Introducción a la Programación

Guía Práctica 9
Testing de caja blanca

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ejercicio 1. ★ Sea el siguiente programa:

```
def max (x: int, y: int) -> int:
L1:     result: int = 0
L2:     if x < y:
L3:         result = y
        else:
L4:         result = x
L5:     return result
```

Y los siguientes casos de test:

- test1:
  - Entrada x = 0, y=0
  - Resultado esperado result=0
- test2:
  - Entrada x = 0, y=1
  - Resultado esperado result=1

1. Describir el diagrama de control de flujo (control-flow graph) del programa `max`.
2. Detallar qué líneas del programa cubre cada test

Test	L1	L2	L3	L4	L5
test1					
test2					

3. Detallar qué decisiones (branches) del programa cubre cada test

Test	L2-True	L2-False
test1		
test2		

4. Decidir si la siguiente afirmación es verdadera o falsa: *"El test suite compuesto por test1 y test2 cubre el 100 % de las líneas del programa y el 100 % de las decisiones (branches) del programa"*. Justificar.

Ejercicio 2. ★ Sea la siguiente especificación del problema de retornar el mínimo elemento entre dos números enteros:

`problema min` (in x: $\mathbb{Z}$, in y: $\mathbb{Z}$) : $\mathbb{Z}$ {
`requiere:` $\{True\}$
`asegura:` $\{(x < y \rightarrow result = x) \wedge (x \geq y \rightarrow result = y)\}$
}

Un programador ha escrito el siguiente programa para implementar la especificación descripta:

```
def min (x: int, y: int) -> int:
L1:     result: int = 0
L2:     if x < y:
L3:         result = x
        else:
L4:         result = x
L5:     return result
```

Y el siguiente conjunto de casos de test (test suite):

- minA:
  - Entrada x=0,y=1
  - Salida esperada 0
- minB:
  - Entrada x=1,y=1
  - Salida esperada 1

1. Describir el diagrama de control de flujo (control-flow graph) del programa `min`.
2. ¿La ejecución del test suite resulta en la ejecución de todas las líneas del programa `min`?
3. ¿La ejecución del test suite resulta en la ejecución de todas las decisiones (branches) del programa?
4. ¿Es el test suite capaz de detectar el defecto de la implementación del problema de encontrar el mínimo?
5. Agregar nuevos casos de tests y/o modificar casos de tests existentes para que el test suite detecte el defecto.

Ejercicio 3. ★ Sea la siguiente especificación del problema de sumar y una posible implementación en lenguaje imperativo.

```
problema sumar (in x: ℤ, in y: ℤ) : ℤ {
      requiere: {True}
      asegura: {result = x + y}
}
```

```
def sumar (x: int, y: int) -> int:
L1:    result: int = 0
L2:    result = result + x
L3:    result = result + y
L4:    return result
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `sumar`.
2. Escribir un conjunto de casos de test (o "test suite") que ejecute todas las líneas del programa `sumar`.

Ejercicio 4. Sea la siguiente especificación del problema de restar y una posible implementación en lenguaje imperativo:

```
problema restar (in x: ℤ, in y: ℤ) : ℤ {
      requiere: {True}
      asegura: {result = x - y}
}
```

```
def restar (x: int, y: int) -> int:
L1:    result: int = 0
L2:    result = result + x
L3:    result = result + y
L4:    return result
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `restar`.
2. Escribir un conjunto de casos de test (o "test suite") que ejecute todas las líneas del programa `restar`.
3. La línea L3 del programa `restar` tiene un defecto, ¿es el test suite descripto en el punto anterior capaz de detectarlo? En caso contrario, modificar o agregar nuevos casos de test hasta lograr detectarlo.

Ejercicio 5. Sea la siguiente especificación del problema de `signo` y una posible implementación en lenguaje imperativo:

```
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
```

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?

Ejercicio 6. Sea la siguiente especificación del problema de `signo` y una posible implementación en lenguaje imperativo:

```
problema fabs (in x: ℝ) : ℝ {
      requiere: {True}
      asegura: {result = |x|}
}
```

```
  def fabs(x: float) -> float:
  L1:    result: float = 0
  L2:    if x<0:
  L3:       result = −x
  L4:    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. Escribir un test suite que ejecute todas las posibles decisiones ("branches") del programa.
4. Escribir un test suite que ejecute todas las líneas del programa pero no ejecute todos las decisiones del programa.
5. ¿Los test suites de los puntos anteriores detectan el defecto en la implementación? De no ser así, modificarlos para que lo hagan.

Ejercicio 7. ★ Sea la siguiente especificación:

```
problema fabs (in x: ℤ) : ℤ {
      requiere: {True}
      asegura: {result = |x|}
}
```

Y la siguiente implementación:

```
def fabs(x: int) -> int:
L1:    if x < 0:
L2:       return −x
       else:
L3:       return +x
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `fabs`.

2. Escribir un test suite que ejecute todas las líneas y todos los branches del programa.

Ejercicio 8. ★ Sea la siguiente especificación del problema de `mult10` y una posible implementación en lenguaje imperativo:

```
problema mult10 (in x: ℤ) : ℤ {
      requiere: {True}
      asegura: {result = x * 10}
}
```

```
def mult10(x: int) -> int:
L1:   result: int = 0
L2:   count: int = 0
L3:   while(count < 10):
L4:       result = result + x
L5:       count = count + 1
L6:   return result
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `mult10`.
2. Escribir un test suite que ejecute todas las líneas del programa `mult10`.
3. ¿El test suite anterior ejecuta todas las posibles decisiones ("branches") del programa?

Ejercicio 9. Sea la siguiente especificación del problema de `sumar` y una posible implementación en lenguaje imperativo:

```
problema sumar (in x: ℤ, in y: ℤ) : ℤ {
      requiere: {True}
      asegura: {result = x + y}
}
```

```
def sumar(x: int , y: int) -> int:
L1:    sumando: int = 0
L2:    abs_y: int = 0
L3:    if y < 0:
L4:      sumando = −1
L5:       abs_y = −y
       else:
L7:      sumando = 1
L8:      abs_y = y
L9:    result: int = x
L10:   count: int = 0
L11:   while(count < abs_y):
L12:       result = result + sumando
L13:       count = count + 1
L14:   return result
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `sumar`.
2. Escribir un test suite que ejecute todas las líneas del programa `sumar`.
3. Escribir un test suite que ejecute todas las posibles decisiones ("branches") del programa.

Ejercicio 10. Sea el siguiente programa que computa el máximo común divisor entre dos enteros.

```
def mcd(x: int , y: int) -> int:
L1:   assert (x >= 0 & y >= 0) # requiere: x e y tienen que ser no negativos
L2:   tmp: int = 0
L3:   while(y != 0):
L4:     tmp = x % y
L5:     x = y
L6:     y = tmp
L7:   return x
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `mcd`.
2. Escribir un test suite que ejecute todas las líneas y todos los branches del programa.

Ejercicio 11. ★ Sea el siguiente programa que retorna diferentes valores dependiendo si a, b y c, definen lados de un triángulo inválido, equilátero, isósceles o escaleno.

```
def triangle(a: int , b: int, c: int) -> int:
L1:     if(a <= 0 | b <= 0 | c <= 0):
L2:         return 4 # invalido
L3:     if(not ((a + b > c) & (a + c > b) & (b + c > a))):
L4:         return 4 # invalido
L5:     if(a == b & b == c):
L6:         return 1 # equilatero
L7:     if(a == b | b == c | a == c):
L8:         return 2 # isosceles
L9:     return 3  # escaleno
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `triangle`.
2. Escribir un test suite que ejecute todas las líneas y todos los branches del programa.

Ejercicio 12. ★ Sea la siguiente especificación del problema de `multByAbs` y una posible implementación en lenguaje imperativo:

`problema multByAbs` (in x: $\mathbb{Z}$, in y:$\mathbb{Z}$, out result: $\mathbb{Z}$) {
`requiere:` $\{True\}$
`asegura:` $\{result = x * |y|\}$
}

```
def multByAbs(x: int , y: int) -> int:
L1:     abs_y: int = fabs(y); # ejercicio anterior
L2:     if abs_y < 0:
L3:         return -1
L4:     else:
L5:         result: int = 0;
L6:         i: int = 0;
L7:         while i < abs_y:
L8:             result = result + x;
L9:             i += 1
L10:     return result
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `multByAbs`.
2. Detallar qué líneas y branches del programa no pueden ser cubiertos por ningún caso de test. ¿A qué se debe?
3. Escribir el test suite que cubra todas las líneas y branches que puedan ser cubiertos.

Ejercicio 13. Sea la siguiente especificación del problema de `vaciarSecuencia` y una posible implementación en lenguaje imperativo:

`problema vaciarSecuencia` (inout s: $seq\langle\mathbb{Z}\rangle$) {
`requiere:` {True}
`modifica:` {s}
`asegura:` $\{|s| = |s@pre| \land (\forall j : \mathbb{Z})(0 \leq j < |s| \rightarrow_L s[j] = 0)\}$
}

```
def vaciarSecuencia(s: [int]):
L1:     for i in range(len(s)):
L2:         s[i] = 0
```

1. Escribir el diagrama de control de flujo (control-flow graph) del programa `vaciarSecuencia`.
2. Escribir un test suite que cubra todos las líneas de programa.
3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.

Ejercicio 14. Sea la siguiente especificación del problema de `existeElemento` y una posible implementación en lenguaje imperativo:

```
problema existeElemento (s: seq⟨ℤ⟩, e: ℤ) : Bool {
    requiere: {True}
    asegura: {result = True ↔ (∃j : ℤ)(0 ≤ j < |s| ∧ s[j] = e)}
}
```

```
def existeElemento(s: [int], e: int) -> bool:
L1:     result: bool = False
L2:     for i in range(len(s)):
L3:         if s[i] == e:
L4:             result = True
L5:             break
L6:     return result
```

1. Escribir el diagrama de control de flujo (control-flow graph) del programa `existeElemento`.
2. Escribir un test suite que cubra todos las líneas de programa (observar que un `for` contiene 3 líneas distintas)
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. Sea la siguiente especificación del problema de `cantidadDePrimos` y una posible implementación en lenguaje imperativo:

```
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
```

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 16. Sea la siguiente especificación del problema de `esSubsecuencia` y una posible implementación en lenguaje imperativo:

in s: $seq\langle \text{Char} \rangle \mathbb{Z}$

`problema esSubsecuencia` (s: $seq\langle \mathbb{Z} \rangle$, r: $seq\langle \mathbb{Z} \rangle$) : Bool {
`requiere:` $\{True\}$
`asegura:` $\{result = True \leftrightarrow |r| \leq |s|$
$\wedge_L (\exists i : \mathbb{Z})((0 \leq i < |s| \wedge i + |r| < |s|) \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |r| \rightarrow_L s[i + j] = r[j])))\}$
}

```
1  def esSubsecuencia(s: [int], r: [int]) -> bool:
2      result: bool = False
3      ultimoIndice: int = len(s) - len(r)
4      for i in range(ultimoIndice + 1):
5          # obtener una subsecuencia de s
6          subseq: [int] = subsecuencia(s, i, len(r))
7          # chequear si la subsecuencia es igual a r
8          sonIguales: [int] = iguales(subseq, r)
9          if sonIguales:
10             result = True
11             break
12     return result
13
14 # procedimiento auxiliar subsecuencia
15 def subsecuencia(s: [int], desde: int, longitud: int) -> [int]:
16     rv: [int] = []
17     hasta: int = desde + longitud
18     for i in range(desde, hasta):
19         elem = s[i]
20         rv.append(elem)
21     return rv
22
23 # procedimiento auxiliar iguales
24 def iguales(a: [int], b: [int]) -> bool:
25     result: bool = True
26     if len(a) == len(b):
27         for i in range(len(a)):
28             if a[i] != b[i]:
29                 result = False
30                 break
31     else:
32         result = False
33     return result
```

1. Escribir los diagramas de control de flujo (control-flow graph) para `esSubsecuencia` y las funcones auxiliares `subsecuencia` e `iguales`.
2. Escribir un test suite que cubra todos las líneas de programa *ejecutables* de todos los procedimientos. Observar que un `for` contiene 3 líneas distintas.
3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.

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"`

La especificación formal es la siguiente:

`problema calcularFragmentoMásLargo` (s: $seq\langle \text{Char} \rangle$) : $\mathbb{Z}$ {

requiere: $\{True\}$
asegura: $\{(\exists i, j : \mathbb{Z})(\texttt{esFragmento}(s, i, j) \land n \leq j - i) \land (\forall i, j : \mathbb{Z})(\texttt{esFragmento}(s, i, j) \longrightarrow n \geq j - i)\}$
}

pred esFragmento (s: $seq\langle \mathsf{Char} \rangle$, $i, j : \mathbb{Z}$) {
$0 \leq i \leq j \leq |s| \land_L (\forall k : \mathbb{Z})(i \leq k < j \longrightarrow_L \neg \mathsf{esPuntoYComa}(s[k]))$
}
pred esPuntoYComa (c: Char) {
$c = \text{';'}$
}

El programa es el siguiente:

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

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.