

+TADs & Testing Estructural o Caja Blanca

Introducción a la Programación

1

Tipos Abstractos de Datos

Repasando

Un Tipo Abstracto de Datos (TAD) es un modelo que define valores y las operaciones que se pueden realizar sobre ellos.

- ▶ Se denomina abstracto ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la representación interna o bien el cómo están implementadas sus operaciones.

El tipo lista que estuvimos viendo es un TAD:

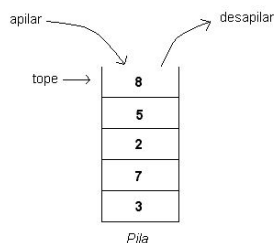
- ▶ Se define como una serie de elementos consecutivos
- ▶ Tiene diferentes operaciones asociadas: append, remove, etc
- ▶ Desconocemos cómo se usa/guarda la información almacenada dentro del tipo

2

Pila

Una pila es una lista de elementos de la cual se puede extraer el último elemento insertado.

- ▶ También se conocen como listas LIFO (Last In - First Out / el último que entra es el primero que sale)
- ▶ Operaciones básicas
 - ▶ apilar: ingresa un elemento a la pila
 - ▶ desapilar: saca el último elemento insertado
 - ▶ tope: devuelve (sin sacar) el último elemento insertado
 - ▶ vacía: retorna verdadero si está vacía



3

Pila

- ▶ En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una pila
- ▶ También, podemos importar el módulo LifoQueue que nos da una implementación de Pila

```
from queue import LifoQueue
pila = LifoQueue()
```

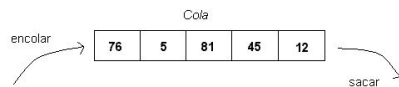
- ▶ Operaciones implementadas en el tipo:
 - ▶ apilar: ingresa un elemento a la cola
 - ▶ put
 - ▶ desapilar: devuelve y quita el último elemento insertado
 - ▶ get
 - ▶ tope: devuelve (sin sacar) el último elemento insertado
 - ▶ No está implementado
 - ▶ vacía: retorna verdadero si está vacía
 - ▶ empty

4

Cola

Una cola es una lista de elementos en donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la lista.

- ▶ También se conocen como listas FIFO (First In - First Out / el primero que entra es el primero que sale)
- ▶ Operaciones básicas
 - ▶ encolar: ingresa un elemento a la cola
 - ▶ sacar: saca el primer elemento insertado
 - ▶ vacia: retorna verdadero si está vacía



5

Cola

- ▶ En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una cola
- ▶ También, podemos importar el módulo Queue que nos da una implementación de Cola

```
from queue import Queue  
cola = Queue()
```

- ▶ Operaciones implementadas en el tipo:
 - ▶ encolar: ingresa un elemento a la cola
 - ▶ `put`
 - ▶ desencolar: saca el primer elemento insertado
 - ▶ `get`
 - ▶ vacia: retorna verdadero si está vacía
 - ▶ `empty`

6

Diccionario

Un diccionario es una estructura de datos que permite almacenar y organizar pares clave-valor.

- ▶ Las claves deben ser inmutables (como cadenas de texto, números, etc), mientras que los valores pueden ser de cualquier tipo de dato.
- ▶ La clave actúa como un identificador único para acceder a su valor correspondiente.
- ▶ Los diccionarios son mutables, lo que significa que se pueden modificar agregando, eliminando o actualizando elementos.
- ▶ No ordenados: Los elementos dentro de un diccionario no tienen un orden específico. No se garantiza que se mantenga el orden de inserción de los elementos.

```
diccionario = clave1:valor1, clave2:valor2, clave3:valor3
```

- ▶ Operaciones básicas de un diccionario:
 - ▶ Agregar un nuevo par Clave-Valor
 - ▶ Eliminar un elemento
 - ▶ Modificar el valor de un elemento
 - ▶ Verificar si existe una clave guardada
 - ▶ Obtener todas las claves
 - ▶ Obtener todos los elementos

7

Diccionario

Un diccionario es una estructura de datos que permite almacenar y organizar pares clave-valor.

- ▶ El valor puede ser cualquier tipo de dato, en particular podría ser otro diccionario

```
infoPaisFrancia = {'Capital':'París',  
                  'Campeonatos de Mundo':2}  
  
infoPaisArgentina = {'Capital':'Buenos Aires',  
                     'Campeonatos de Mundo':3}  
  
infoPaisChile = {'Capital':'Santiago',  
                 'Campeonatos de Mundo':0}  
  
infoPaises = {'Chile': infoPaisChile ,  
              'Argentina': infoPaisArgentina,  
              'Francia':infoPaisFrancia}
```

8

Manejo de Archivos

El manejo de archivos, también puede pensarse mediante la abstracción que nos brindan los TADs

- ▶ Necesitamos una operación que nos permita abrir un archivo
- ▶ Necesitamos una operación que nos permita leer sus líneas
- ▶ Necesitamos una operación que nos permita cerrar un archivo

```
# Abrir un archivo en modo lectura
```

```
archivo = open("archivo.txt", "r")
```

```
# Leer el contenido del archivo
```

```
contenido = archivo.read()
```

```
print(contenido)
```

```
# Cerrar el archivo
```

```
archivo.close()
```

9

Manejo de Archivos

```
archivo = open("PATH AL ARCHIVO", MODO, ENCODING)
```

- ▶ Algunos de los modos posibles son: escritura (w), lectura (r), texto (t - es el default)
- ▶ El encoding se refiere a como está codificado el archivo: UTF-8 o ASCII son los más frecuentes.

Operaciones básicas

- ▶ Lectura de contenido:

- ▶ read(size): Lee y devuelve una cantidad específica de caracteres o bytes del archivo. Si no se especifica el tamaño, se lee el contenido completo.
- ▶ readline(): Lee y devuelve la siguiente línea del archivo.
- ▶ readlines(): Lee todas las líneas del archivo y las devuelve como una lista.

- ▶ Escritura de contenido:

- ▶ write(texto): Escribe un texto en el archivo en la posición actual del puntero. Si el archivo ya contiene contenido, se sobrescribe.
- ▶ writelines(líneas): Escribe una lista de líneas en el archivo. Cada línea debe terminar con un salto de línea explícito.

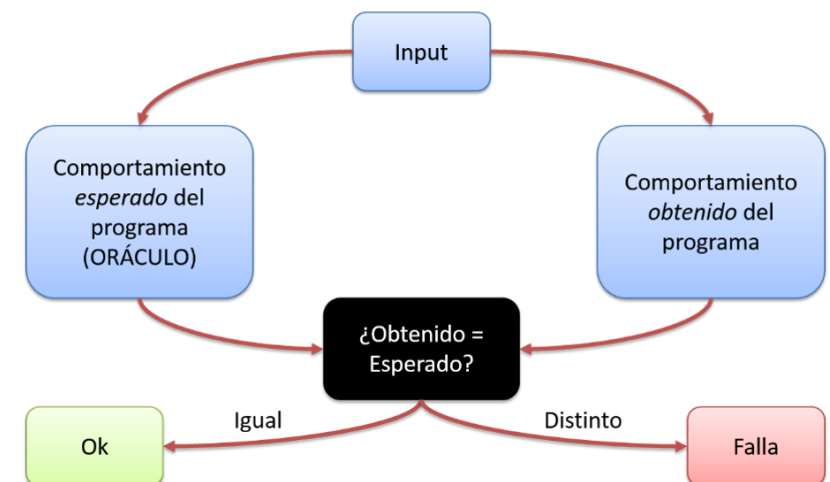
10

¿Podremos implementar este problema?

```
problema invertirTexto(in archivoOrigen: string, in archivoDestino: string): {  
    requiere: {El archivo nombreArchivo debe existir.}  
    asegura: {Se crea un archivo llamado archivoDestino cuyo contenido será el resultado de hacer un reverse en cada una de sus filas}  
    asegura: {Si el archivo archivoDestino existia, se borrará todo su contenido anterior}  
}
```

11

¿Cómo se hace testing?

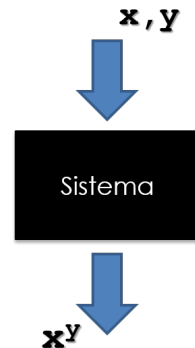


12

Criterios de **caja negra** o funcionales

- Los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

```
problema fastexp(x : Z, y : Z) : Z{  
  requiere: {(0 < x ∧ 0 ≤ y)}  
  asegura: {res = xy}}  
}
```

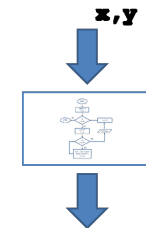


13

Criterios de **caja blanca** o estructurales

- Los datos de test se derivan a partir de la estructura interna del programa.

```
def fastexp(x: int, y: int) → int:  
  z: int = 1  
  while(y != 0):  
    if(esImpar(y)):  
      z = z * x  
      y = y - 1  
  
    x = x * x  
    y = y / 2  
  
  return z
```



¿Qué pasa si y es potencia de 2?
¿Qué pasa si y = 2n - 1?

14

Criterios de **caja blanca** o estructurales

Los criterios de caja blanca permiten identificar casos especiales según el flujo de control de la aplicación.

- Ver que sucede si entra o no en un IF
- Ver que sucede si entra o no a un ciclo
- Etc

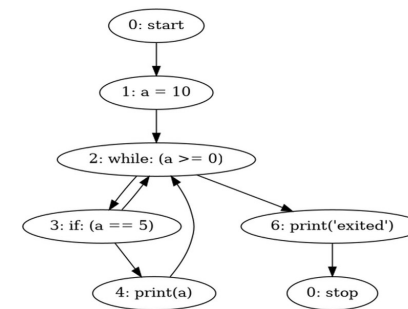
Pero tiene una tremenda dificultad: determinar el resultado esperado de un programa sin una especificación no es para nada trivial.

Por este motivo, el test de caja blanca se suele utilizar como:

- **Complemento al Test de Caja Negra:** permite encontrar más casos o definir casos más específicos
- Como **criterio de adecuación** del Test de Caja Negra: brinda herramientas que nos ayudan a determinar cuán bueno o confiable resultaron ser los test suites definidos.
 - En este contexto hablaremos de **Criterios de Cubrimiento**

15

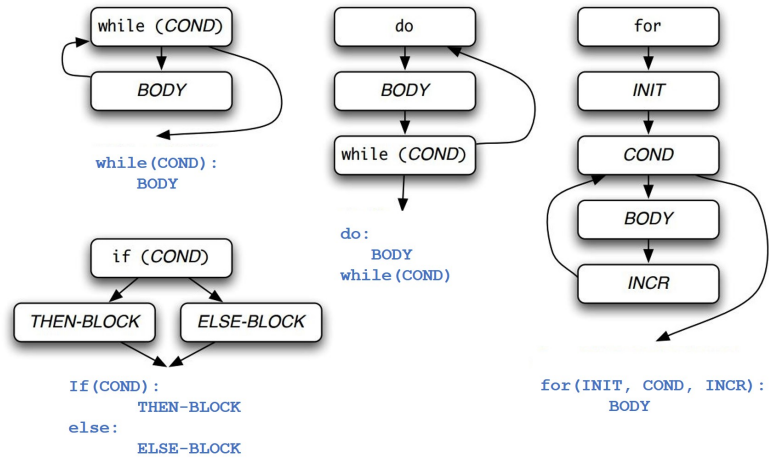
Control-Flow Graph



- El control flow graph (CFG) de un programa es sólo una **representación gráfica del programa**.
- El CFG es independiente de las entradas (su definición es estática)
- Se usa (entre otras cosas) para definir criterios de adecuación para test suites.
- Cuanto más *partes* son ejercitadas (cubiertas), mayores las chances de un test de descubrir una falla
- *partes* pueden ser: nodos, arcos, caminos, decisiones...

16

Control Flow Patterns



17

Ejemplo #1

```
problema valorAbsoluto(in x :  $\mathbb{Z}$ ) :  $\mathbb{Z}$ {
  requiere: { True }
  asegura: { res = ||x|| }
}
```

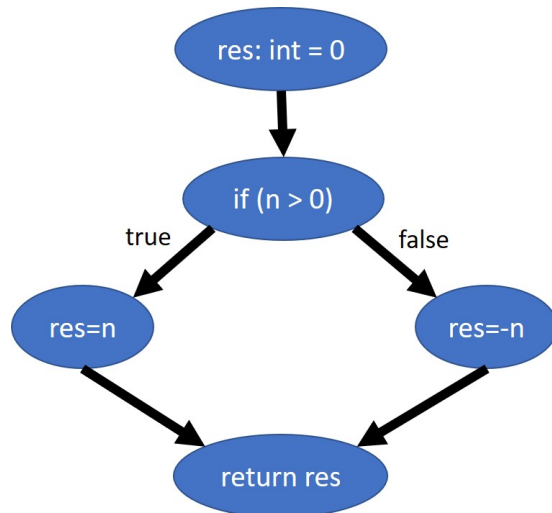
```
def valorAbsoluto(n: int) → int:
  res: int = 0

  if ( n > 0 ):
    res = n
  else:
    res = -n

  return res
```

18

CFG de valorAbsoluto



19

Ejemplo #2

```
problema sumar(in n :  $\mathbb{Z}$ ) :  $\mathbb{Z}$ {
  requiere: {  $n \geq 0$  }
  asegura: {  $res = \sum_{i=1}^n i$  }
}
```

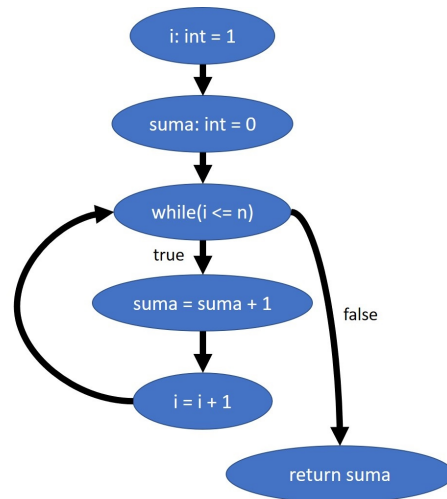
```
def sumar(n: int) → int:
  i: int = 1
  suma: int = 0

  while ( i ≤ n ):
    suma = suma + i
    i = i + 1

  return suma
```

20

CFG de sumar



21

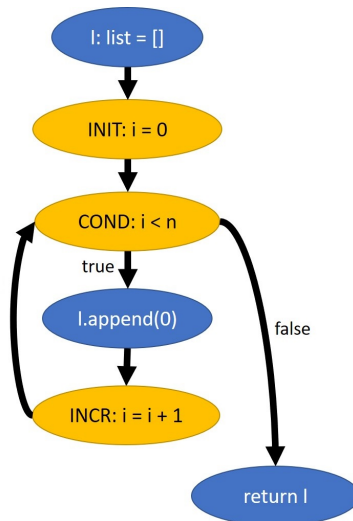
Ejemplo #3

```
problema crearListaN(in n :  $\mathbb{Z}$ ) : seq( $\mathbb{Z}$ ) {  
    requiere: {  $n \geq 0$  }  
    asegura: {  $|res| = n \wedge \#apariciones(res, 0) = n$  }  
}
```

```
def crearListN(int n) → list:  
    l: list = []  
  
    for i in range(0, n, 1):  
        l.append(0)  
  
    return l
```

22

CFG de crearListaN



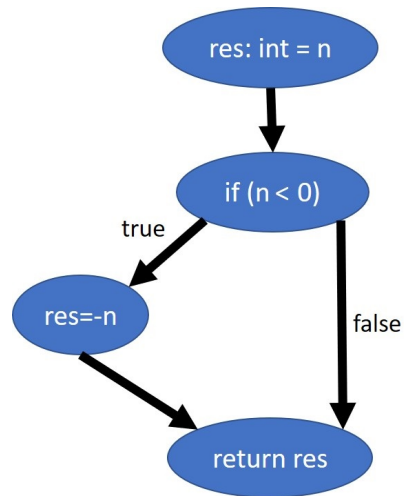
23

Ejemplo #4

```
def valorAbsoluto(n: int) → int:  
    res: int = n  
  
    if ( n < 0 ):  
        res = -n  
  
    return res
```

24

CFG de valor Absoluto



25

Criterios de Adecuación

- ▶ ¿Cómo sabemos que un *test suite* es *suficientemente bueno*?
- ▶ Un criterio de adecuación de test es un predicado que toma un valor de verdad para una tupla $\langle \text{programa}, \text{test suite} \rangle$
- ▶ Usualmente expresado en forma de una regla del estilo:
todas las sentencias deben ser ejecutadas

26

Cubrimiento de Sentencias

- ▶ Criterio de Adecuación: cada nodo (sentencia) en el CFG debe ser ejecutado al menos una vez por algún test case.
- ▶ Idea: un defecto en una sentencia sólo puede ser revelado ejecutando el defecto.
- ▶ Cobertura:

$$\frac{\text{cantidad nodos ejercitados}}{\text{cantidad nodos}}$$

27

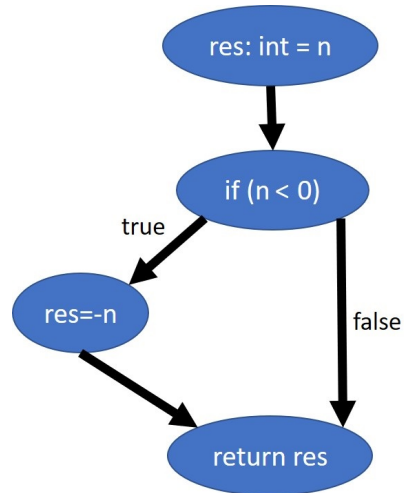
Cubrimiento de Arcos

- ▶ Criterio de Adecuación: todo arco en el CFG debe ser ejecutado al menos una vez por algún test case.
 - ▶ Si recorremos todos los arcos, entonces recorremos todos los nodos. Por lo tanto, el cubrimiento de arcos incluye al cubrimiento de sentencias.
 - ▶ Cobertura:
- $$\frac{\text{cantidad arcos ejercitados}}{\text{cantidad arcos}}$$
- ▶ El cubrimiento de sentencias (nodos) no incluye al cubrimiento de arcos. ¿Por qué?

28

Cubrimiento de Nodos no incluye cubrimiento de Arcos

Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los nodos pero que no cubra todos los arcos.

29

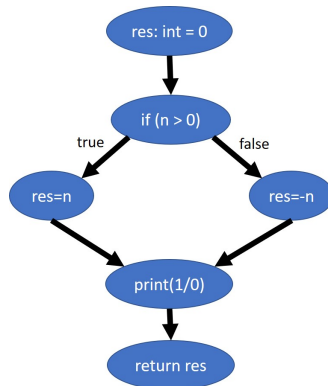
Cubrimiento de Decisiones (o Branches)

- Criterio de Adecuación: cada decisión (arco True o arco False) en el CFG debe ser ejecutado.
- Por cada arco **True** o arco **False**, debe haber al menos un test case que lo ejecute.
- Cobertura:
$$\frac{\text{cantidad decisiones ejercitadas}}{\text{cantidad decisiones}}$$
- El cubrimiento de decisiones **no implica** el cubrimiento de los arcos del CFG. ¿Por qué?

30

Cubrimiento de Branches no incluye cubrimiento de Arcos

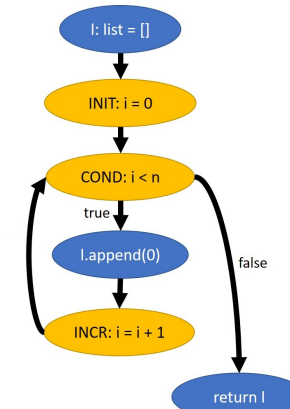
Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los branches pero que no cubra todos los arcos.

31

CFG de crearListaN



- ¿Cuántos nodos (sentencias) hay? 6
- ¿Cuántos arcos (flechas) hay? 6
- ¿Cuántas decisiones (arcos True y arcos False) hay? 2

32

Cubrimiento de Condiciones Básicas

- Una condición básica es una fórmula atómica (i.e. no divisible) que componen una decisión.
 - Ejemplo: `(digitHigh==1 || digitLow==1) && len>0`
 - Condiciones básicas:
 - `digitHigh==1`
 - `digitLow==1`
 - `len>0`
 - No es condición básica: `(digitHigh==1 || digitLow==1)`
- Criterio de Adecuación: cada condición básica de cada decisión en el CFG debe ser evaluada a verdadero y a falso al menos una vez
- Cobertura:

$$\frac{\text{cantidad de valores evaluados en cada condición}}{2 \times \text{cantidad condiciones básicas}}$$

33

Cubrimiento de Condiciones Básicas

- Sea una única decisión:
`(digitHigh==1 || digitLow==1) && len>0`

- Y el siguiente test case:

Entrada	digitHigh==1?	digitLow==1?	len>0?
digitHigh=1, digitLow=0 len=1,	True	False	True

- ¿Cuál es el cubrimiento de condiciones básicas?

$$C_{\text{cond.básicas}} = \frac{3}{2 \times 3} = \frac{3}{6} = 50\%$$

34

Cubrimiento de Condiciones Básicas

- Sea una única decisión:
`(digitHigh==1 || digitLow==1) && len>0`

- Y el siguiente test case:

Entrada	digitHigh==1?	digitLow==1?	len>0?
digitHigh=1, digitLow=0 len=1,	True	False	True
digitHigh=0, digitLow=1 len=0,	False	True	False

- ¿Cuál es el cubrimiento de condiciones básicas?

$$C_{\text{cond.básicas}} = \frac{6}{2 \times 3} = \frac{6}{6} = 100\%$$

35

Cubrimiento de Branches y Condiciones Básicas

- **Observación** Branch coverage no implica cubrimiento de Condiciones Básicas

- Ejemplo: **if(a && b)**
- Un test suite que ejercita solo `a = true, b = true` y `a = false, b = true` logra cubrir ambos branches de **if(a && b)**
- **Pero:** no alcanza cubrimiento de decisiones básica ya que falta `b = false`

- El criterio de cubrimiento de Branches y condiciones básicas necesita 100 % de cobertura de branches y 100 % de cobertura de condiciones básicas
- Para ser aprobado, todo software que controla un avión necesita ser testeado con cubrimiento de branches y condiciones básicas (RTCA/DO-178B y EUROCAE ED-12B).

36

Cubrimiento de Caminos

- Criterio de Adecuación: cada camino en el CFG debe ser transitado por al menos un test case.

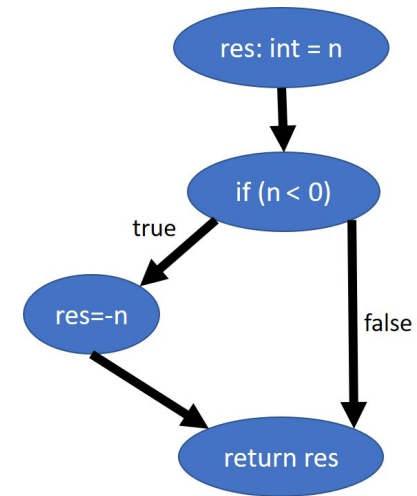
- Cobertura:

$$\frac{\text{cantidad caminos transitados}}{\text{cantidad total de caminos}}$$

37

Caminos para el CFG de valor Absoluto

Sea el siguiente CFG:

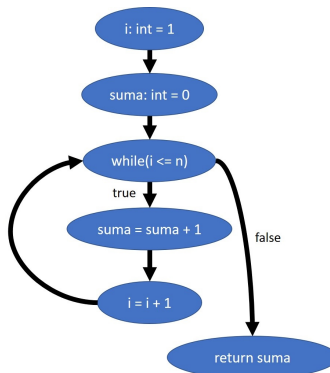


¿Cuántos caminos hay en este CFG? 2

38

Caminos para el CFG de sumar

Sea el siguiente CFG:



¿Cuántos caminos hay en este CFG? La cantidad de caminos no está acotada (∞)

39

Recap: Criterios de Adecuación Estructurales

- En todos estos criterios se usa el CFG para obtener una métrica del test suite
- **Sentencias:** cubrir todas los nodos del CFG.
- **Arcos:** cubrir todos los arcos del CFG.
- **Decisiones (Branches):** Por cada if, while, for, etc., la guarda fue evaluada a verdadero y a falso.
- **Condiciones Básicas:** Por cada componente básico de una guarda, este fue evaluado a verdadero y a falso.
- **Caminos:** cubrir todos los caminos del CFG. Como no está acotado o es muy grande, se usa muy poco en la práctica.

40

esPrimo()

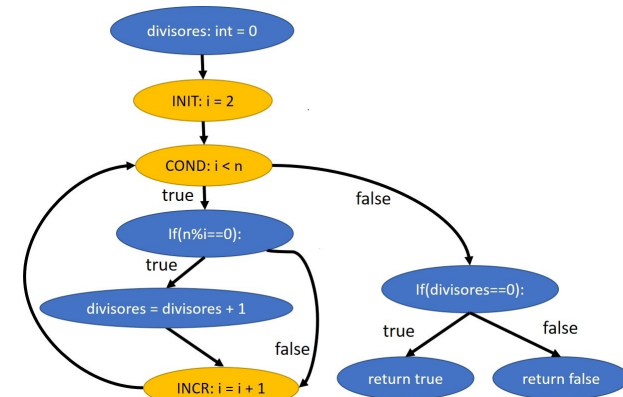
Sea la siguiente implementación que decide si un número $n > 1$ es primo:

```
def esPrimo(n : int) → bool:  
    divisores: int = 0  
    for i in range(2, n, 1):  
        if (n % i == 0):  
            divisores = divisores + 1  
  
    if (divisores == 0):  
        return true  
    else:  
        return false
```

Graficar el CFG de la función esPrimo().

41

esPrimo()



42

Cubrimientos

Sea el siguiente test suite para esPrimo():

- ▶ Test Case #1: valorPar
 - ▶ Entrada: $n = 2$
 - ▶ Salida esperada: $result = true$
- ▶ Test Case #2: valorImpar
 - ▶ Entrada: $n = 3$
 - ▶ Salida esperada: $result = true$
- ▶ ¿Cuál es el cubrimiento de sentencias (nodos) del test suite?

$$Cov_{sentencias} = \frac{7}{9} \sim 77\%$$

- ▶ ¿Cuál es el cubrimiento de decisiones (brances) del test suite?

$$Cov_{branches} = \frac{4}{6} \sim 66\%$$

43

Cubrimientos

Sea el siguiente test suite para esPrimo():

- ▶ Test Case #1: valorPrimo
 - ▶ Entrada: $n = 3$
 - ▶ Salida esperada: $result = true$
- ▶ Test Case #2: valorNoPrimo
 - ▶ Entrada: $n = 4$
 - ▶ Salida esperada: $result = false$
- ▶ ¿Cuál es el cubrimiento de sentencias (nodos) del test suite?

$$Cov_{sentencias} = \frac{9}{9} = 100\%$$

- ▶ ¿Cuál es el cubrimiento de decisiones (brances) del test suite?

$$Cov_{branches} = \frac{6}{6} = 100\%$$

44

Discusión

- ▶ ¿Puede haber partes (nodos, arcos, branches) del programa que no sean alcanzables con **ninguna** entrada válida (i.e. que cumplan la precondition)?
- ▶ ¿Qué pasa en esos casos con las métricas de cubrimiento?
- ▶ Existen esos casos (por ejemplo: código defensivo o código que sólo se activa ante la presencia de un estado inválido)
- ▶ El 100 % de cubrimiento suele ser no factible, por eso es una medida para analizar con cuidado y estimar en función al proyecto (ejemplo: 70 %, 80 %, etc.)