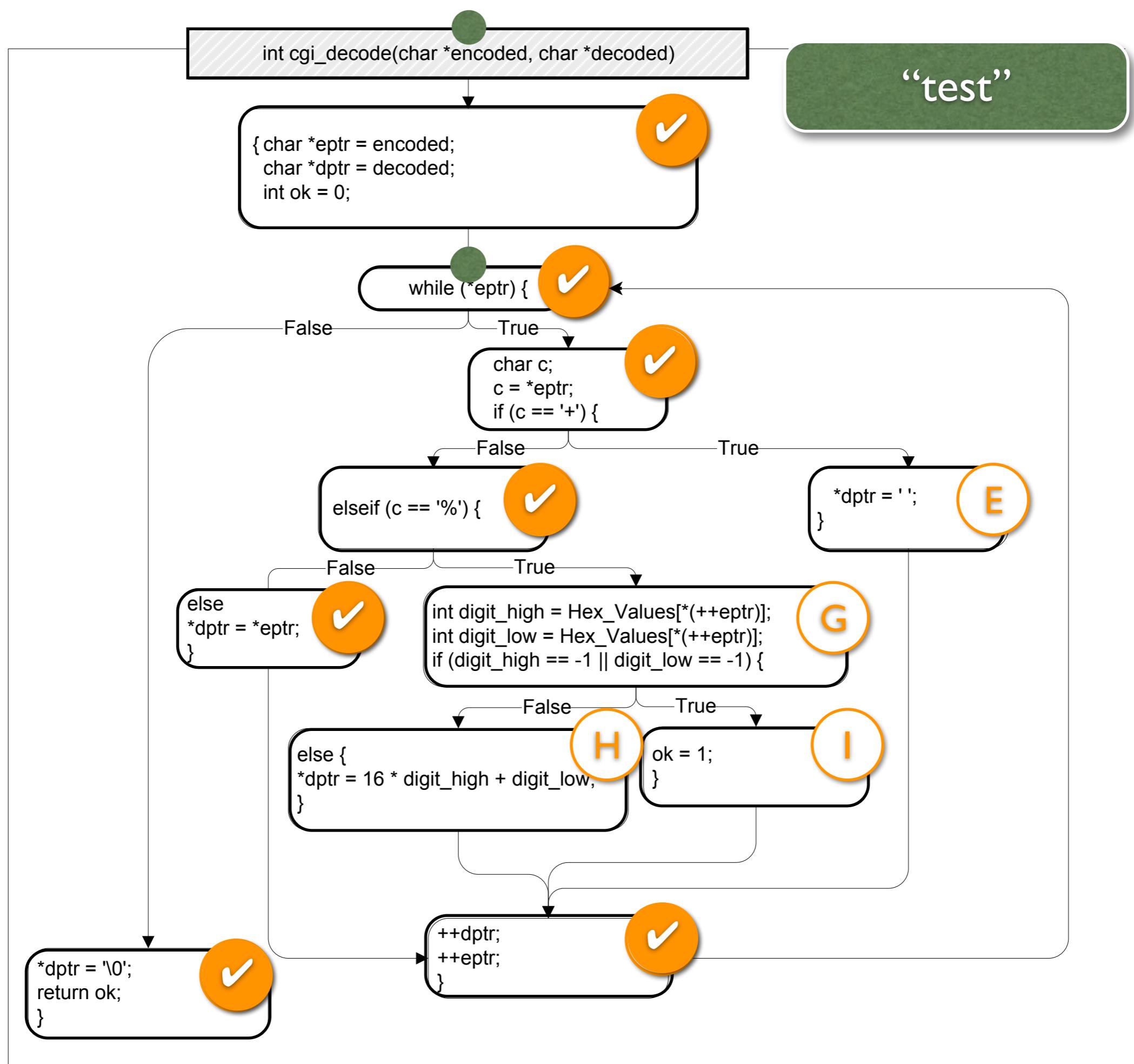
Ejemplo: cgi_decode

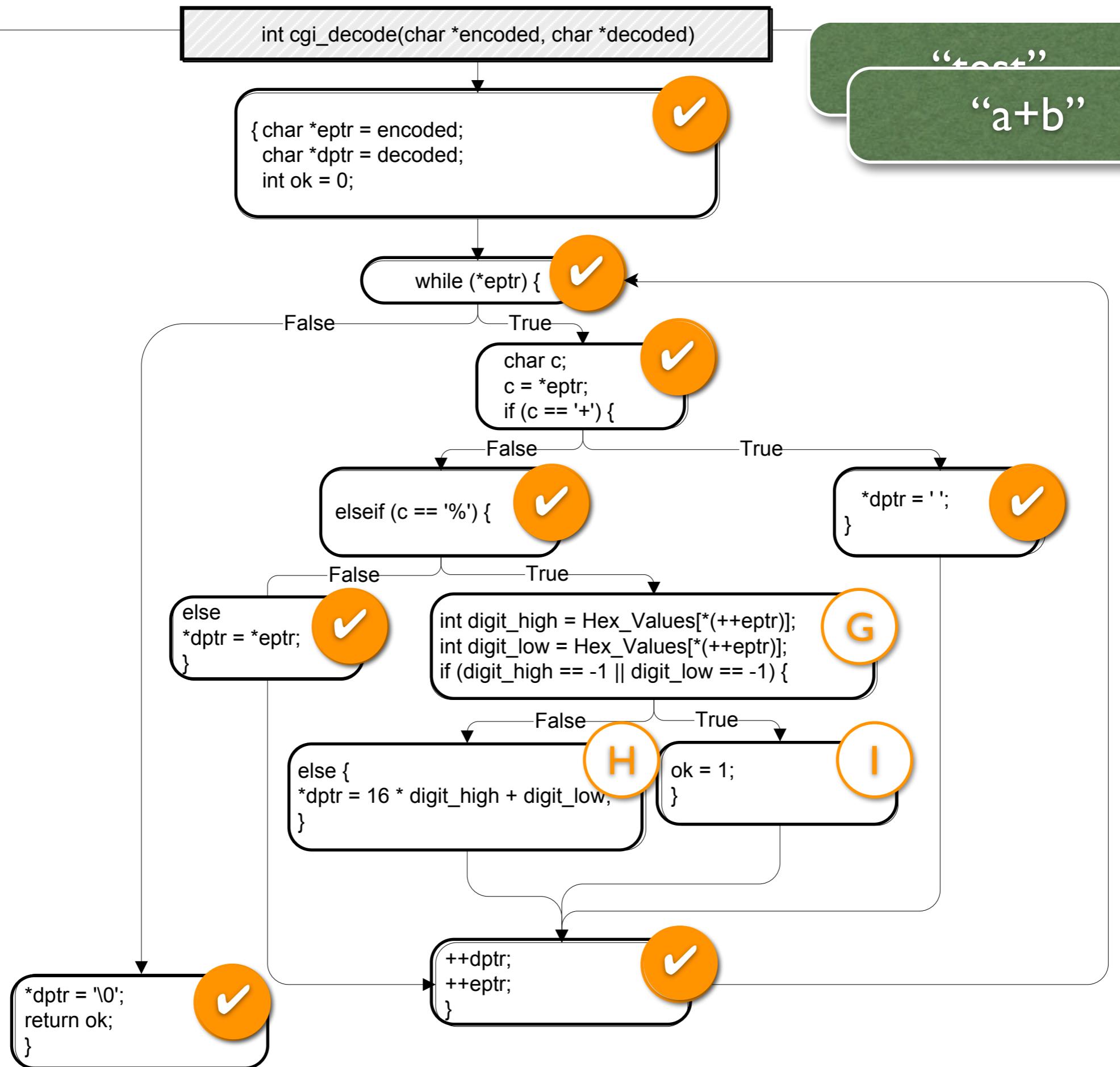
```
/**  
 * @title cgi_decode  
 * @desc  
 * Translate a string from the CGI encoding to plain ascii text  
 * '+' becomes space, %xx becomes byte with hex value xx,  
 * other alphanumeric characters map to themselves  
 *  
 * returns 0 for success, positive for erroneous input  
 * 1 = bad hexadecimal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

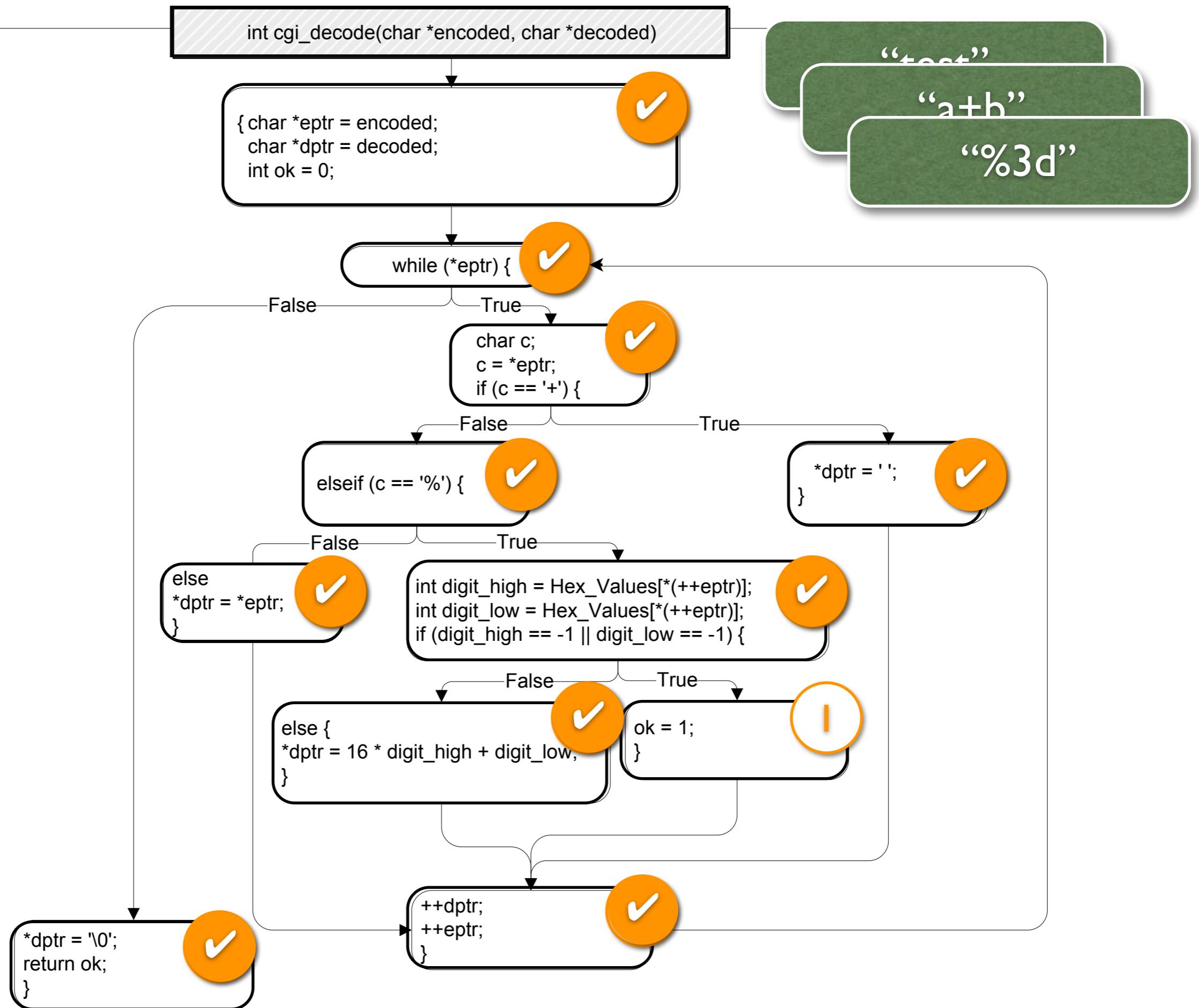
```
while (*eptr) /* loop to end of string ('\0' character) */ B
{
    char c; C
    c = *eptr;
    if (c == '+') { /* '+' maps to blank */
        *dptr = ' '; E
    } else if (c == '%') { /* '%xx' is hex for char xx */ D
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];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

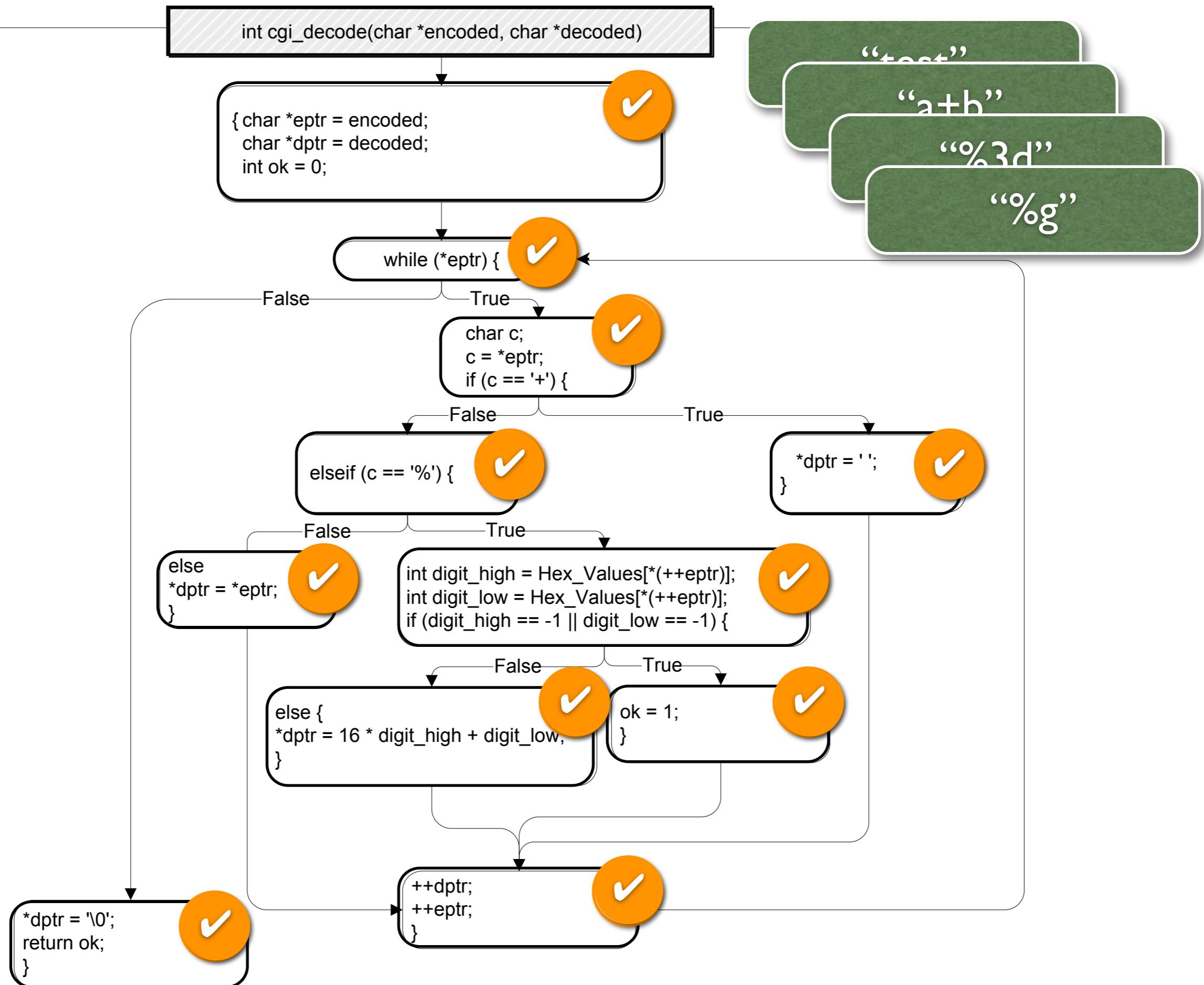
*dptr = '\0'; /* Null terminator for string */ M
return ok;
}
```





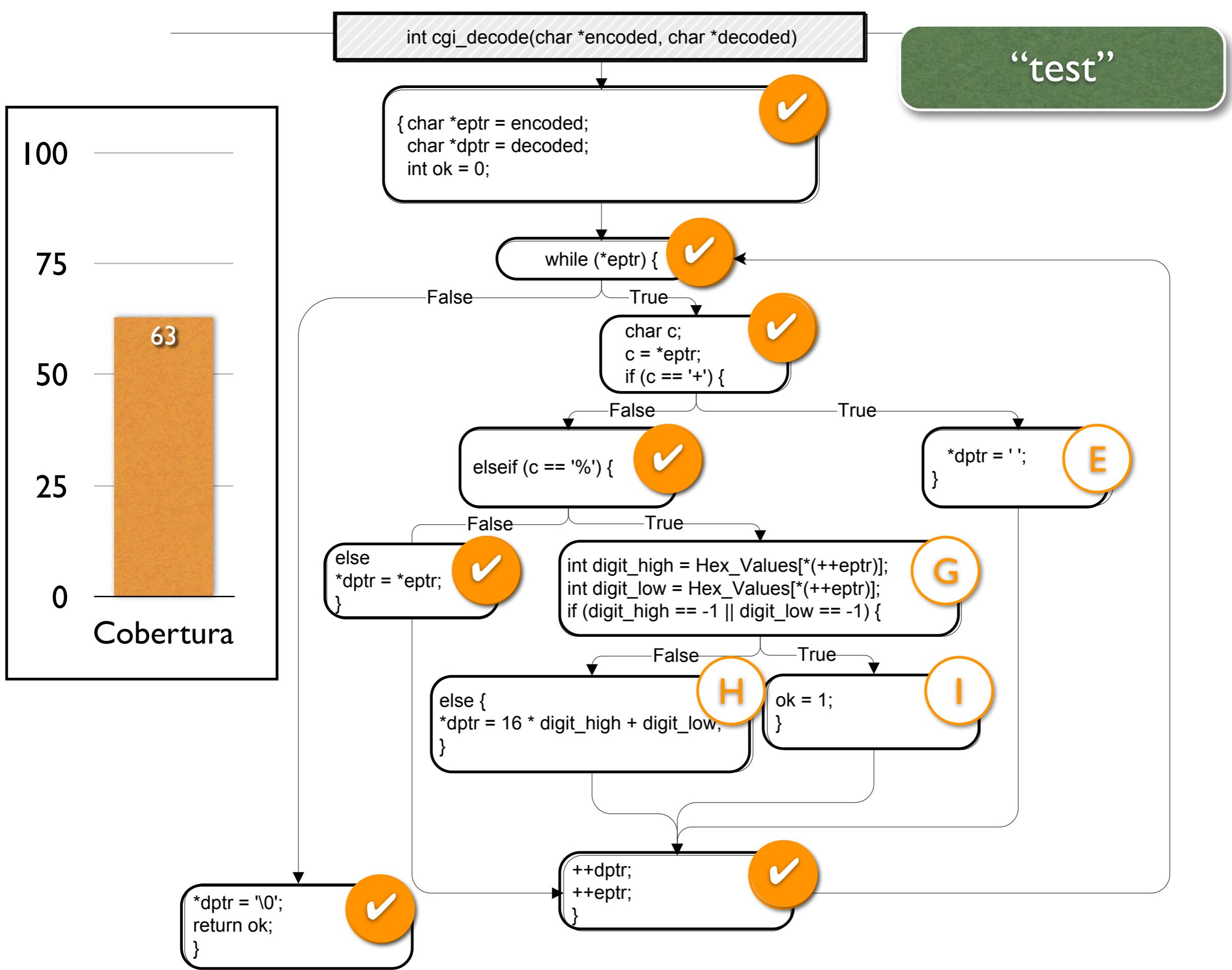






Statement Testing

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



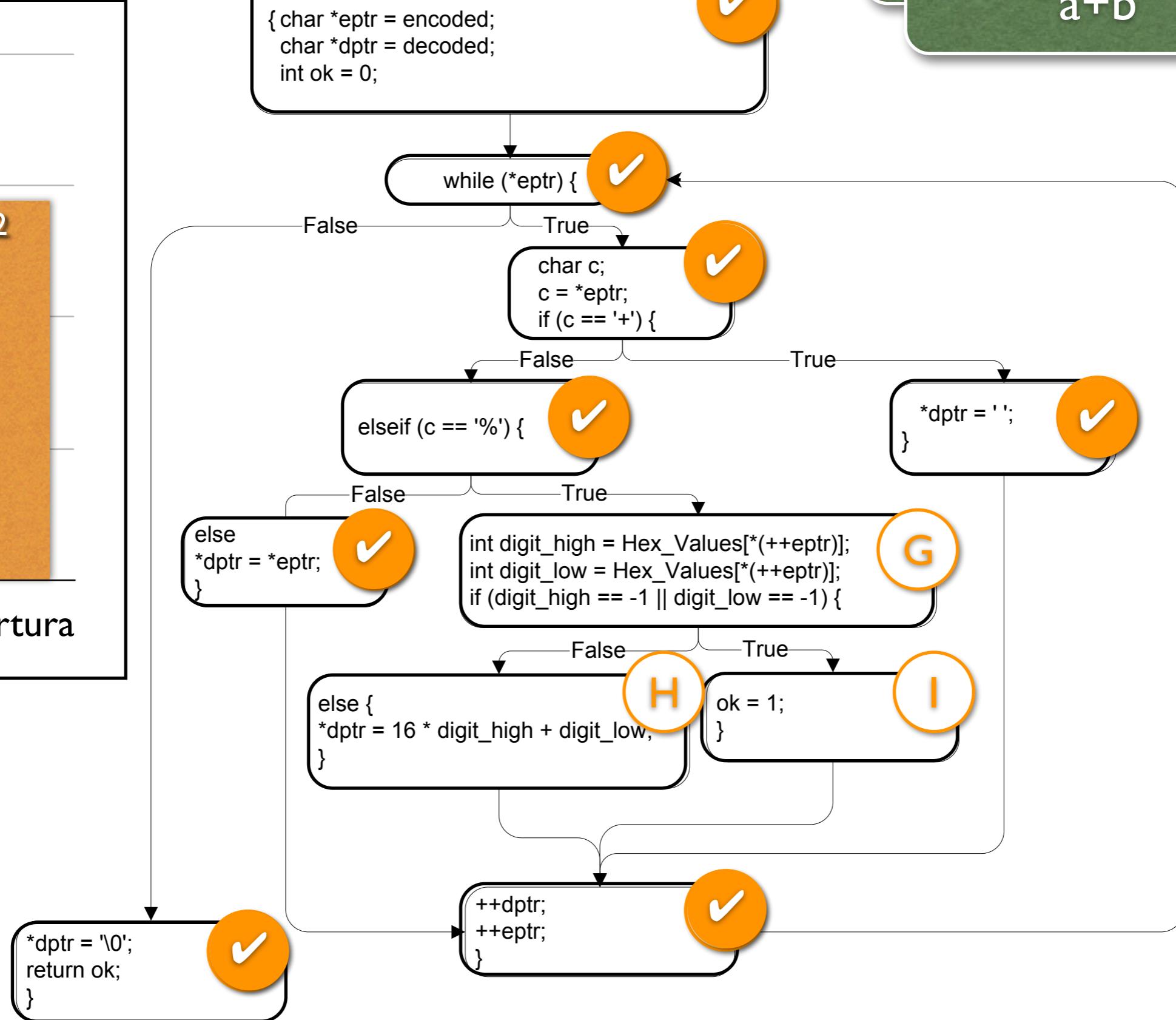
100

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,
}
```

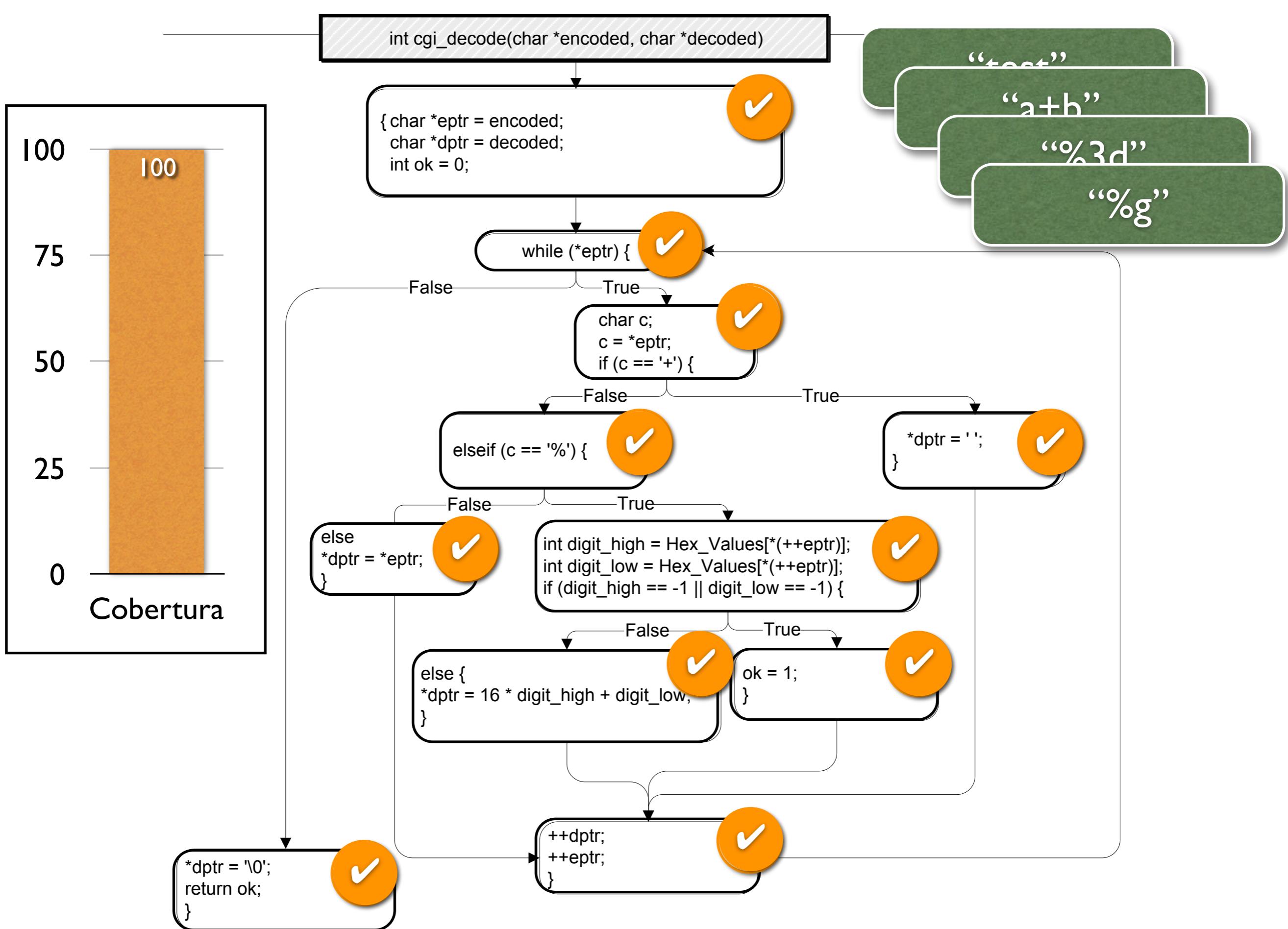
`ok = 1;`

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



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

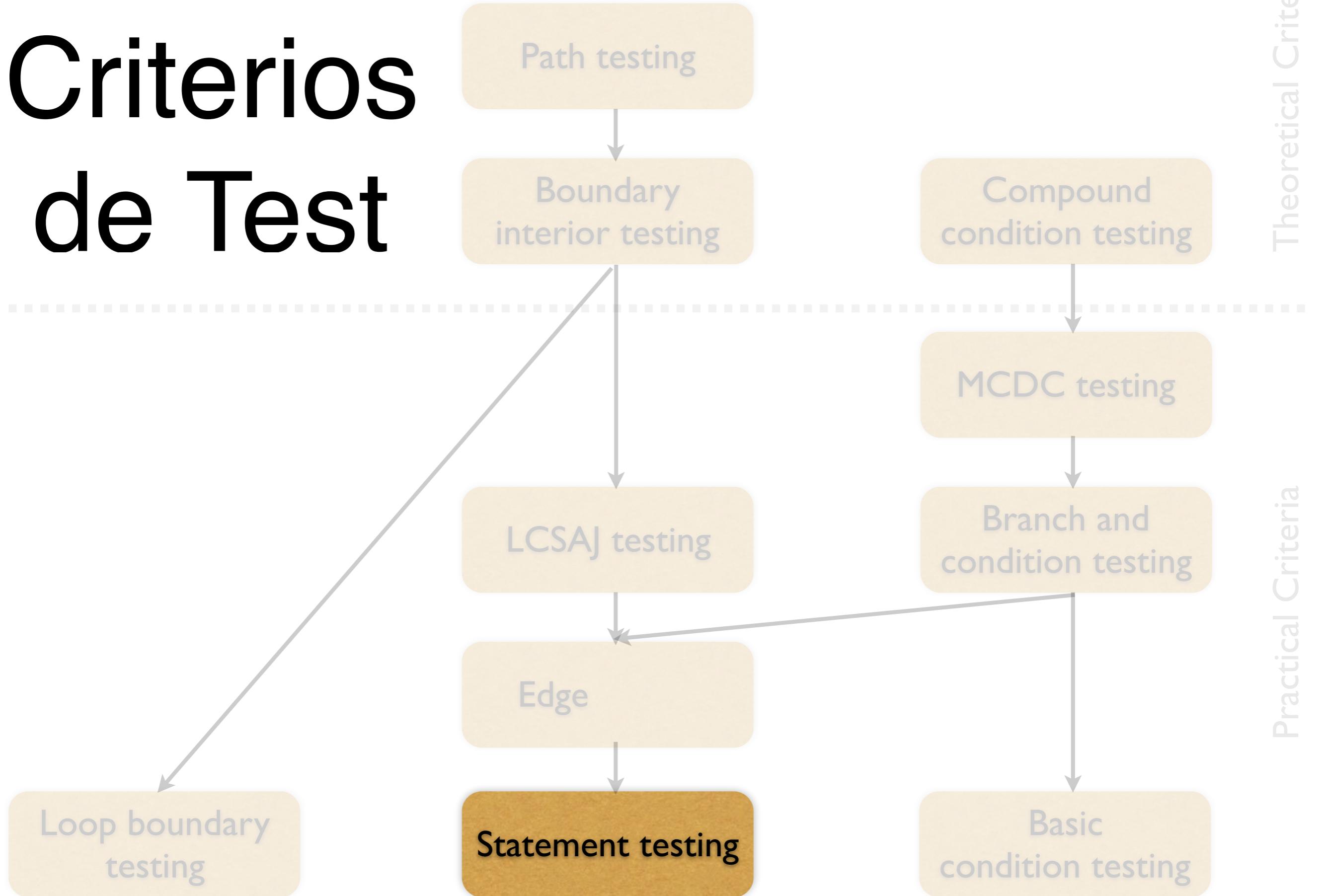




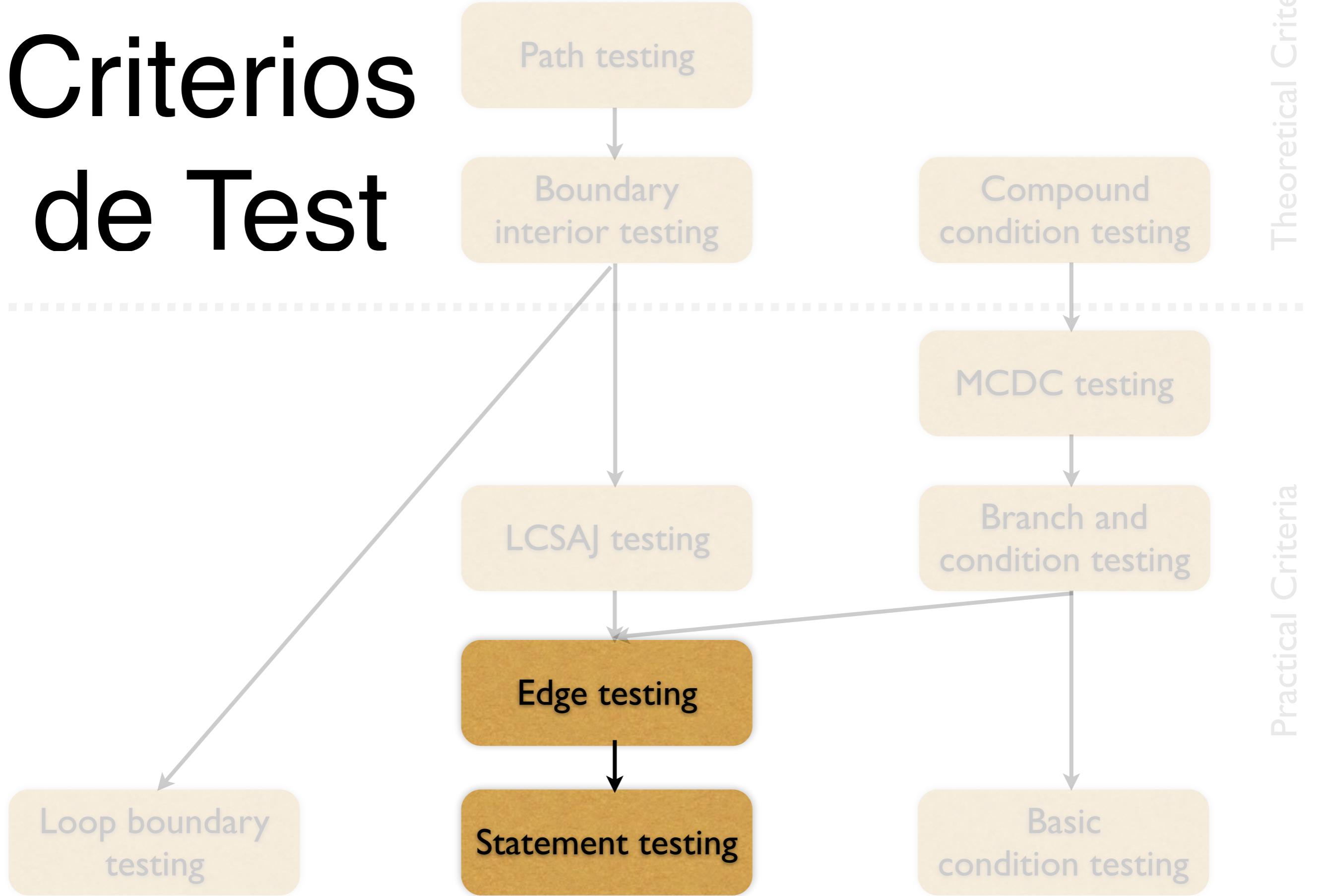
Calculando la Cobertura

- 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

Criterios de Test



Criterios de Test

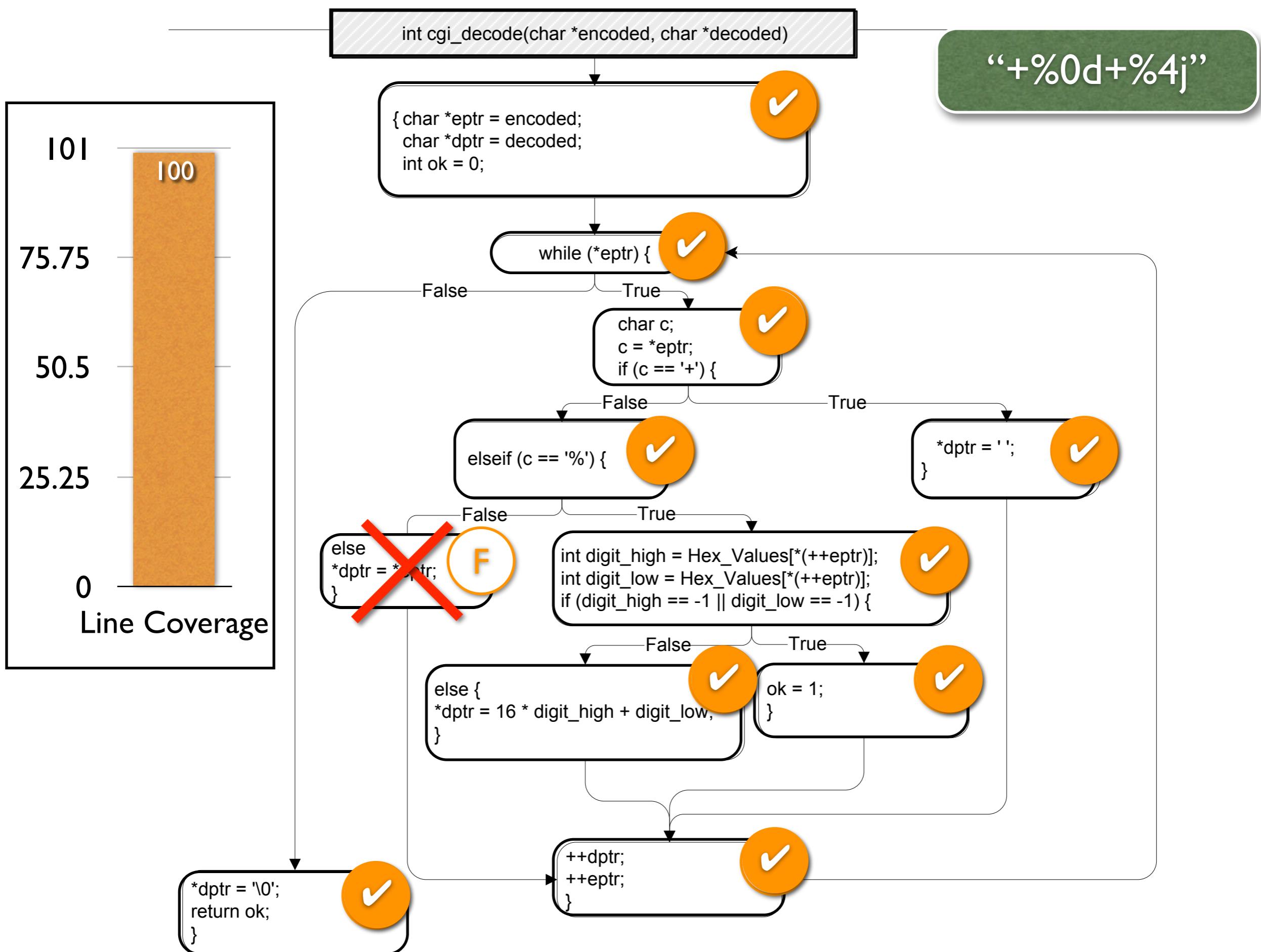


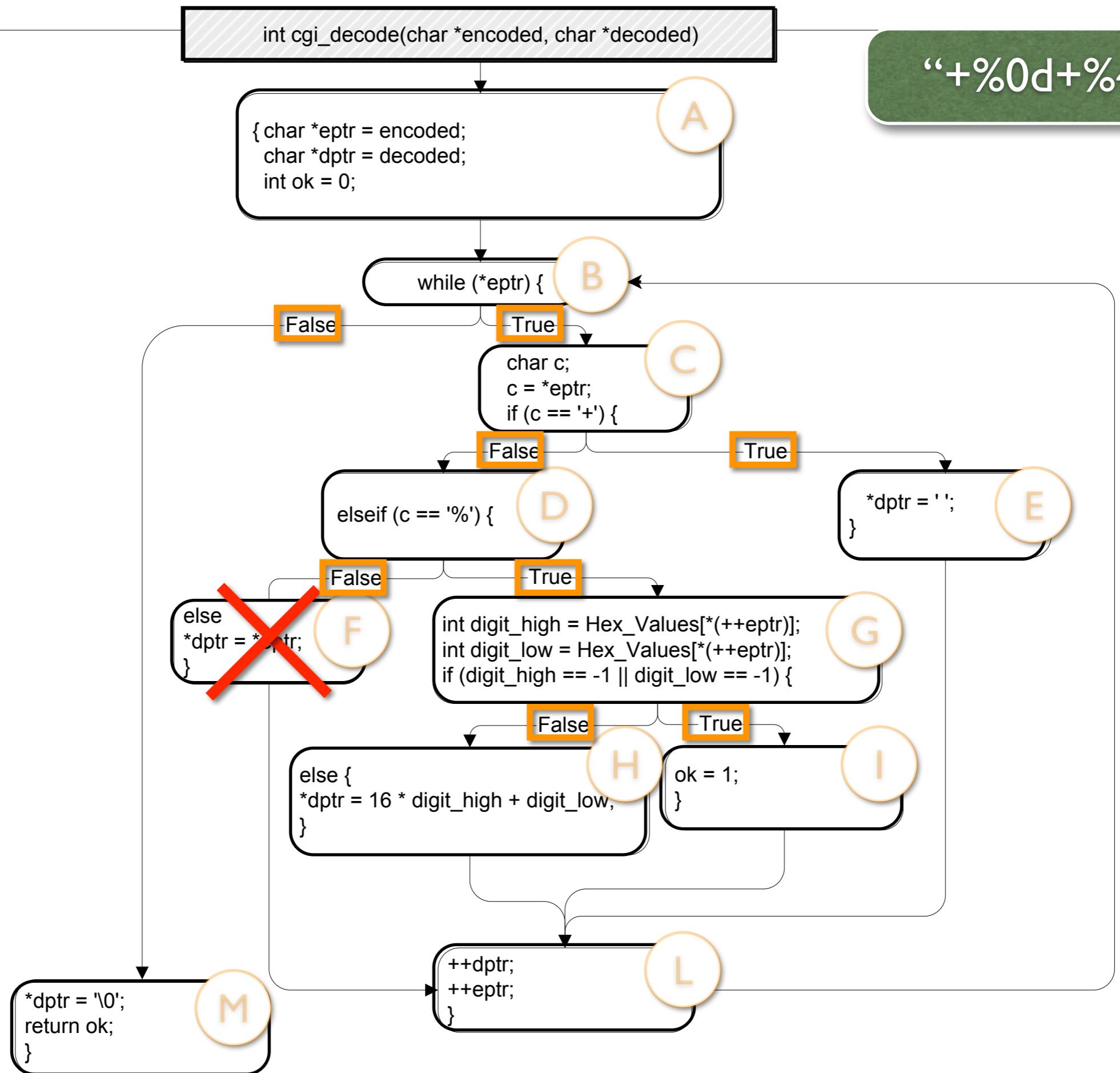
Edge Testing

- Criterio de adecuación: cada arco en el CFG debe ser ejecutado al menos una vez
- Cobertura :
$$\frac{\text{\# arcos ejecutados}}{\text{\# arcos}}$$
- Subsume Statement Testing
 - ya que al recorrer todos los arcos recorremos todos los nodos

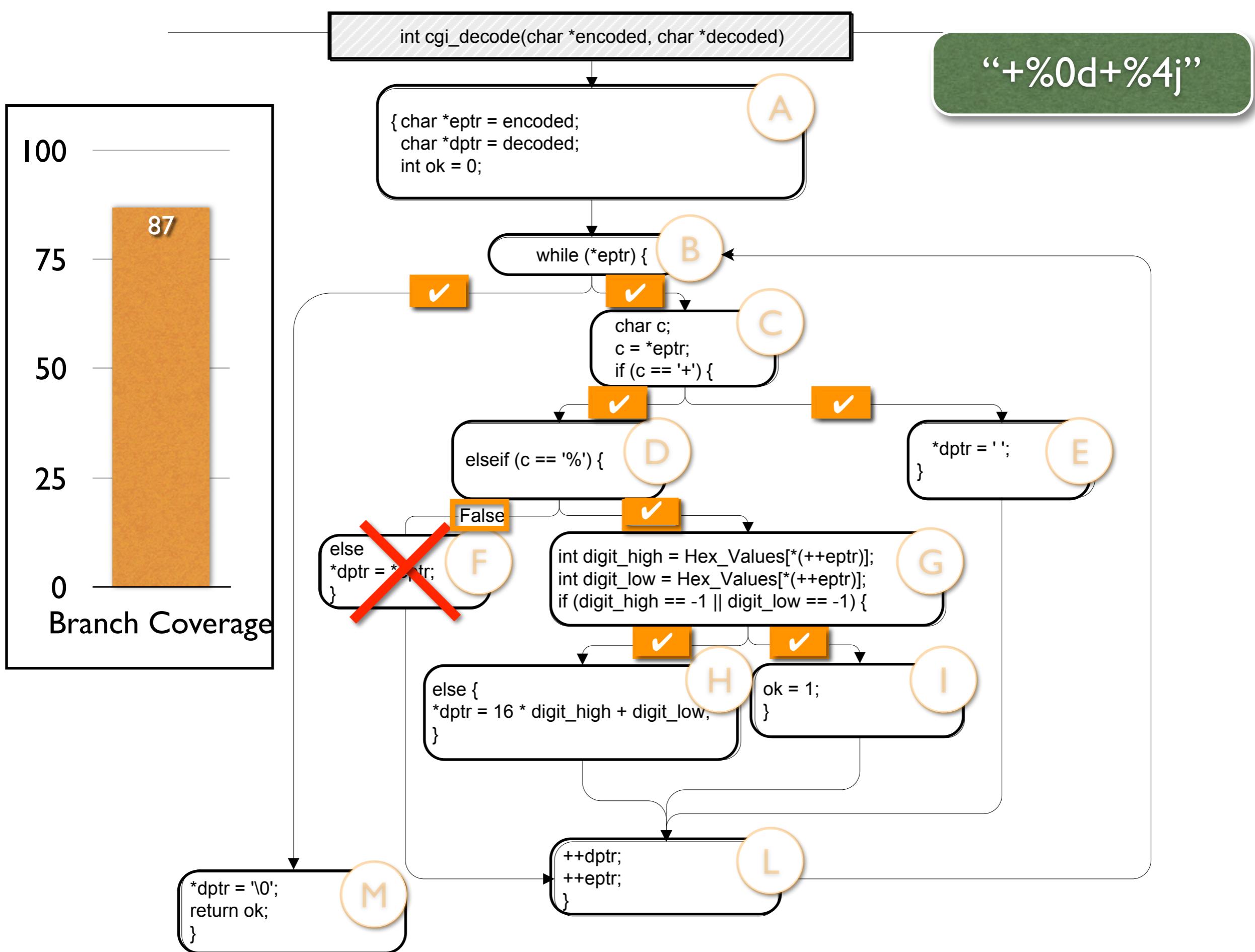
Branch Testing

- Criterio de adecuación: cada branch en el CFG debe ser ejecutado al menos una vez
- Cobertura :
$$\frac{\text{\# branches ejecutados}}{\text{\# branches}}$$
- No Subsume Statement Testing ni Edge Testing puedo ejercitar todos los branches y no cubrir todos los arcos/nodos
- Mas comúnmente usado en la Industria

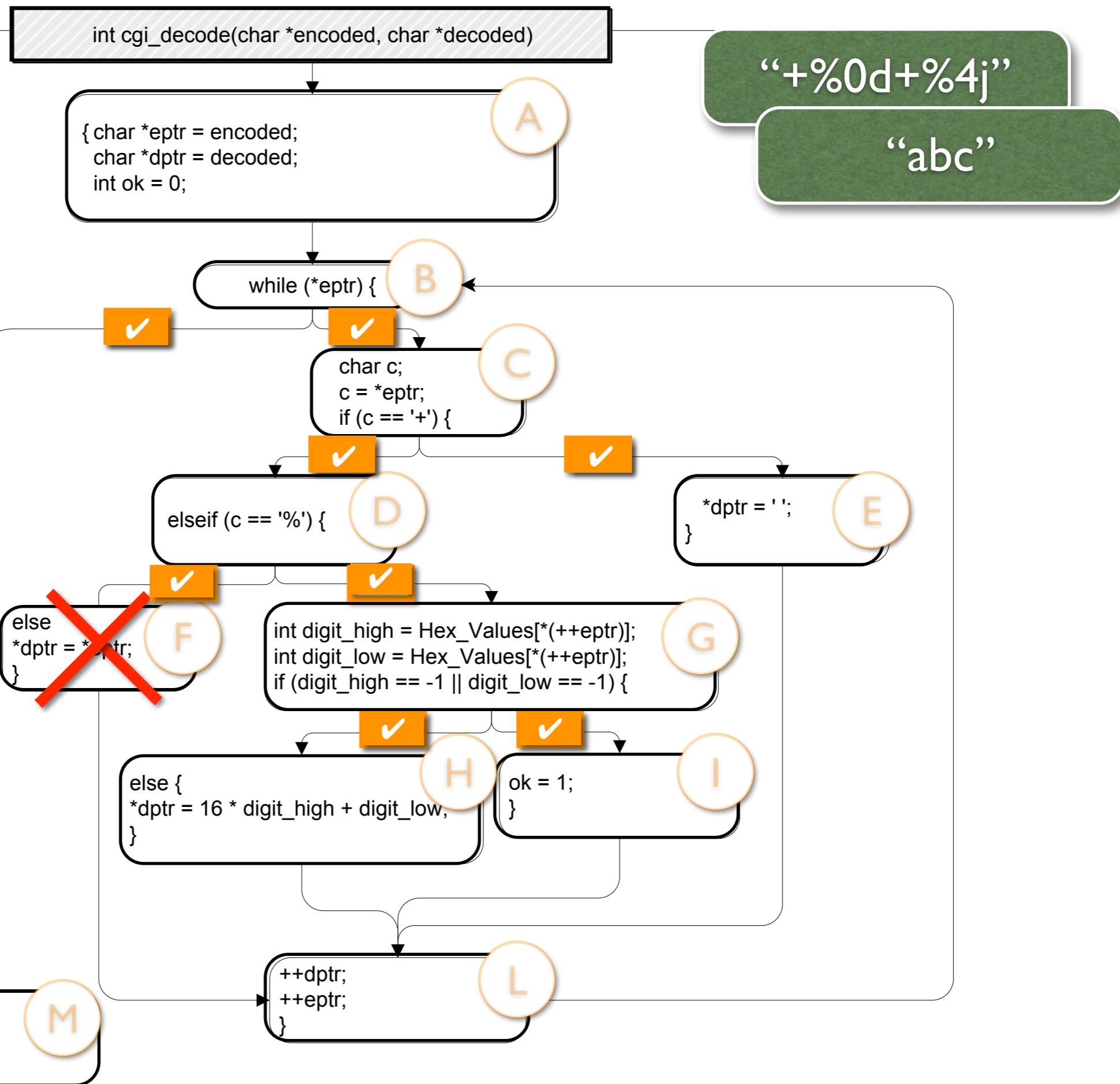
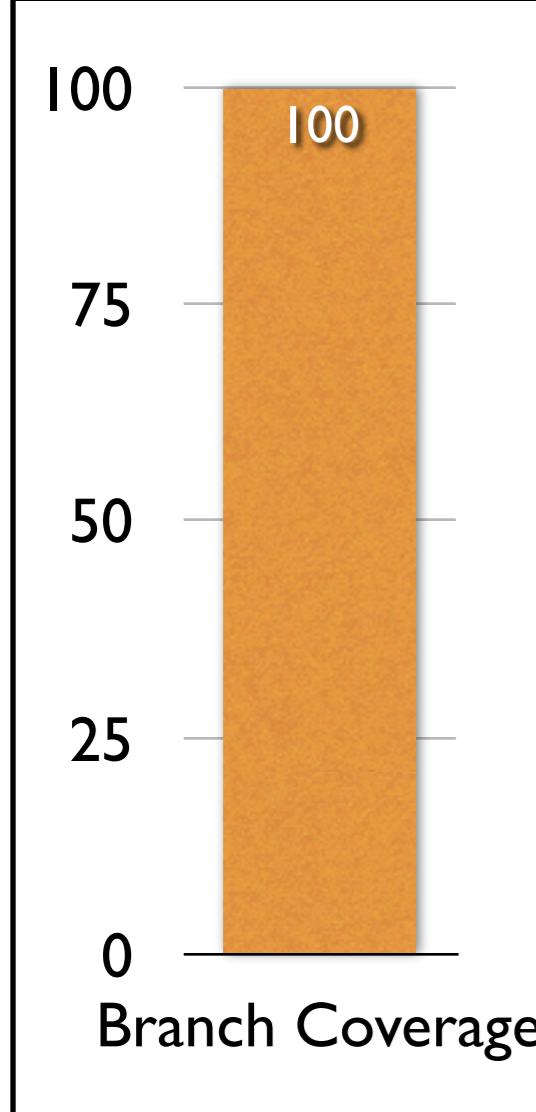




Asumamos que F no existe (defecto)



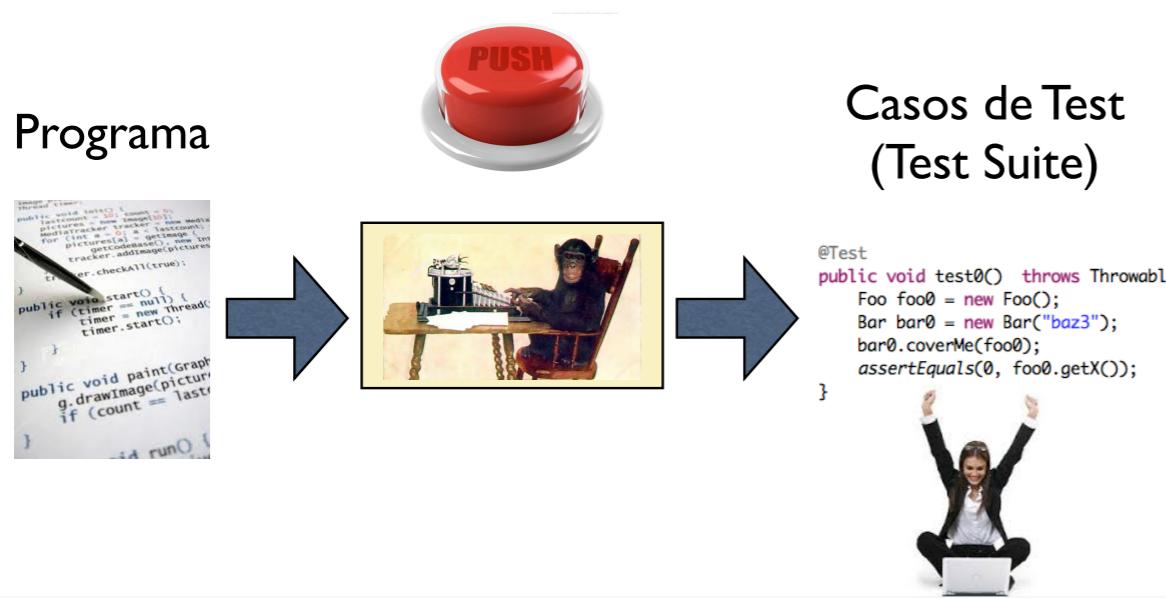
Si consideramos los branches, entonces ejercitaríamos el defecto



Taller #1: Cobertura

- Escribir un test suite JUnit Test con 100% branch/statemente coverage en la clase org.autotest.StackAr
- Descubrir al menos 2 bugs en la clase StackAr

Creación Automática de Casos de Tests



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 programa, test suite \rangle
- Usualmente expresado en forma de una regla – e.g., “todos los statements deben ser ejecutados”

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

Edge Testing

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