Testing de caja blanca

Guía 9

Repaso

- **Test input** (dato de prueba): Es un valor concreto de los parámetros de entrada que usamos para ejecutar un programa.

- **Test case** (caso de prueba): Es un pequeño programa que ejecuta el test input y chequea de forma automática si estos inputs cumplen el resultado esperado o no.

- **Test Suite**: Es un conjunto de test cases.

Guía 9. Ejercicio 5

problema signo (in x: $\mathbb{R}$, out result: $\mathbb{Z}$) {
    requiere: $\{True\}$
    asegura: $\{(result = 0 \land x = 0) \lor (result = -1 \land x < 0) \lor (result = 1 \land x > 0)\}$
}

```
def signo(x: float) -> int:
L1:    result: int = 0
L2:    if x<0:
L3:        result = -1
L4:    elif x>0:
L5:        result = 1
L6:    return result
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `signo`.
2. Escribir un test suite que ejecute todas las líneas del programa `signo`.
3. ¿El test suite del punto anterior ejecuta todas las posibles decisiones ("branches") del programa?

SOLUCIÓN

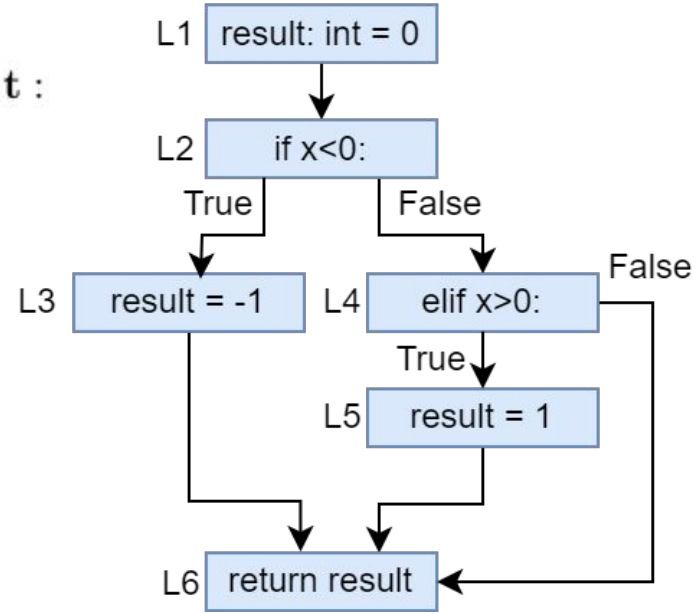
Ejercicio 5

```
def signo(x: float) -> int:
L1:     result: int = 0
L2:     if x<0:
L3:         result = -1
L4:     elif x>0:
L5:         result = 1
L6:     return result
```

Implementación **Control Flow-Graph** **Test Suite**

Ejercicio 5

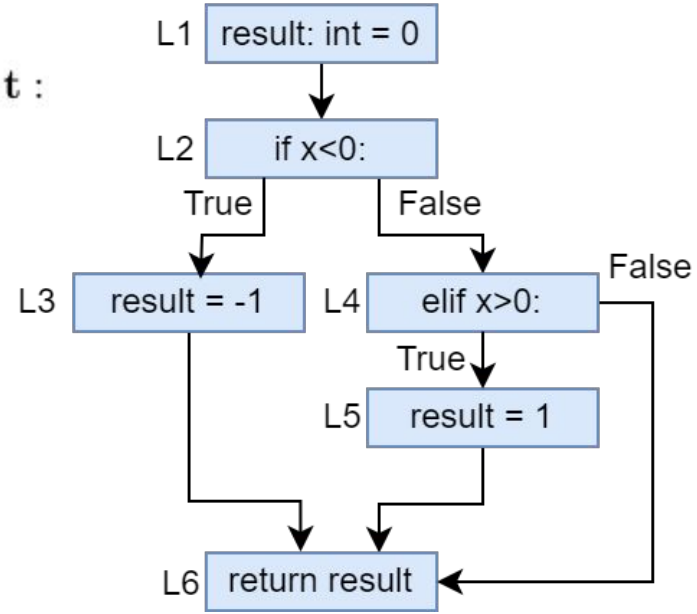
```
def signo(x: float) -> int:
L1:     result: int = 0
L2:     if x<0:
L3:         result = -1
L4:     elif x>0:
L5:         result = 1
L6:     return result
```



Implementación **Control Flow-Graph** **Test Suite**

Ejercicio 5

```
def signo(x: float) -> int:
L1:     result: int = 0
L2:     if x<0:
L3:         result = -1
L4:     elif x>0:
L5:         result = 1
L6:     return result
```



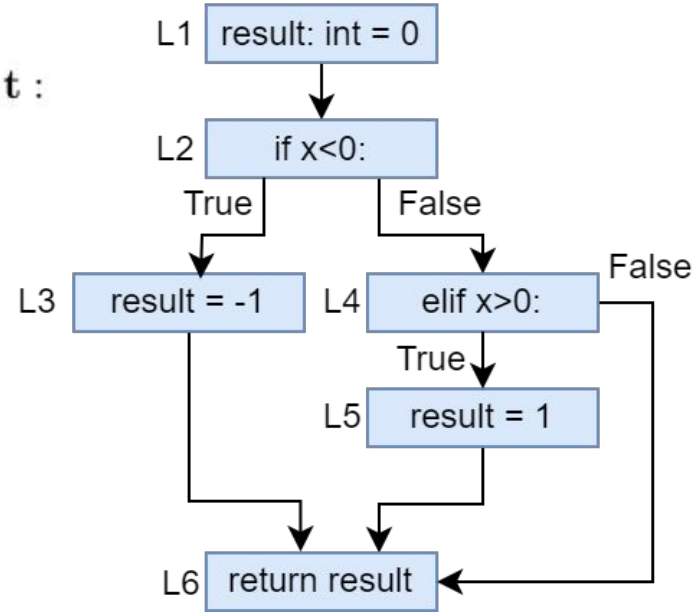
¿Los casos de test los hacemos viendo el código o la especificación?

Implementación **Control Flow-Graph** **Test Suite**

Ejercicio 5

```
problema signo (in x: ℝ) : ℤ  {
      requiere: {True}
      asegura: {(result = 0 ∧ x = 0) ∨ (result = −1 ∧ x < 0) ∨ (result = 1 ∧ x > 0)}
}
```

```
def signo(x: float) -> int:
L1:    result: int = 0
L2:    if x<0:
L3:        result = -1
L4:    elif x>0:
L5:        result = 1
L6:    return result
```

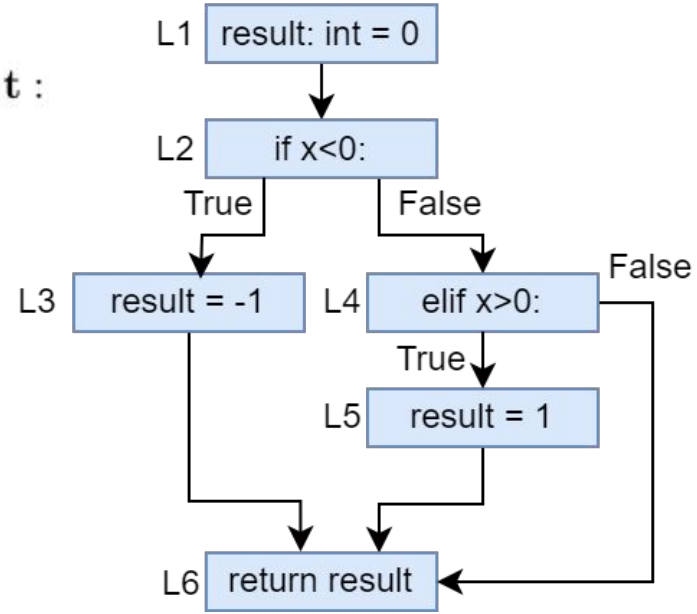


Implementación **Control Flow-Graph** **Test Suite**

Ejercicio 5

problema signo (in x: $\mathbb{R}$) : $\mathbb{Z}$ {
    requiere: $\{True\}$
    asegura: $\{(result = 0 \land x = 0) \lor (result = -1 \land x < 0) \lor (result = 1 \land x > 0)\}$
}

```
def signo(x: float) -> int:
L1:    result: int = 0
L2:    if x<0:
L3:        result = -1
L4:    elif x>0:
L5:        result = 1
L6:    return result
```



Test Case #1: valorPositivo
    entrada x = 1
    salida esperada = 1

Test Case #2: valorNegativo
    entrada x = -3
    salida esperada = -1

Implementación

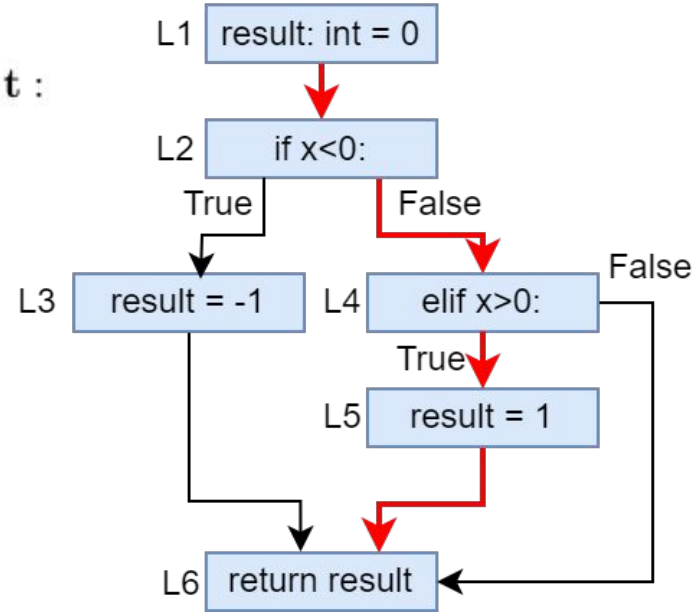
Control Flow-Graph

Test Suite

Ejercicio 5

problema signo (in x: $\mathbb{R}$) : $\mathbb{Z}$ {
    requiere: $\{True\}$
    asegura: $\{(result = 0 \land x = 0) \lor (result = -1 \land x < 0) \lor (result = 1 \land x > 0)\}$
}

```
def signo(x: float) -> int:
L1:     result: int = 0
L2:     if x<0:
L3:         result = -1
L4:     elif x>0:
L5:         result = 1
L6:     return result
```



Test Case #1: valorPositivo
entrada x = 1
salida esperada = 1

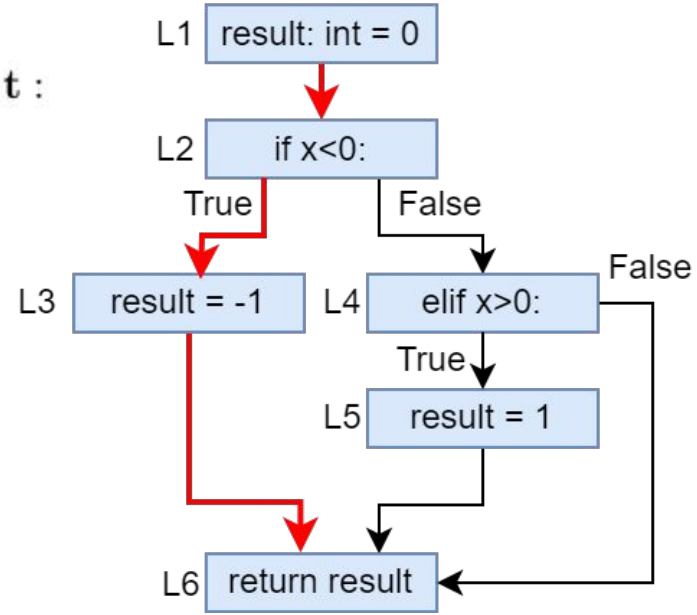
Test Case #2: valorNegativo
entrada x = -3
salida esperada = -1

Implementación | **Control Flow-Graph** | **Test Suite**

Ejercicio 5

problema signo (in x: $\mathbb{R}$) : $\mathbb{Z}$ {
    requiere: $\{True\}$
    asegura: $\{(result = 0 \land x = 0) \lor (result = -1 \land x < 0) \lor (result = 1 \land x > 0)\}$
}

```
def signo(x: float) -> int:
L1:     result: int = 0
L2:     if x<0:
L3:         result = -1
L4:     elif x>0:
L5:         result = 1
L6:     return result
```



Test Case #1: valorPositivo
entrada x = 1
salida esperada = 1

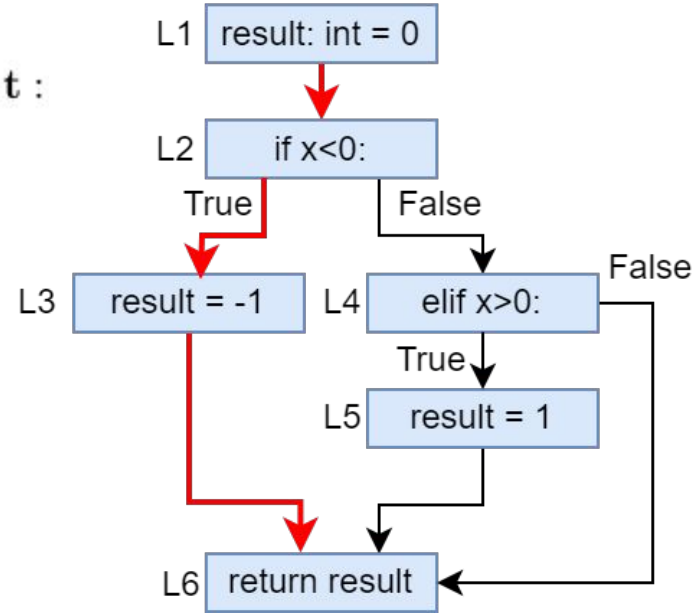
Test Case #2: valorNegativo
entrada x = -3
salida esperada = -1

Implementación | **Control Flow-Graph** | **Test Suite**

Ejercicio 5

problema signo (in x: $\mathbb{R}$) : $\mathbb{Z}$ {
    requiere: {$True$}
    asegura: {$(result = 0 \land x = 0) \lor (result = -1 \land x < 0) \lor (result = 1 \land x > 0)$}
}

```
def signo(x: float) -> int:
L1:    result: int = 0
L2:    if x<0:
L3:        result = -1
L4:    elif x>0:
L5:        result = 1
L6:    return result
```



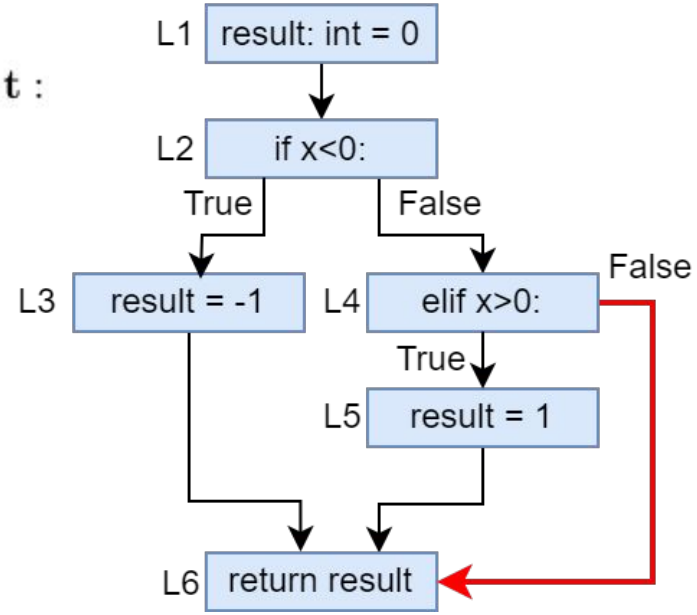
¿Cubrimos todas las branches con estos dos casos de test?

Implementación | **Control Flow-Graph** | **Test Suite**

Ejercicio 5

problema signo (in x: $\mathbb{R}$) : $\mathbb{Z}$ {
    requiere: {$True$}
    asegura: {$(result = 0 \land x = 0) \lor (result = -1 \land x < 0) \lor (result = 1 \land x > 0)$}
}

```
def signo(x: float) -> int:
L1:     result: int = 0
L2:     if x<0:
L3:         result = -1
L4:     elif x>0:
L5:         result = 1
L6:     return result
```



Test Case #1: valorPositivo
entrada x = 1
salida esperada = 1

Test Case #2: valorNegativo
entrada x = -3
salida esperada = -1

Test Case #3: valorCero
entrada x = 0
salida esperada = 0

Implementación | **Control Flow-Graph** | **Test Suite**

Ejercicio 15

1. Escribir los diagramas de control de flujo (control-flow graph) para cantidadDePrimos y la función auxiliar esPrimo.

2. Escribir un test suite que cubra todos las líneas de programa del programa cantidadDePrimos y esPrimo.

3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.

Ejercicio 15

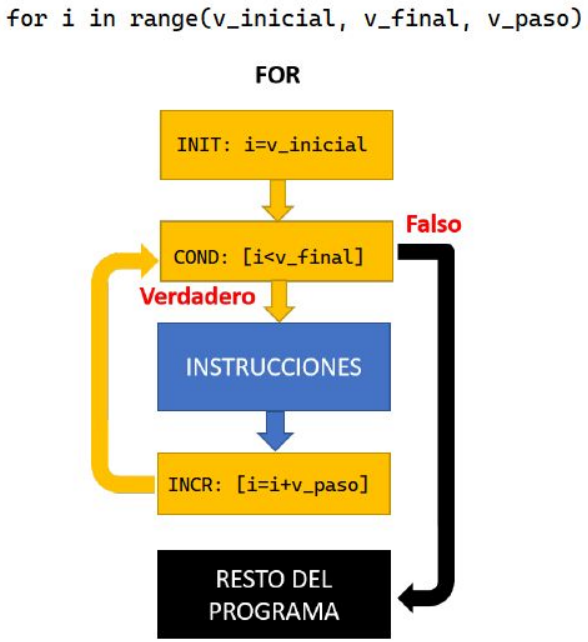
```
problema cantidadDePrimos (in n: ℤ) : ℤ {
    requiere: {n ≥ 0}
    asegura: {result = Σ_{i=2}^{n} (if esPrimo(i) then 1 else 0 fi)}
}
```

```
        def cantidadDePrimos(n: int) -> int:
L1:         result: int = 0
L2,L3,L4:   for i in range(2,n+1,1):
L5:             inc: bool = esPrimo(i)
L6:             if inc==True:
L7:                 result += 1
L8:         return result
```

```
        #Funcion auxiliar
        def esPrimo(x: int) -> bool:
L9:         result: bool = True
L10,L11,L12:    for i in range(2, x, 1):
L13:                if x % i == 0:
L14:                    result = False
L15:            return result
```

Control Flow Graph [FOR]

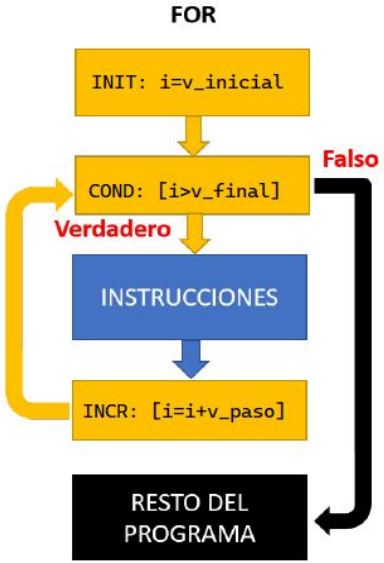
si v_paso $>$ 0



Control Flow Graph [FOR]

si v_paso < 0

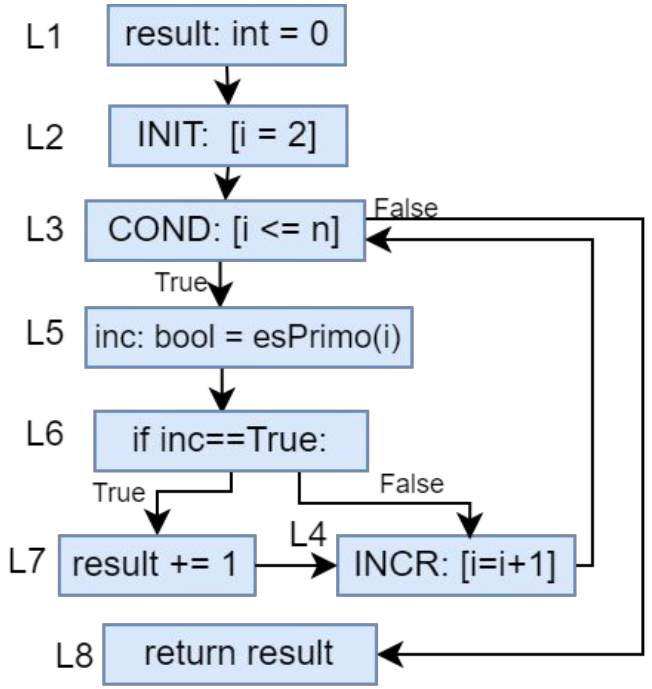
```
for i in range(v_inicial, v_final, v_paso)
```



SOLUCIÓN

1. Escribir los CFG para cantidadDePrimos y la función auxiliar esPrimo.

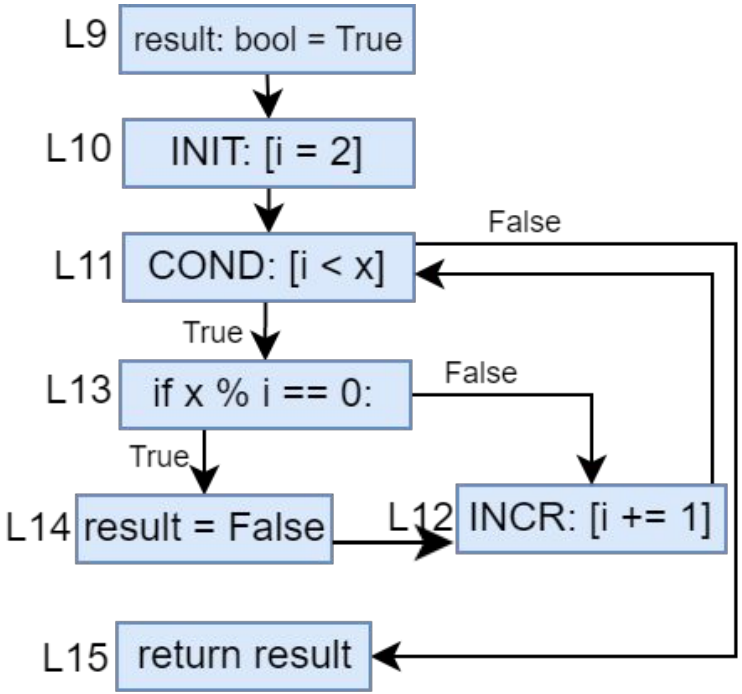
```
          def cantidadDePrimos(n: int) -> int:
L1:           result: int = 0
L2,L3,L4:     for i in range(2,n+1,1):
L5:               inc: bool = esPrimo(i)
L6:               if inc==True:
L7:                   result += 1
L8:           return result
```



cantidadDePrimos

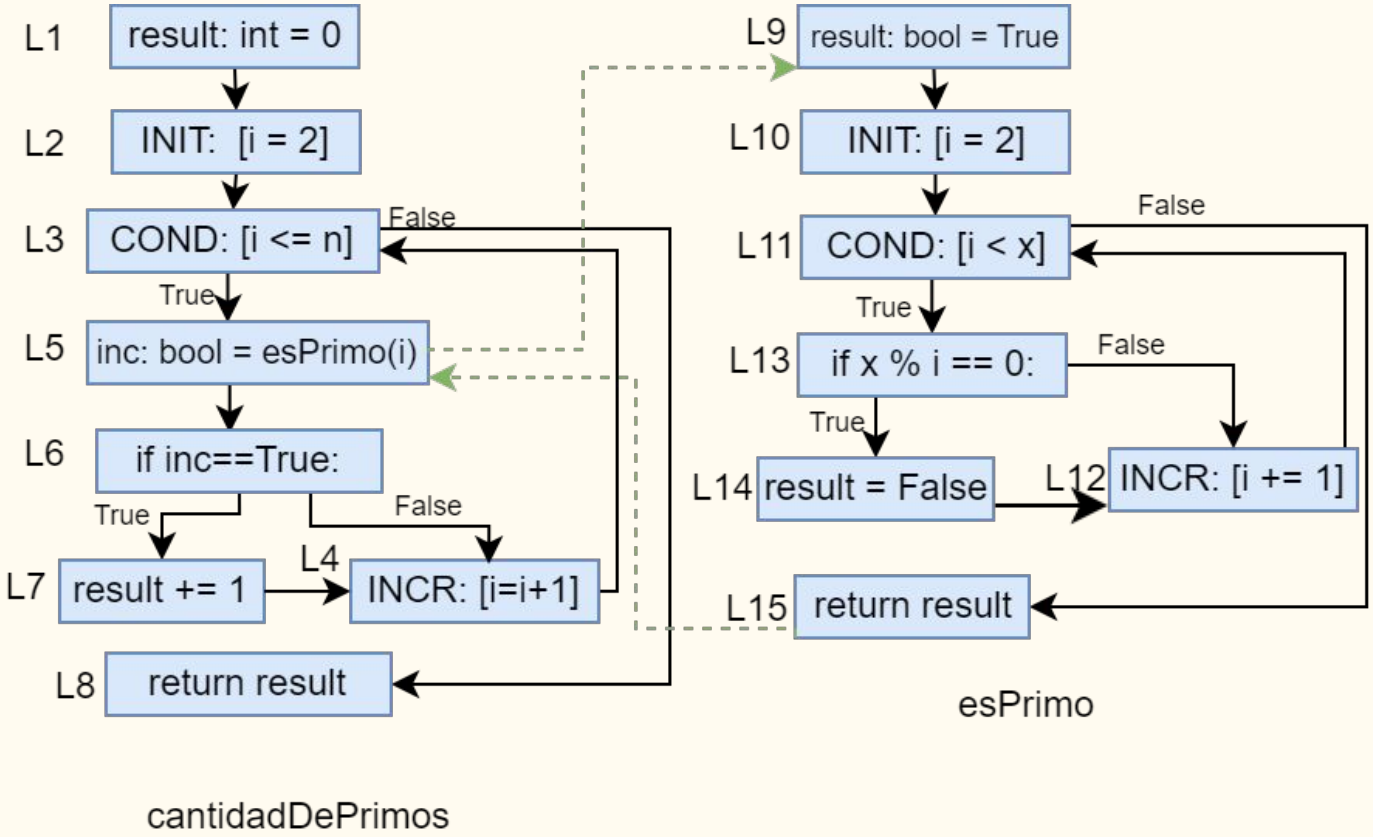
1. Escribir los CFG para cantidadDePrimos y la función auxiliar esPrimo.

```
                #Funcion auxiliar
                def esPrimo(x: int) -> bool:
L9:                 result: bool = True
L10,L11,L12:        for i in range(2, x, 1):
L13:                    if x % i == 0:
L14:                        result = False
L15:                return result
```

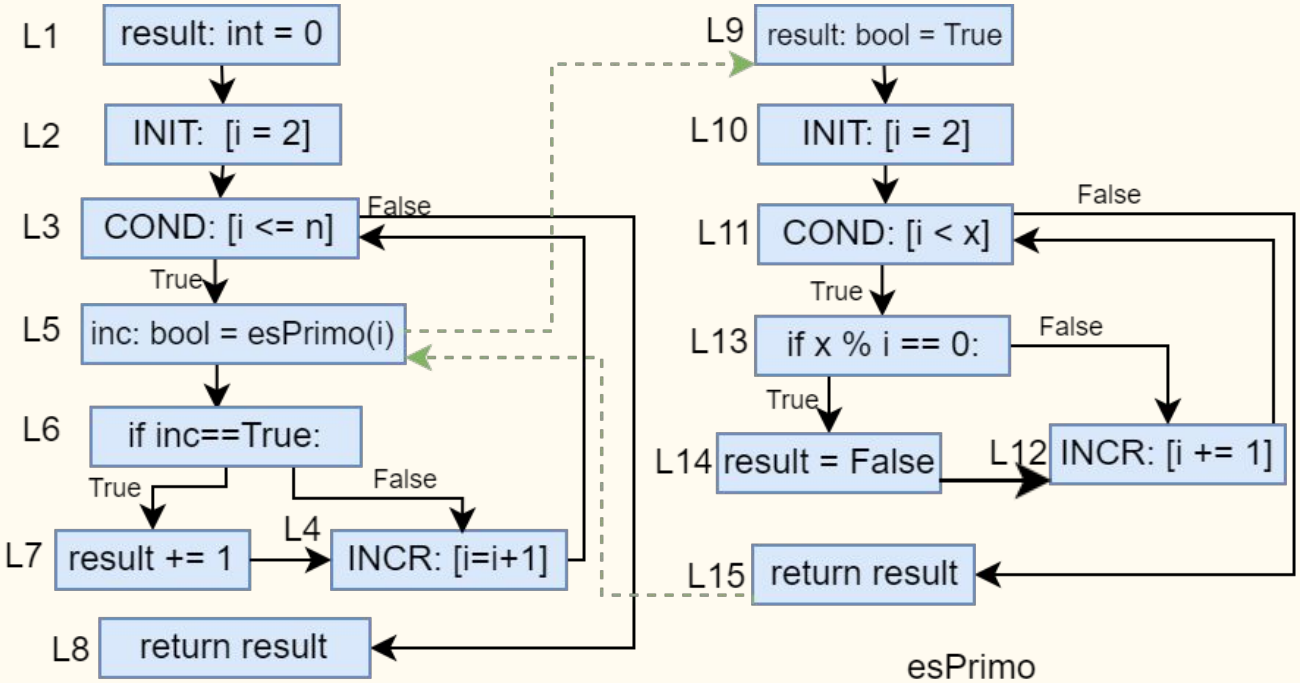


esPrimo

1. Escribir los CFG para cantidadDePrimos y la función auxiliar esPrimo.



2. Escribir un test suite que cubra todos las líneas de programa del programa cantidadDePrimos y esPrimo.

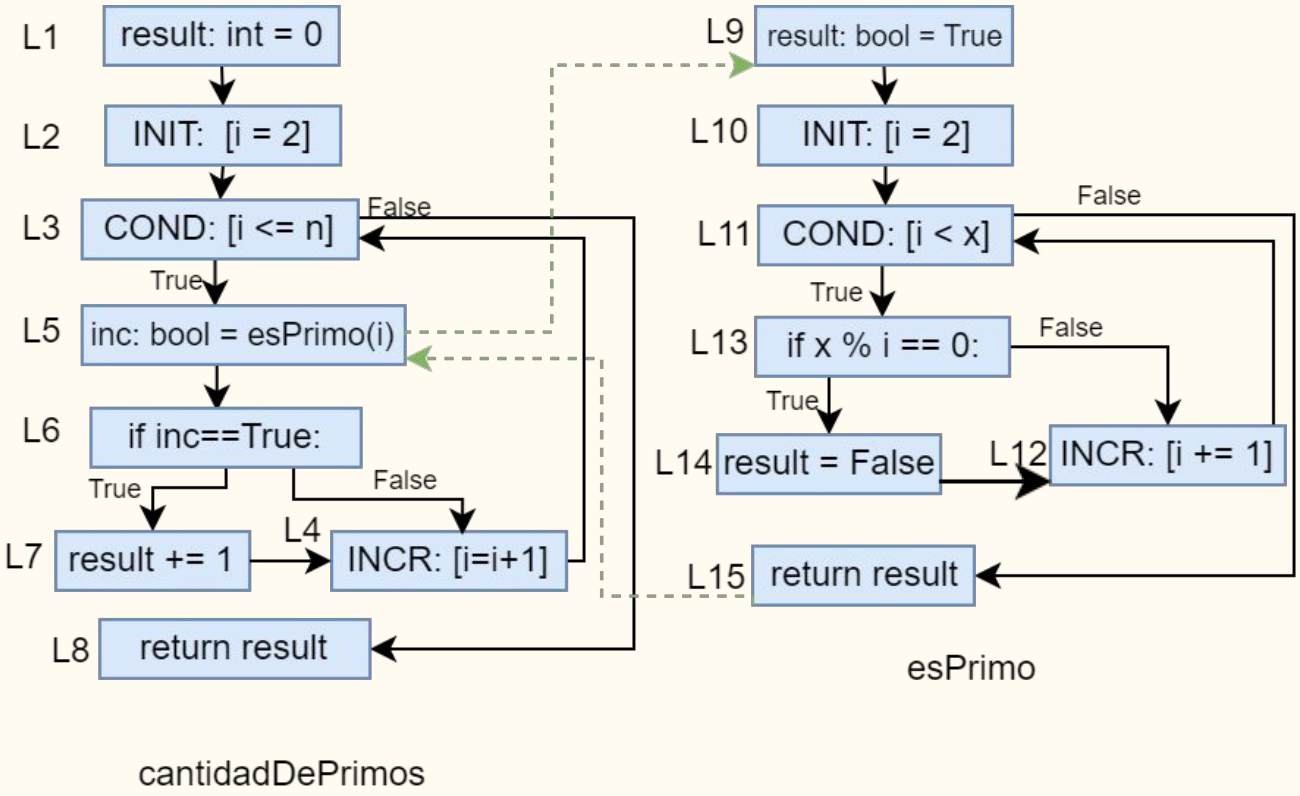


Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Suite

2. Escribir un test suite que cubra todos las líneas de programa del programa cantidadDePrimos y esPrimo.

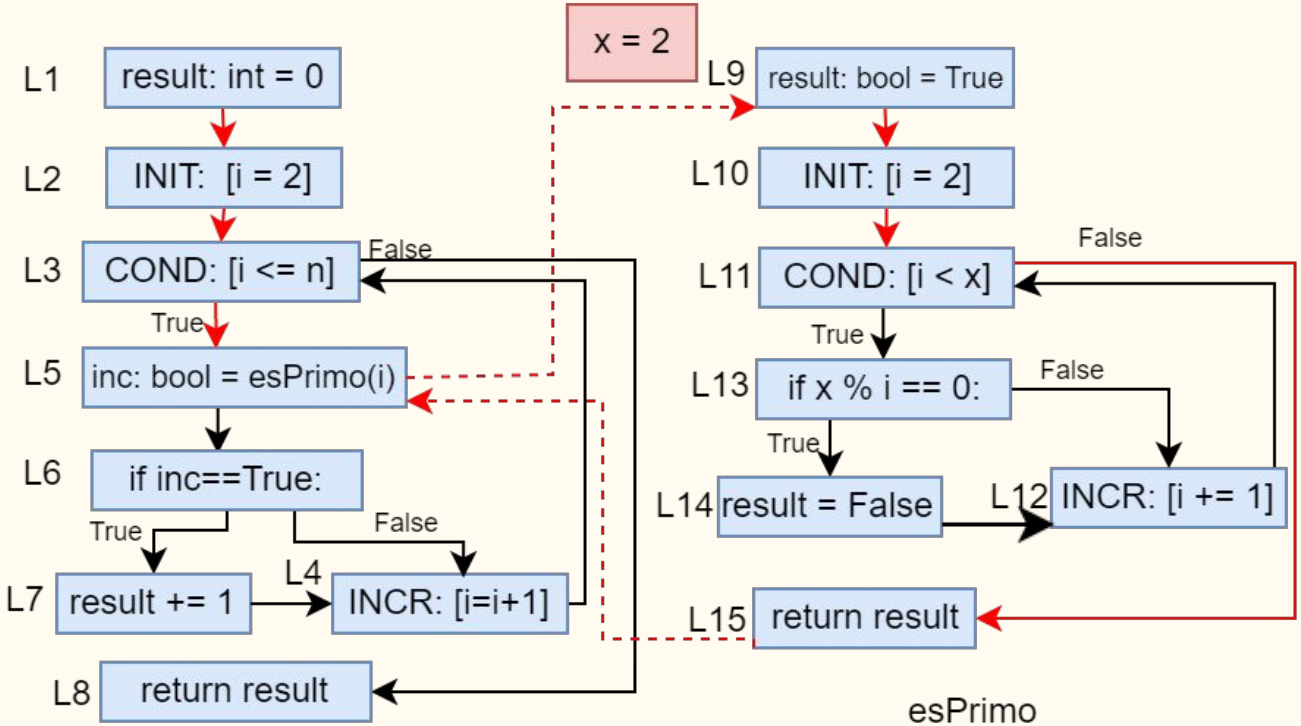


Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Suite

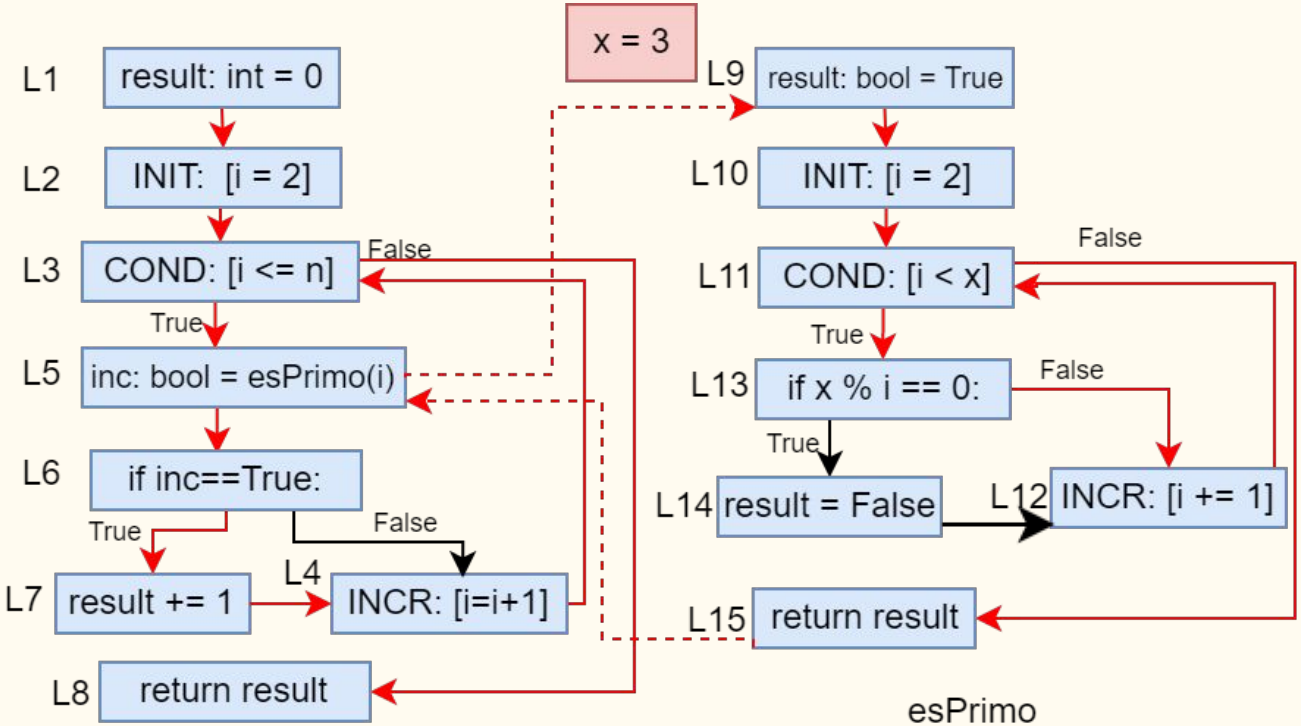
"detenemos" la "ejecución" cuando se ejecutó esPrimo(2)



Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Suite

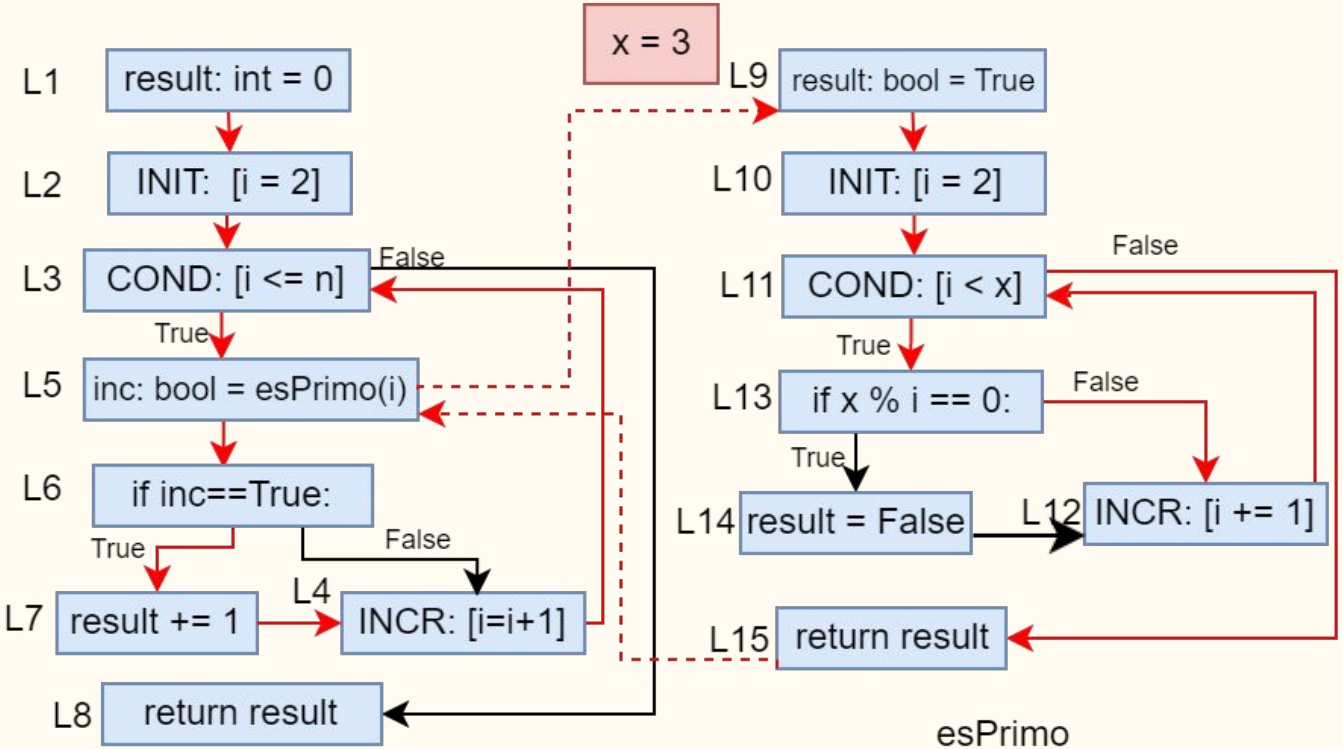


Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Suite

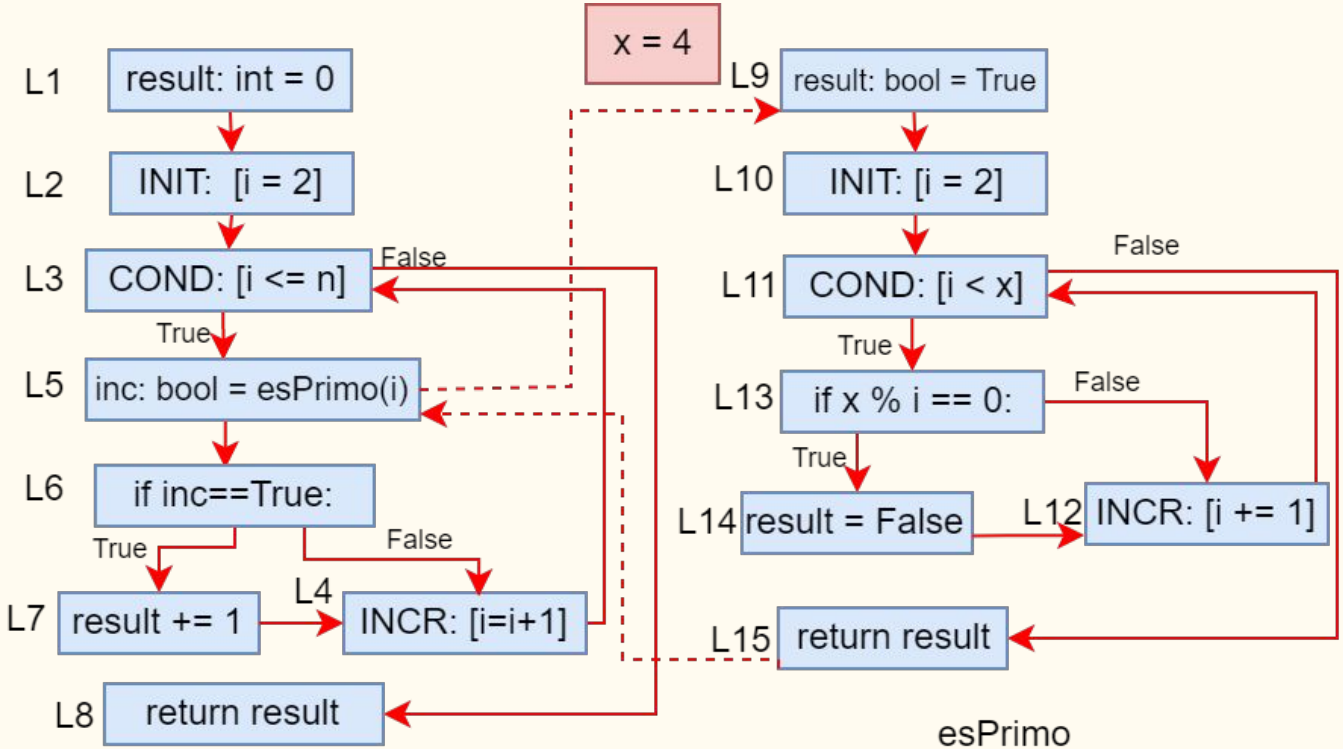
"detenemos" la "ejecución" cuando se ejecutó esPrimo(3) .
Hasta acá es igual que antes. Ahora continuamos con x=4



Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Suite

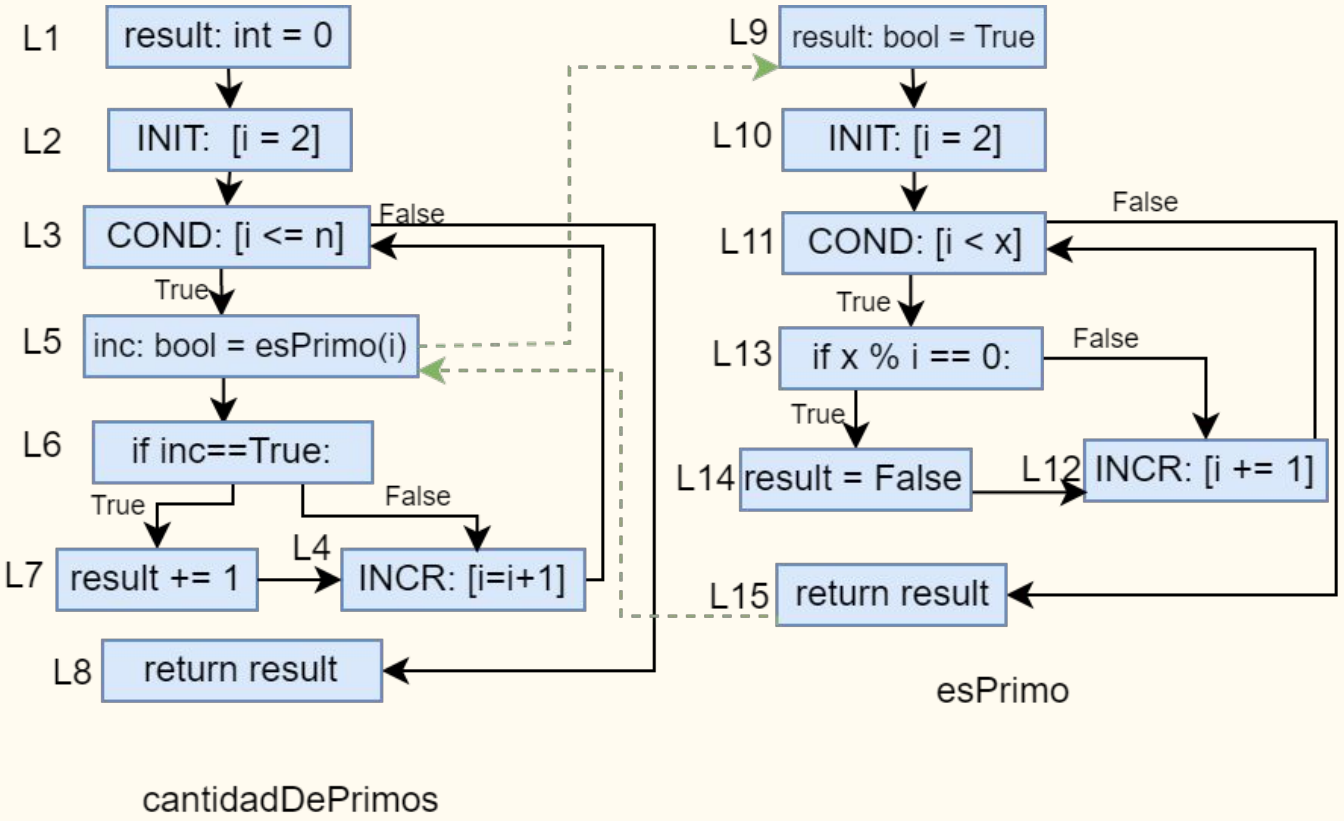


Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Suite

Es importante testear además el caso cuando nunca entra al ciclo!

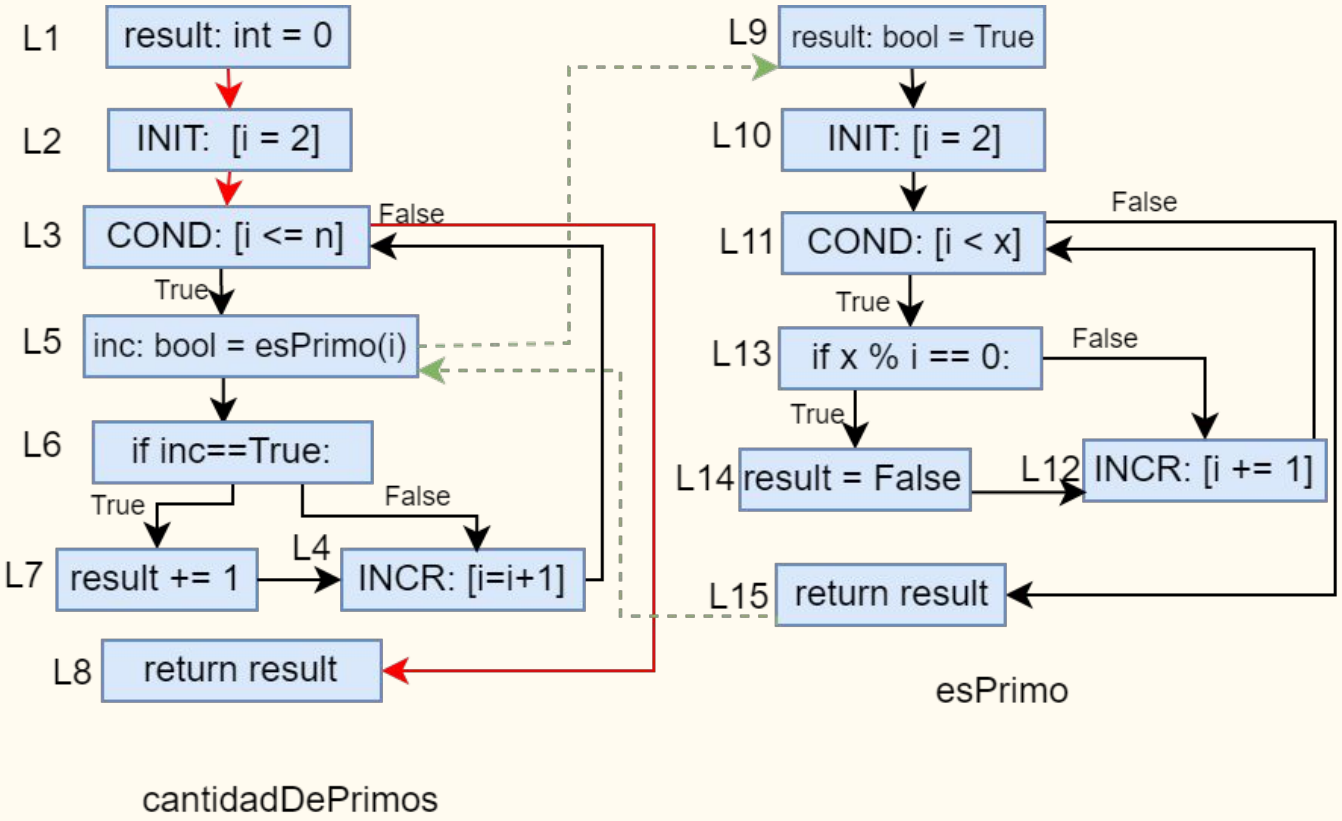


Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Case #2: noPrimo
entrada n = 1
salida esperada = 0

Test Suite



Test Case #1: primo
entrada n = 3
salida esperada = 2

Test Case #2: noPrimo
entrada n = 4
salida esperada = 2

Test Case #2: noPrimo
entrada n = 1
salida esperada = 0

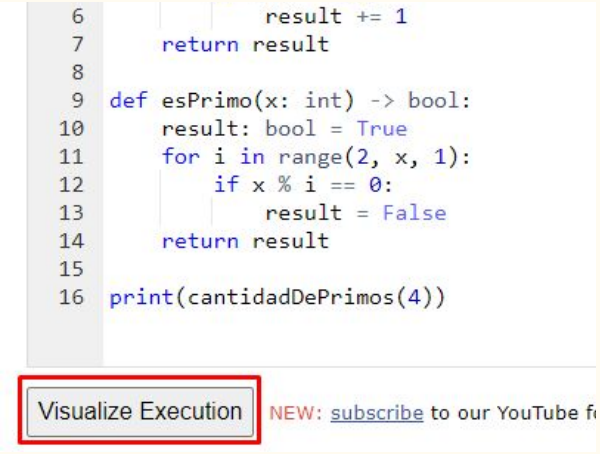
Test Suite

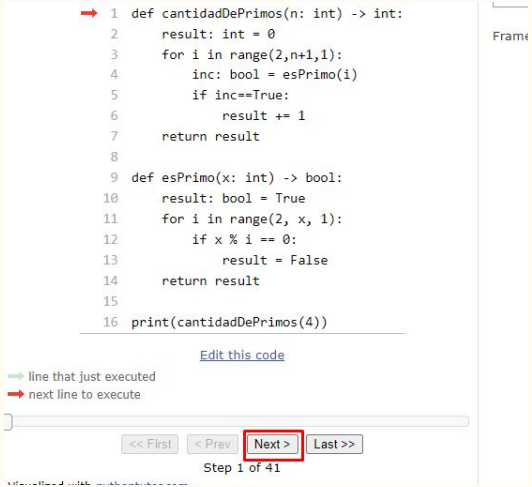
3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.

¡Cubrimos todas las líneas y branches con los casos de test anteriores!

Python Tutor

- Permite ver la ejecución de un programa escrito en Python (entre otros lenguajes) siguiendo el “paso a paso” de forma visual.
- Para ver el ejemplo anterior, ingresar acá.
  - Click en “Visualize Execution” y luego en “Next”





Bonus

Ejercicio 17. El programa que se transcribe más abajo pretende determinar la longitud del *fragmento* más largo en un texto. Los fragmentos son porciones del texto que no contienen punto y coma. Por ejemplo, en el siguiente texto el fragmento más largo es `"Mercurio"`, de ocho letras:

```
"Mercurio;Venus;Tierra;Marte;Júpiter"
```

1. Escribir el CFG (control-flow graph) de la función.
2. Escribir *un test* que encuentre el defecto presente en el código. Es decir, escribir una entrada que cumple con el *requiere* pero que el resultado de ejecutar el código no cumple el *asegura* (Justificar la respuesta).
3. Agregar casos de test para cubrir todos los branches del programa.

Bonus - Especificación

```
problema calcularFragmentoMásLargo (s: seq⟨Char⟩) : ℤ {
      requiere: {True}
      asegura: {(∃i, j : ℤ)(esFragmento(s, i, j) ∧ result ≤ j − i) ∧ (∀i, j : ℤ)(esFragmento(s, i, j) ⟶ result ≥ j − i)}
}
pred esFragmento (s: seq⟨Char⟩, i, j : ℤ) {
    0 ≤ i ≤ j ≤ |s| ∧_L (∀k : ℤ)(i ≤ k < j ⟶_L ¬esPuntoYComa(s[k]))
}
pred esPuntoYComa (c: Char) {
    c = ';'
}
```

Bonus - Implementación

```
def calcular_fragmento_mas_largo(s: str) -> int:
        result: int = 0
        i: int = 0
        f: int = 0 # Longitud del fragmento actual
        while i < len(s):
                if f > result:
                        result = f
                if s[i] == ';':
                        f = 0
                else:
                        f+=1
                i+=1
        return result
```