
Universidad Nacional de Córdoba


Facultad de Matemática, Astronomía, Física y Computación

Facultad de Matemática, Astronomía, Física y Computación

# Laboratorio 4:

# Propiedades de los lenguajes de programación

Autores:
Mario E. Ferreyra
Alan G. Bracco

---

Docentes:
Laura Alonso Alemany
Cristian Cardellino
Ezequiel Orbe

---

Asignatura:
Paradigmas de la Programación

15 de Mayo de 2015

# Índice

# 1. Test Nº1

## 1.1. Python

**Código**

```
print("Entre")
a = 3
a = a + "casa"
print("Sali")
```

**Salida del Programa**

```
Entre
Traceback (most recent call last):
    File "estatico_vs_dinamico.py", line 4, in <module>
        a = a + "casa"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

**Explicación**

Python es de tipado dinámico, ya que este realiza los chequeos de tipo durante el tiempo de ejecución. Como se ve en la salida del programa, realiza el printeo del string "Entrez después tira un error, ya que queremos sumar un entero y un string.

## 1.2. Haskell

**Código**

```
tipado x y = x + y
ejecutor = tipado "hola" 4
```

**Salida del Programa**

```
estatico_vs_dinamico.hs:3:12:
    No instance for (Num [Char]) arising from a use of 'tipado'
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: tipado "hola" 4
    In an equation for 'ejecutor': ejecutor = tipado "hola" 4
Failed, modules loaded: none.
```

**Explicación**

Haskell es de tipado estático ya que al compilar el programa se da cuenta de que intentamos sumar dos variables de distinto tipo, como lo son un string y un entero, por lo que nos tira el error mostrado en la salida del programa.

## 1.3. Java

**Código**

```
public class miPrimeraClase {
    public static void main(String []args) {
        System.out.println("Entre'\n");
        int var = 0;
        String palabra = "Esto es horrible";
        var = var + palabra;
        System.out.println("Sali\n");
    }
}
```

**Salida del Programa**

```
estatico_vs_dinamico.java:1: error: class miPrimeraClase is public,
                                    should be declared in a file
named miPrimeraClase.java
public class miPrimeraClase
    ^
estatico_vs_dinamico.java:11: error: incompatible types
    var = var + palabra;
          ^
required: int
found: String
2 errors
```

**Explicación**

Java es de tipado estático, porque realiza el chequeo de tipos durante el tiempo de compilación, en el código, intentamos sumar un entero y un string, y al querer compilar el programa, nos tira el error de que los tipos son incompatibles.

## 1.4. Scala

**Código**

```
object HolaMundo {
    def main(args: Array[String]) {
        println("Entre")
        var a = 10
        var b = "Soy un koala"
        var c = a ++ b
        println(c)
        println("Sali")
    }
}
```

**Salida del Programa**

```
/home/mario/Escritorio/PARADIGMAS/lab_04/lang-tests/test-1
/estatico_vs_dinamico.scala:8:
error: value ++ is not a member of Int
    var c = a ++ b
            ^
one error found
```

**Explicación**

Scala tiene tipado estático, ya que el querer compilar el programa, salta un error de querer concatenar un entero con un string, y no se realiza el *print* de "Entre", por tal razón decimos que Scala es de tipado estático.

## 1.5. C

**Código**

```
#include <stdio.h>

int main() {
    printf("Entre\n");
    int var = 10;
    char var2[10] = "Hola Mundo";
    var = var + var2;
    printf("Afuera\n");
    return 0;
}
```

**Salida del Programa**

```
estatico_vs_dinamico.c: In function 'main':
estatico_vs_dinamico.c:11:9: warning: assignment makes integer
                                      from pointer without a cast
[enabled by default]
var = var + var2;
    ^
```

**Explicación**

C es de tipado estático, ya que el chequeo de tipos se realiza durante el tiempo de compilación, el error proviene de querer sumar un string y un entero, por lo que C lo detecta durante la compilación del programa.

## 1.6. Javascript

**Código**

```
var x = 1
console.log(x)
var y = "hola"
console.log(y)
x = x + y
console.log(x)
```

**Salida del Programa**

```
hola
1hola
```

**Explicación**

Javascript no tira error de ningún tipo por lo que permite que variables de distinto tipo como, por ejemplo un entero y un string sean "sumados". Javascript tiene tipado dinámico.

## 1.7. Ruby

**Código**

```
puts "Entre"
a = 10
b = "Me llamo silla"
a = a + b
puts "Sali"
```

**Salida del Programa**

```
Entre
estatico_vs_dinamico.rb:4:in `+': String can't be coerced
                                  into Fixnum (TypeError)
    from estatico_vs_dinamico.rb:4:in `<main>'
```

**Explicación**

Ruby es de tipado dinámico ya que el error de tipos de querer sumar un entero y un string se da después de que haya hecho el print de "Entre".

# 2. Test Nº2

**Tipado Fuerte:** Un lenguaje de programación es fuertemente tipado si no se permiten violaciones de los tipos de datos, es decir, dado el valor de una variable de un tipo concreto, no se puede usar como si fuera de otro tipo distinto a menos que se haga una conversión.

**Tipado Debil:** Los lenguajes de programación no tipados o débilmente tipados no controlan los tipos de las variables que declaran, de este modo, es posible usar variables de cualquier tipo en un mismo escenario.

## 2.1. Python

**Código**

```
def types(a):
    c = 'f'
    print(a + c)

types(2)
```

**Salida del Programa**

```
Traceback (most recent call last):
    File "fuerte_vs_debil.py", line 7, in <module>
        types(2)
    File "fuerte_vs_debil.py", line 5, in types
        print(a + c)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

**Explicación**

Python es de tipado fuerte, por lo que no permite realizar la operación de sumar un entero con un string (o char en este caso), es por eso que tira el error, ya que no existe una conversión previa de los tipos.

## 2.2. Haskell

**Código**

```
types :: Int -> Int
types a = let c = 'f' in
          a + c
```

**Salida del Programa**

```
Couldn't match expected type 'Int' with actual type 'Char'
    In the second argument of '(+)', namely 'c'
    In the expression: a + c
    In the expression: let c = 'f' in a + c
```

**Explicación**

Haskell es de tipado fuerte, es por eso que ocurre lo mismo que con Python, y lanza un error al querer realizar una operación aritmética entre un entero y un char, ya que no realiza una conversión implícita de tipos.

## 2.3. Java

**Código**

```
class fuerte_vs_debil {
    public static void main(String []args) {
        int a = 3.5;
        System.out.println(a);
    }
}
```

**Salida del Programa**

```
fuerte_vs_debil.java:5: possible loss of precision
found : double
required: int
    int a = 3.5;
            ^
1 error
```

**Explicación**

Java es de tipado fuerte, es por eso que si defino a la variable 'a' como int y le pongo un valor de un tipo double, lanza un error de tipo ya que no realiza la conversión.
Mas adelante veremos que en C esto es posible, debido a su tipado débil.

## 2.4. Scala

**Código**

```
object fuertevsdebil {
    def main(args: Array[String]) {
        val b : Double = 2
        var c : Int = types(b)
        println(c)
    }

    def types(a: Int) : Int = a + 3
}
```

**Salida del Programa**

```
fuerte_vs_debil.scala:5: error: type mismatch;
found : Double
required: Int
    var c : Int = types(b)
                  ^
one error found
```

**Explicación**

Scala es de tipado fuerte, por lo que en este caso no permite poder ingresar un argumento de tipo double cuando el parámetro es de tipo int, y esto es así porque no realiza la conversión implícita de tipos, y eso por eso que lanza un error de tipos.

## 2.5. C

**Código**

```c
#include <stdio.h>

    float g(float m) {
        return m + 1.5;
    }

    int main(void) {
        int a = 2.3;
        float c = 2.3;
        char b = 'm';
        c = a + b + c;
        printf("%f\n", g(a));
        printf("%f\n", c);
        return 0;
    }
```

**Salida del Programa**

```
3.500000
113.300003
```

**Explicación**

C es de tipado débil, es por eso que el primer resultado impreso resulta de declarar una variable int con un valor de un tipo float, y puede realizar una conversión implícita de tipos, luego de realizar la función puede retornar el valor como si fuera un float y no como un int que fue declarado originalmente. Además se realiza otra prueba obteniendo el segundo resultado luego de sumar tres variables de tres tipos distintos (int, float y char), y al ser un lenguaje débilmente tipado, puede realizar estas operaciones.

## 2.6. Javascript

**Código**

```
<!DOCTYPE html>
<html>
    <title> fuerte_vs_debil </title>
    <head>
        <script>
            var a = 20
            var c = "f"
            var b = ["hola" ,2]
            var result = a + c + b
            var result2 = a - c - b
            document.write(result)
            document.write(result2)
        </script>
    </head>
    <body>
    </body>
</html>
```

**Salida del Programa**

```
20fhola ,2NaN
```

**Explicación**

Javascript permite las operaciones entre tipos distintos, por lo que es débilmente tipado.
En este caso se obtiene un resultado de sumar un tipo int, un tipo char y una lista.
El segundo resultado se obtiene restando los elementos anteriores.
Al ser débilmente tipado, no lanza errores y realiza las conversiones internamente.

## 2.7. Ruby

**Código**

```
def fuerte_vs_debil(a)
    c = a + "hola"
    print c
end

fuerte_vs_debil(2)
```

**Salida del Programa**

```
fuerte_vs_debil.rb:3:in `+': String can't be coerced
                             into Fixnum (TypeError)
    from fuerte_vs_debil.rb:3:in `fuerte_vs_debil'
    from fuerte_vs_debil.rb:8:in `<main>'
```

**Explicación**

Ruby es de tipado fuerte, es por eso que no permite realizar una operación de suma entre una variable de tipo int y otra de tipo string.
No realiza la conversión implícita y por eso lanza un error de tipo.

# 3. Test Nº3

**Asignación Única:** Una vez declarado un identificador con un valor, lo mantiene durante todo el programa.

## 3.1. Python

**Código**

```
def asignacion():
    x = 1
    print("x vale " + str(x))
    x = 2
    print("ahora x vale " + str(x))

asignacion()
```

**Salida del Programa**

```
x vale 1
ahora x vale 2
```

**Explicación**

Python no es de asignación única por lo que tiene variables que cambian sus valores durante la ejecución del programa.

## 3.2. Haskell

**Código**

```
asignacion :: Int -> Int
asignacion x = let x = 1 in
        let x = x + 5 in
        x + 2
```

**Salida del Programa**

```
*Main> asignacion 3
*** Exception: <<loop>>
```

**Explicación**

El programa compila correctamente, pero luego de llamar a la función, lanza una excepción ya que es de asignación única, y al querer realizar la segunda asignación, no puede realizarla porque se define dependiendo del valor tomado anteriormente.

## 3.3. Java

**Código**

```
public class asignacion {
    public static void main(String []args) {
        int a = 0;
        System.out.println("a vale: " + a);
        a = 5;
        System.out.println("ahora a vale: " + a);
    }
}
```

**Salida del Programa**

```
a vale: 0
ahora a vale: 5
```

**Explicación**

Java no es de asignación única y eso queda demostrado luego de inicializar la variable a imprimirla y luego cambiarle su valor e imprimirla nuevamente, y podemos ver que en ambas impresiones los valores son diferentes.

## 3.4. Scala

**Código**

```
object asignacion {
    def main(args: Array[String]) {
        var a = 10
        println(a)
        a = 20
        println(a)
    }
}
```

**Salida del Programa**

```
10
20
```

**Explicación**

Scala no tiene asignación única, es por eso que podemos modificar el valor de 'a' e imprimir dos valores distintos.

## 3.5. C

**Código**

```
#include <stdio.h>

int main(void){
    int a = 50;
    printf("'a' ahora vale %i\n", a);
    a = 30;
    printf("'a' ahora vale %i\n", a);
    return 0;
}
```

**Salida del Programa**

```
'a' ahora vale 50
'a' ahora vale 30
```

**Explicación**

C no tiene asignación única, pudiéndose ver esto que el valor de 'a' es distinto en las dos impresiones por consola.

## 3.6. Javascript

**Código**

```
<!DOCTYPE html>
<html>
    <title> asignacion </title>
    <head>
        <script>
            var a = 20
            document.write("'a' vale: " + a)
            a = 30
            document.write(" y ahora 'a' vale: " + a)
        </script>
    </head>
    <body>
    </body>
</html>
```

**Salida del Programa**

```
'a' vale 20 y ahora 'a' vale 30
```

**Explicación**

Javascript no es de asignación única, es por eso que 'a' cambia de valor durante la ejecución y se imprimen distintos valores.

## 3.7. Ruby

**Código**

```
def asignacion()
    x = 1
    print "x vale #{x}"
    x = 2
    print "\nahora x vale #{x} \n"
end

asignacion()
```

**Salida del Programa**

```
x vale 1
ahora x vale 2
```

**Explicación**

Ruby no es de asignación única, es por esto que x toma distintos valores durante la ejecución, y se muestran dos valores distintos.

# 4. Test Nº4

**Alcance estático:** El valor de las variables globales se obtiene del bloque inmediatamente contenedor.

**Alcance dinámico:** El valor de las variables globales se obtiene del activation record mas reciente.

## 4.1. Python

**Código**

```
x = 200

def alcance(n):
    def g(n):
        return x+n
    return g(n)
x = 3
z = alcance(0)
print z
```

**Salida del Programa**

```
3
```

**Explicación**

Si fuera de alcance estático imprimiría el valor de 200, pero imprime el valor de x como 3, es por eso que es de alcance dinámico, ya que x = 3 es de la ultima asignación de x.

## 4.2. Haskell

**Código**

```
f = let x = 10 in (\z -> x + z)

g = let h = f
    x = 15
  in show $ h 1
```

**Salida del Programa**

```
"11"
```

**Explicación**

Luego de llamar a g, imprime el valor 11, por lo que tiene alcance dinámico, apuntando al activation record mas cercano. De lo contrario imprimiría el valor 15.

## 4.3. Java

**Código**

```
public class alcance {
    static int a = 5;
    public static int g() {
        return a;
    }

    public static void main(String []args) {
        int a = 0;
        int b = g();
        System.out.println("b vale: " + b);
    }
}
```

**Salida del Programa**

```
b vale: 5
```

**Explicación**

Es de alcance estático, ya que imprime el primer valor de 'a' que fue declarado previamente como 5.
Al no ser dinámico, la sentencia 'int a = 0' no tiene efectos sobre la función g.
Queda así demostrado que es de alcance estático.

## 4.4. Scala

**Código**

```
object alcance {
    def main(args: Array[String]) {
        var a : Int = g()
        println(a)
    }

    var x = 2

    def f(): Int ={
        return x
    }

    def g(): Int ={
        var x = 3
        return f()
    }
}
```

**Salida del Programa**

```
2
```

**Explicación**

Es de alcance estático, ya que la x impresa no muestra el ultimo valor del ultimo activation record, si no del bloque envolvente mas cercano, donde x vale 2.

## 4.5. C

**Código**

```
#include <stdio.h>

int x = 0;

int g(void) {
    return x;
}

int main(void){
    int x = 1;
    if (g()){
        printf("dinamico\n");
    }
    else {
        printf("estatico\n");
    }
    return 0;
}
```

**Salida del Programa**

```
estatico
```

**Explicación**

Si x vale 1 imprime “estático”, si vale 0 imprime “dinámico”, pero al ser de alcance estático, toma el valor declarado en el bloque inmediatamente contenedor, donde el valor es 1, es por eso que luego imprime “estático”.

## 4.6. Javascript

**Código**

```
<!DOCTYPE html>
<html>
    <title> alcance </title>
    <head>
        <script>
            var a = 20
            if (true) {
                var a = 35
                document.write("'a' dentro del bloque vale: " + a)
            }
            document.write(" y 'a' fuera del bloque vale: " + a)
        </script>
    </head>
    <body>
    </body>
</html>
```

**Salida del Programa**

```
'a' dentro del bloque vale: 35 y 'a' fuera del bloque vale: 35
```

**Explicación**

En el segundo *print* se queda con el ultimo valor de a, es por eso que es de alcance dinámico, ya que toma el valor del activation record mas reciente.

## 4.7. Ruby

**Código**

```
$x = 200
def alcance()
    return $x
end

def main()
    $x = 3
    z = alcance()
    puts z
end

main()
```

**Salida del Programa**

```
3
```

**Explicación**

Es de alcance dinámico, ya que imprime el valor 3 en lugar de 200.
Toma el valor del activation record mas cercano, que es en el que x tiene el valor impreso.

# 5. Test Nº5

## 5.1. Python

**Código**

```
def g(n):
    if n == 0: return 1/0
    else: return g(n-1)

g(10)
```

**Salida del Programa**

```
Traceback (most recent call last):
    File "recursion.py", line 10, in <module>
        g(10)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 7, in g
        else: return g(n-1)
    File "recursion.py", line 6, in g
        if n == 0: return 1/0
ZeroDivisionError: integer division or modulo by zero
```

**Explicación**

Python no realiza la optimización de la llamada a la cola, ya que al momento en el que el método llega a la parte del programa en la cual tiene que dividir por 0, salta una excepción y nos muestra en el stack, por lo que si tuviera la optimización a la cola, mostraría un solo bloque y no varios

## 5.2. Haskell

**Código**

```
suma_rara x | x == 0 = 1
            | otherwise = 10 + suma_rara(x - 1)
```

**Salida del Programa**

```
Hacemos esto en consola:
    $ ghci recursion.hs
    <<< Abre GHCI y carga compila el archivo recursion.hs >>>
    Si hacemos llamamos a suma_rara de esta manera:
    > suma_rara 50000000

Obtendremos los siguiente:
    ***Exception: stack overflow
```

**Explicación**

Al ejecutar esa función en Haskell, la computadora tardo un buen tiempo en computar dicho valor, por lo tanto podes observar que Haskell tiene una forma de optimiza el valor del stack, pero llega un punto en el que el stack se desborda y ocurre el "stack overflow", por lo que concluimos que Haskell no tiene la optimización de la llamada a la cola.

## 5.3. Java

**Código**

```
public class recursion {
    public static void main(String []args) {
        mario(10);
    }

    static int mario(int x) {
        if (x == 0) {
            return 1/x;
        }
        else {
            return mario(x-1);
        }
    }
}
```

**Salida del Programa**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at recursion.mario(recursion.java:14)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.mario(recursion.java:18)
    at recursion.main(recursion.java:7)
```

**Explicación**

Java no realiza la optimización de la llamada a la cola. Es el mismo argumento que los demás test's de este tipo, al momento que llega la parte en la que tiene que dividir por 0, salta una excepción y nos muestra el stack con varios bloques, y no uno solo, por lo que no realiza la optimización antes nombrada.

## 5.4. Scala

**Código**

```
object recursion {
    def main(args: Array[String]) {
        var x = 0 // Cambiar por x = 1, si se quiere ver
                  // como seria el stack con un solo bloques
        try {
            if (x == 1) f(10) else g(10)
        }
        catch {
            case e: Exception => e.printStackTrace()
        }
    }

    // Funcion Recursiva, tendria que tener 1 bloque f
    def f(n : Int) {
        if (n == 0) 1/0 else f(n-1)
    }

    // Funcion NO Recuersiva, tendria que tener 10 bloques g
    def g(n : Int): Int = {
        if (n == 0) 1/0 else 1 + g(n-1)
    }
}
```

**Salida del Programa**

```
Con x = 0:
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.g(recursion.scala:27)
    at Main$.main(recursion.scala:10)
    at Main.main(recursion.scala)

Con x = 1:
    at Main$.f(recursion.scala:21)
    at Main$.main(recursion.scala:10)
    at Main.main(recursion.scala)
```

**Explicación**

Scala si tiene la optimización de la llamada a la cola, como vemos en el programa cuando quiera dividir por 0, se captura la excepción y muestra el stack, analicemos los casos en que x = 1 y x = 0.
Si x = 1, quiere decir que se llamara a una función la cual se llama así misma en una parte del código, por lo que al ver el stack vemos un solo bloque de la función f, por lo que concluimos que Scala si tiene la optimización.
Si x = 0, quiere decir que se llamara a una función la cual se también se llama así misma pero con la diferencia que se le suma 1, por lo que se usaran distintos Activation Records, y serán tantos como el parámetro pasado a la función.

## 5.5. C

**Código**

```
#include <stdio.h>

int main() {
    g(100000000000000);
    return 0;
}

int g(int x) {
    if (x == 0) {
        return 0;
    }
    else {
        return g(x - 1);
    }
}
```

**Salida del Programa**

```
Con Flag:
    No hay nada en la salida, ya que en el programa
    no se hace ninguna accion de stdout

Sin Flag:
    Violacion de segmento ('core' generado)
```

**Explicación**

C no tiene optimización de llamada a la cola, pero si le pasamos el flag -O2 (quedaría así a la hora de compilar gcc -O2 -o NOMBRE_EJECUTABLE FILE.c) Hace que el compilador de C haga la optimización de la llamada a la cola. Con el siguiente programa, sin ese flag, llega un punto en el que el stack "explota", por lo que tira una "Violación de Segmento", en cambio si colocamos ese flag al compilar no tirará ningún error.

## 5.6. Javascript

**Código**

```
function mario(x) {
    if (x == 0) {
        1/x;
        console.log(x);
    }
    else {
        mario(x -1);
        console.log(x);
    }
}

mario(1000000000000000);
```

**Salida del Programa**

```
Uncaught RangeError: Maximum call stack size exceeded
```

**Explicación**

Como vemos a la salida del programa, Javascript no realiza la optimización de la llamada a la cola, ya que la salida dice que "El tamaño del stack fue excedido".

## 5.7. Ruby

**Código**

```
def g(n)
    if n == 0
        1/n
    else
        g(n-1)
    end
end

g(10)
```

**Salida del Programa**

```
recursion.rb:5:in '/': divided by 0 (ZeroDivisionError)
    from recursion.rb:5:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:7:in 'g'
    from recursion.rb:12:in '<main>'
```

**Explicación**

Como podemos ver en la salida del programa, vemos que al llegar a la parte en la cual tiene que dividir por 0 salta una excepción y nos muestra el stack con las llamadas recursivas de la función g. Por lo que Ruby no realiza la optimización de la llamada a la cola.

# 6. Test Nº6

**Código**

```
object alto_orden {
    def main(args: Array[String]) {
        val y = () => { println ("no lazy") ; 10}
        println (y())
        println (y())

        lazy val x = { println ("lazy") ; 5}
        println (x)
        println (x)
        println ()
    }
}
```

**Salida del Programa**

```
no lazy
10
no lazy
10
lazy
5
5
```

**Explicación**

Al no declararse como lazy la primer variable, vuelve a calcular su valor cada vez que es llamada, es por eso que imprime los mensajes del *print*.
La segunda variable es declarada como lazy, siendo que su valor es computado solo la primera vez, y luego en las siguientes llamadas no vuelve a calcularse su valor, por lo que el mensaje del *print* lo imprime una sola vez.

# 7. Test Nº7

**Código**

```
object Pasaje {
    def main(args: Array[String]) {
        val x = 10
        val y = 100
        call_by_value({println ("HOLA"); x})
        call_by_name({println ("CHAU"); y})
    }

    def call_by_value (x: Int) {
        println("AGUA")
        println(x)
        println("\n")
    }

    def call_by_name (y: => Int) // => : hace que sea pasaje por nombre
                                       en vez de por valor
                                       (por defecto en scala) {
        println("FUEGO")
        println(y)
        println("\n")
    }
}
```

**Explicación**

Al llamar a call_by_value (Al ser por Valor), lo primero que hace es Imprimir:

HOLA

AGUA

10

Porque ejecuta los parámetro pasado a la función, en el orden adecuado y después el cuerpo de la función llamada.

Al llamar a call_by_name (Al ser por Nombre), lo primero que hace es Imprimir:

FUEGO

CHAU

100

Porque ejecuta el parámetro pasado a la función cuando lo necesita, por lo que primero imprime FUEGO y después ejecuta lo pasado a dicha función cuando es llamada.

# 8. Test Nº8

## 8.1. C

**Código**

```
#include <stdio.h>

void interchange_reference(int *x1, int *y1) {
    int z1;
    z1 = *x1;
    *x1 = *y1;
    *y1 = z1;
}

void interchange_value(int x1, int y1) {
    int z1;
    z1 = x1;
    x1 = y1;
    y1 = z1;
}

int main() {
    int x = 50, y = 70;
    int change;

    printf("\nIngrese '0' si quiere Pasaje por Valor
                \nIngrese '1' si quiere Pasaje por Referencia
                \nIngrese la opcion: ");
    scanf("%d", &change);

    printf("\nANTES DE interchange: x = %d, y = %d\n", x, y);
    if (change == 0) {
        interchange_value(x, y);
    }
    else if (change == 1) {
        interchange_reference(&x, &y);
    }

    printf("\nDESPUES DE interchange: x = %d, y = %d\n\n", x, y);
    return 0;
}
```

**Salida del Programa**

```
Si la opcion es 0:
    Ingrese '0' si quiere Pasaje por Valor
    Ingrese '1' si quiere Pasaje por Referencia
    Ingrese la opcion: 0
    ANTES DE interchange: x = 50, y = 70
    DESPUES DE interchange: x = 50, y = 70

Si la opcion es 1:
    Ingrese '0' si quiere Pasaje por Valor
    Ingrese '1' si quiere Pasaje por Referencia
    Ingrese la opcion: 1
    ANTES DE interchange: x = 50, y = 70
    DESPUES DE interchange: x = 70, y = 50
```

**Explicación**

C por defecto tiene pasaje por valor, el pasaje por referencia se consigue pasando explícitamente las direcciones de memoria de dichos parámetros. Por lo que este programa quiere expresar las dos formas de pasaje de parámetro.
Vemos que si el pasaje se realiza por valor, el valor de las variables al pasar por el procedimiento "interchange"no cambias sus valores (Esto es porque C es de pasaje por valor por defecto).
En cambio si el pasaje se realiza por referencia los valores de las variables cambian después de pasar por el procedimiento "interchange", ya que se accede a sus direcciones de memoria y se modifica el valor que estaba alojado en esa celda de memoria.

## 8.2. Perl

**Código**

```
$x = 50;
$y = 70;

print "\nANTES DE INTERCHANGE: x = $x, y = $y\n";
&interchange ($x, $y);
print "\nDESPUES DE INTERCHANGE: x = $x, y = $y\n\n";

sub interchange {
    ($x1, $y1) = @_; # x1 e y1 son variables locales que
                     # toman x e y respectivamente
    @_[0] = $y1;
    @_[1] = $x1;
    $x1=10;

    print "\nADENTRO DE INTERCHANGE (VALORES): x1 = $x1, y1 = $y1\n"
}
```

**Salida del Programa**

```
ANTES DE INTERCHANGE: x = 50, y = 70
ADENTRO DE INTERCHANGE (VALORES): x1 = 50, y1 = 70
DESPUES DE INTERCHANGE: x = 70, y = 50
```

**Explicación**

Perl tiene pasaje de parámetros por referencia, ya que al salir de la función “interchange”los valores de x e y se ven modificados.

# 9. Test Nº9

## 9.1. Java

**Código**

```
public class Point {
    public int x;
    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public static void tricky1(Point arg1, Point arg2) {
        arg1.x = 100;
        arg1.y = 100;
        Point temp = arg1;
        arg1 = arg2;
        arg2 = temp;
    }

    public static void tricky2(Point arg1, Point arg2) {
        arg1 = null;
        arg2 = null;
    }

    public static void main(String [] args) {
        Point pnt1 = new Point(0,0);
        Point pnt2 = new Point(0,0);
        System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y: " +pnt1.y);
        System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: " +pnt2.y);
        System.out.println("triki1\n");
        tricky1(pnt1,pnt2);
        System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y:" + pnt1.y);
        System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: " +pnt2.y);
        System.out.println("\ntriki2");
        tricky2(pnt1,pnt2);
        System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y:" + pnt1.y);
        System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: " +pnt2.y);
    }
}
```

**Salida del Programa**

```
pnt1 X: 0 pnt1 Y: 0
pnt2 X: 0 pnt2 Y: 0

triki1
pnt1 X: 100 pnt1 Y:100
pnt2 X: 0 pnt2 Y: 0

triki2
pnt1 X: 100 pnt1 Y:100
pnt2 X: 0 pnt2 Y: 0
```

**Explicación**

Java tiene pasaje de parámetros por Valor, por lo que podemos observar que tanto en la función "tricky1çomo en "tricky2"los valore de las pasados no se modifican (por ser pasaje de valor, se pasan copias de las variables, por lo que las originales no se modifican), pero hay algo que hay que tener en cuenta, en la función "tricky1"vemos esto:

- arg1.x = 100;
- arg1.y = 100;

Y al ver la salida del programa los valores de X e Y se modificaron, esto es porque estamos accediendo a los "campos"de arg1 (que se un Point) por lo que al acceder a los campos de arg1 estamos modificando a lo que apuntan estos, es decir, sus valores es sus celdas de memoria.