Examen 2

Lenguaje: V

Estudiante:
Angel Valero

Profesor:
Fernando Lovera

Pregunta 1

Dejo aquí el link a la carpeta del proyecto

(a) Dé una breve descripción del lenguaje escogido.

V es un lenguaje de programación de alto nivel y propósito general, diseñado para ser simple, rápido, seguro y fácil de mantener. Su sintaxis es similar a Go y Rust, pero con un enfoque minimalista. Está compilado y produce binarios muy pequeños.
V es fuertemente tipado, con inferencia de tipos, y puede usarse tanto para programación de sistemas como para aplicaciones de alto nivel (CLI, web, gráficos, etc.).

I. Enumere y explique las estructuras de control de flujo que ofrece.

V ofrece las estructuras tradicionales de control de flujo de los lenguajes imperativos:

- **Secuencia:** ejecución línea por línea de instrucciones.
- **Condicionales:**
  - *if / else if / else*
  - *match* (similar al *switch* de otros lenguajes).
- **Bucles:**
  - **for** es la única estructura de bucle, pero puede actuar como:
    - *for i := 0; i < n; i++ { ... } → bucle clásico.*
    - *for x in array { ... } → iteración sobre colecciones.*
    - *for condition { ... } → bucle while.*
- **Control de flujo adicional:**
  - *break y continue* para controlar la ejecución dentro de los bucles.
  - *return* para salir de funciones.

II. Diga en qué orden evalúan expresiones y funciones.

A. ¿Tiene evaluación normal o aplicativa? ¿Tiene evaluación perezosa?

El lenguaje V utiliza una estrategia de evaluación aplicativa, también conocida como call-by-value. Esto quiere decir que todas las expresiones y argumentos se evalúan completamente antes de que la función sea ejecutada. En otras palabras, V no aplaza la evaluación de los argumentos ni los reevalúa cada vez que se necesitan dentro de la función, sino que pasa los valores ya calculados.

Por otro lado, V no tiene evaluación perezosa (*lazy evaluation*) como Haskell o algunos lenguajes funcionales. Sin embargo, en mis pruebas pude simularla usando funciones anónimas (*closures*), que solo se ejecutan cuando se llaman explícitamente:

```
fn lazy_sum(f fn () int, g fn () int)
int {
    return f() + g()
}

fn main() {
    result := lazy_sum(fn () int {
return 2 + 3 }, fn () int { return 4 *
5 })
    println(result) // 25
}
```

Aquí los cálculos 2 + 3 y 4 * 5 no se evalúan hasta que la función **lazy_sum** los ejecuta.
Fue interesante ver que, aunque V no está pensado para evaluación perezosa, su diseño minimalista permite explorar ese comportamiento sin complicaciones.

B. La evaluación de argumentos/operandos se hace de izquierda a derecha, de derecha a izquierda o en un orden arbitrario.

En V, las expresiones y los argumentos de funciones se evalúan en orden de izquierda a derecha, de forma determinista y predecible. En una llamada a función o en una operación binaria primero se evalúa el operando o argumento del lado izquierdo, y luego el del derecho.

Esto evita comportamientos ambiguos o dependientes del compilador, como ocurre en C o C++. Gracias a esto, los efectos secundarios (como impresiones, operaciones de entrada/salida o modificaciones de variables) suceden siempre en el mismo orden en el que aparecen en el código, lo que facilita la depuración y la lectura.

En el caso de los operadores lógicos **&&** y **||**, V aplica evaluación con cortocircuito, es decir, el segundo operando solo se evalúa si el primero no determina por sí mismo el resultado de la expresión. Esto mejora la eficiencia y evita errores por evaluaciones innecesarias.

En resumen, V evalúa de izquierda a derecha, con un comportamiento consistente y seguro, y solo interrumpe la evaluación en los casos donde el lenguaje aplica cortocircuito lógico.

(b) Implemente los siguientes programas en el lenguaje escogido:

I. Considere la siguiente función:

$$f(n) = \begin{cases} n/2 & \text{si } n \text{ es par} \\ 3n+1 & \text{si } n \text{ es impar} \end{cases}$$

La función se trata de una secuencia de Collatz, este es el código para aplicarla en V.

```
import os

// Función que calcula la cantidad de pasos para llegar a 1
fn collatz_count(n u64) u32 {
    mut x := n          // variable mutable que representa el número actual
    mut steps := u32(0)   // contador de pasos

    for x != 1 {        // bucle que se repite mientras el número sea distinto de 1
        if x % 2 == 0 {       // si el número es par
            x /= 2             // se divide entre 2
        } else {              // si el número es impar
            x = 3 * x + 1      // se aplica la fórmula 3n + 1
        }
        steps++               // se incrementa el contador
    }
    return steps              // devuelve la cantidad total de pasos
}

// Programa principal
fn main() {
    input := os.input('Ingrese un número entero positivo: ').trim_space()
    n := input.u64()                      // conversión de texto a número sin signo
    println('Cantidad de pasos: ${collatz_count(n)}')
}
```

A continuación, desglosaré el código

Importación del módulo os**:**
En V, el módulo os contiene funciones de entrada y salida básicas. Aquí se usa para leer desde la consola (os.input()).

Función collatz_count**:**

- Recibe un número entero sin signo (u64), lo que evita problemas con números negativos.
- Usa variables mutables (mut) para poder modificarlas dentro del bucle.
- Implementa un ciclo **for** que se repite hasta que x sea igual a 1.
- Dentro del ciclo:
    - Si x es par, se divide entre 2.
    - Si x es impar, se aplica la fórmula 3n + 1.
- En cada iteración se incrementa un contador (**steps**), que representa la cantidad de veces que se aplica la función f(n) hasta llegar a 1.
- Al final, se retorna **steps**.

Función main**:**

- Es el punto de entrada del programa (como en C o Go).
- Pide al usuario un número, lo convierte a tipo u64 y lo pasa a la función **collatz_count**.
- Imprime el resultado final.

II. Implemente el algoritmo Mergesort y explique los detalles de su implementación

```
// Implementación del algoritmo Mergesort en
V
fn mergesort(mut a []int) {
    if a.len <= 1 {
        return
    }
    // Se divide el arreglo en dos mitades
    mid := a.len / 2
    mut left := a[..mid]
    mut right := a[mid..]
    // Ordenar recursivamente ambas mitades
    mergesort(mut left)
    mergesort(mut right)
    // Combinar los resultados en 'a'
    mut i := 0
    mut j := 0
    mut k := 0
    for i < left.len && j < right.len {
        if left[i] <= right[j] {
            a[k] = left[i]
            i++
        } else {
            a[k] = right[j]
            j++
        }
        k++
    }
    // Copiar los elementos restantes
    for i < left.len {
        a[k] = left[i]
        i++
        k++
    }
    for j < right.len {
        a[k] = right[j]
        j++
        k++
    }
}
fn main() {
    mut datos := [5, 2, 8, 3, 1, 7]
    println('Arreglo original: $datos')
    mergesort(mut datos)
    println('Arreglo ordenado:  $datos')
}
```

Estructura general:

fn mergesort(mut a []int):

Función principal que recibe un arreglo de enteros mutable.

if a.len <= 1: Caso base

mid := a.len/2: Calculo del punto medio para dividir

a[..mid] y a[mid..]: Creación de los dos subarreglos.

El funcionamiento es el básico de merge.

El arreglo se subdivide en dos mitades, repite esto hasta llegar a subarreglos de un solo elemento.

Cada mitad se ordena de manera independiente con llamadas recursivas a la misma función

Se fusionan las mitades en un solo arreglo ordenado comparando los elementos uno a uno, para esto uso los índices **i, j, k**, pues estas controlan las posiciones en los subarreglos **izquierdo, derecho y en el arreglo final,** respectivamente.

Si una mitad aún tiene elementos sin copiar (porque quizá la otra se agotó primero) se agregan al final.

```
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2> v run mergeS.v
mergeS.v:9:15: notice: an implicit clone of the slice was done here
    7 |     // Se divide el arreglo en dos mitades
    8 |     mid := a.len / 2
    9 |     mut left := a[..mid]
      |                 ~~~~~~~
   10 |     mut right := a[mid..]
   11 |
Details: mergeS.v:9:15: details: To silence this notice, use either an explicit `a[..].clone()
`,
or use an explicit `unsafe{ a[..] }`, if you do not want a copy of the slice.
    7 |     // Se divide el arreglo en dos mitades
    8 |     mid := a.len / 2
    9 |     mut left := a[..mid]
      |                 ~~~~~~~
   10 |     mut right := a[mid..]
   11 |
mergeS.v:10:16: notice: an implicit clone of the slice was done here
    8 |     mid := a.len / 2
    9 |     mut left := a[..mid]
   10 |     mut right := a[mid..]
      |                  ~~~~~~~
   11 |
   12 |     // Ordenar recursivamente ambas mitades
Details: mergeS.v:10:16: details: To silence this notice, use either an explicit `a[..].clone(
)`,
or use an explicit `unsafe{ a[..] }`, if you do not want a copy of the slice.
    8 |     mid := a.len / 2
    9 |     mut left := a[..mid]
   10 |     mut right := a[mid..]
      |                  ~~~~~~~
   11 |
   12 |     // Ordenar recursivamente ambas mitades
Arreglo original: [235, 124, 15, 1, 654, 4, 23, 9, 0, 87]
Arreglo ordenado:  [0, 1, 4, 9, 15, 23, 87, 124, 235, 654]
```

Este es el output con un ejemplo dado, se observa que el código tiene ciertas advertencias, estas se deben a que:

```
    mut left := a[..mid]          ←   Cuando ejecutamos
    mut right := a[mid..]         ←   estas lineas
```

el compilador crea copias independientes de esas porciones del arreglo, porque son mutables (mut) y V no permite que dos secciones distintas apunten al mismo bloque de memoria mutable sin control (por seguridad de memoria).

Entonces, para mantener la seguridad, V hace un “clone” implícito de esas partes.

Pero las advertencias no detienen la ejecución del código, y se puede ver que justo luego se ejecuta y funciona correctamente.

Pregunta 2

Estructura del código (expr.v)

1. Tipos y estructuras

- **enum Kind { num op }:** define el tipo de nodo (número u operador).
- **struct Node:** representa cada nodo del árbol de expresión con:
  - ***kind***: tipo (num o op).
  - ***val***: valor numérico (si es número).
  - ***op***: operador (si es op).
  - ***left, right***: punteros a nodos hijos **(&Node),** inicializados en unsafe { nil }.

2. Funciones auxiliares

- **tokenize(expr string):** separa la cadena de entrada en *tokens* (números y operadores).
- **is_op(t string):** devuelve true si el *token* es un operador (+ - * /).
- **prec(op string):** establece la precedencia de los operadores.
- **needs_paren(parent_op, child_op string, is_right bool):** decide si un subárbol necesita paréntesis al generar la expresión infija.

3. Parsers

- **parse_prefix(mut ts TStream):** construye recursivamente un árbol desde notación prefija (usa opciones **?string** y retorna errores si faltan *tokens*).
- **parse_postfix(tokens []string):** construye un árbol iterativamente usando una **pila**.
- **parse(order, expr_tokens):** selecciona el parser según el modo (PRE o POST).

4. Evaluación y conversión

- **eval(n &Node) i64**: evalúa el árbol de forma recursiva y devuelve el resultado numérico.
- **to_infix(n &Node) string:** convierte el árbol a una cadena en notación infija con paréntesis según la precedencia.

5. Estructura principal (main)

a) Implementa un bucle interactivo:
   - Usa **os.input** para leer comandos del usuario.
   - Comandos:
     - EVAL PRE|POST <expr>: Evalúa la expresión.
     - MOSTRAR PRE|POST <expr>: Muestra la versión infija.
     - SALIR: Termina el programa.

b) Maneja errores con el sistema de opciones de **V (or { ... }).**

6. Rasgos específicos de V usados

- Tipos **opcional (?T)** y el manejo de **none.**
- Conversión directa **str.i64()** para números.
- Uso de **punteros seguros (&T)** con **unsafe { nil }** para inicialización.
- Estructuras y *match* con chequeo exhaustivo del compilador.

Dejo aquí el link al github del proyecto.

PRUEBAS MANUALES

```
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2> v run expr.v
Calculadora PRE/POST (+ - * /)
Comandos: EVAL/MOSTRAR PRE|POST <expr>  |  SALIR
> EVAL PRE + * + 3 4 5 7
42
> MOSTRAR PRE + * + 3 4 5 7
(3 + 4) * 5 + 7
> EVAL POST 8 3 - 8 4 4 + * +
69
> MOSTRAR POST 8 3 - 8 4 4 + * +
8 - 3 + 8 * (4 + 4)
> MOSTRAR PRE - * 10 + 1 1 5
10 * (1 + 1) - 5
> MOSTRAR PRE + 5 * 3 4
5 + 3 * 4
> EVAL PRE - * 10 + 1 1 5
15
> MOSTRAR PRE - * 10 + 1 1 5
10 * (1 + 1) - 5
> MOSTRAR POST 5 1 2 + 4 * + 3 -
5 + (1 + 2) * 4 - 3
> EVAL POST 5 1 2 + 4 * + 3 -
14
> EVAL PRE / * 10 4 2
20
> EVAL PRE + + 3 4
V panic: Expresión prefix incompleta
v hash: ac2e7d7
C:/Users/PC/AppData/Local/Temp/v_0/expr.01K8NN1SQ6YQSG5QHWERVZWT49.tmp.c:4399: at builtin___v_panic: Backtrace
C:/Users/PC/AppData/Local/Temp/v_0/expr.01K8NN1SQ6YQSG5QHWERVZWT49.tmp.c:7539: by main__parse_prefix
C:/Users/PC/AppData/Local/Temp/v_0/expr.01K8NN1SQ6YQSG5QHWERVZWT49.tmp.c:7547: by main__parse_prefix
C:/Users/PC/AppData/Local/Temp/v_0/expr.01K8NN1SQ6YQSG5QHWERVZWT49.tmp.c:7658: by main__parse
C:/Users/PC/AppData/Local/Temp/v_0/expr.01K8NN1SQ6YQSG5QHWERVZWT49.tmp.c:7698: by main__main
C:/Users/PC/AppData/Local/Temp/v_0/expr.01K8NN1SQ6YQSG5QHWERVZWT49.tmp.c:7739: by wmain
004411c0 : by ???
00441323 : by ???
7fffac8ee8d7 : by ???
```

Nota En **V**, la función panic(msg string) se usa para detener la ejecución del programa cuando ocurre un error crítico o inesperado. Al llamarla, el programa muestra el mensaje especificado y termina inmediatamente. En el ejemplo anterior se muestra el *backtrace* que realiza **Panic** desde donde se dio el error hasta la función que la llamó originalmente. Esto fue de mucha ayuda a la hora de depurar el código.

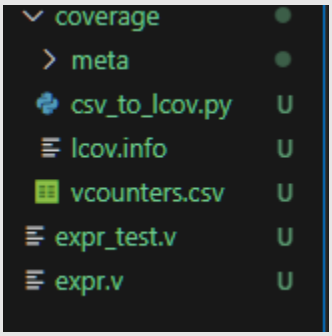
El proyecto además incluye pruebas unitarias, abajo dejo evidencia de la ejecución de las pruebas

```
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta2> v -stats test .
---- Testing... ----------------------------------------------------------------------------------------------------------------------------------
       V  source  code size:      31757 lines,     145715 tokens,     867742 bytes,   363 types,    13 modules,   141 files
generated  target  code size:       9828 lines,     363080 bytes
compilation took: 1590.215 ms, compilation speed: 19970 vlines/s, cgen threads: 11
running tests in: C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta2\expr_test.v
      OK    [ 1/11]     0.029 ms     1 assert  | main.test_eval_enunciado_1()
      OK    [ 2/11]     0.005 ms     1 assert  | main.test_mostrar_enunciado_1()
      OK    [ 3/11]     0.004 ms     1 assert  | main.test_eval_enunciado_2()
      OK    [ 4/11]     0.003 ms     1 assert  | main.test_mostrar_enunciado_2()
      OK    [ 5/11]     0.002 ms     1 assert  | main.test_mostrar_precedencia_sin_parentesis_extras()
      OK    [ 6/11]     0.005 ms     2 asserts | main.test_mostrar_misma_precedencia_no_asociativa_requiere_parentesis()
      OK    [ 7/11]     0.004 ms     2 asserts | main.test_mostrar_division_no_asociativa()
      OK    [ 8/11]     0.003 ms     1 assert  | main.test_mostrar_mixta_mas_menos_y_por_div()
      OK    [ 9/11]     0.001 ms     1 assert  | main.test_eval_anidado()
      OK    [10/11]     0.001 ms     1 assert  | main.test_eval_division_entera()
      OK    [11/11]     0.007 ms     2 asserts | main.test_postfix_largo_eval_y_mostrar()
    Summary for running V tests in "expr_test.v": 14 passed, 14 total. Elapsed time: 0 ms.

 OK    1738.869 ms C:/Users/PC/Desktop/Laboratorios/Lenguajes/Tarea2/Pregunta2/expr_test.v
------------------------------------------------------------------------------------------------------------------------------------------------
Summary for all V _test.v files: 1 passed, 1 total. Elapsed time: 1741 ms, on 1 job. Comptime: 0 ms. Runtime: 1738 ms.
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta2>
```

Este output no incluye todas las pruebas de cobertura, el procedimiento para eso será explicado a continuación.

Para obtener la cobertura por líneas en el lenguaje V, primero se ejecutan las pruebas con el comando



v -cov Coverage test

Esto genera una carpeta llamada **Coverage** que contiene los resultados en formato **CSV** (vcounters.csv) junto con otra carpeta llamada **meta**, que contiene los metadatos. Luego, este archivo se convierte al formato estándar LCOV mediante un script (que se realizó en Python en este caso), el cual procesa el **CSV** y produce el archivo **Coverage/lcov.info**.

Una vez generado, la cobertura puede visualizarse con ***lcov --summary Coverage/lcov.info***, que muestra un resumen porcentual en consola. Este proceso permite obtener y analizar la cobertura por línea.

```
angelryzen@ANGEL-RYZEN:/mnt/c/Users/PC/Desktop/Laboratorios/Lenguajes/Tarea2/Pregunta2$ lcov --summary Coverage/lcov.info
Summary coverage rate:
  lines......: 100.0% (573 of 573 lines)
  functions..: no data found
  branches...: no data found
```

Lamentablemente el ***Coverage*** actual de **V** no incluye cobertura de funciones ni ramificación, así que sólo se muestra la cobertura de líneas.

Pregunta 3

Considere los siguientes iteradores escritos en Python:

```
def suspenso(a, b):
    if b == []:
        yield a
    else:
        yield a + b[0]
        for x in suspenso(b[0], b[1:]):
            yield x
```

```
for x in suspenso(X + Y + Z, [X, Y, Z]):
    print(x)
```

```
def misterio(n):
    if n == 0:
        yield [1]
    else:
        for x in misterio(n-1):
            r = []
            for y in suspenso(0, x):
                r = [*r, y]
            yield r
```

```
for x in misterio(5):
    print(x)
```

a)Tomando como referencia las consantes X, Y y Z planteadas en el examen, considere también el siguiente fragmento de código que hace uso del iterador suspenso, en nuestro caso:

X = 4 Y = 3 Z = 6

Ejecute, paso a paso, el fragmento de código mostrado (por lo menos a nivel de cada nuevo marco de pila creado) y muestre lo que imprime. El iterador Misterio:

Considere también el siguiente fragmento de código que hace uso del iterador misterio:

Ejecute, paso a paso, el fragmento de código mostrado (por lo menos a nivel de cada nuevo marco de pila creado) y muestre lo que imprime.

Nota: Como ya conocemos el comportamiento del iterador suspenso, no es necesario mostrar los pasos internos en el ciclo interno de misterio.

RESPUESTAS

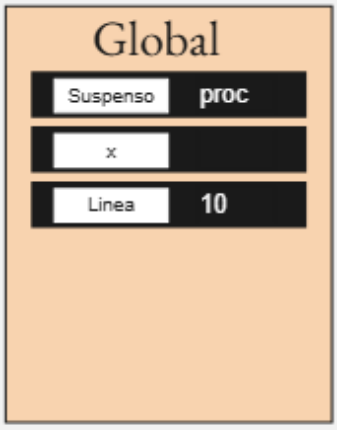
a) Se empieza con la declaración del procedimiento suspenso, para luego interpretar x

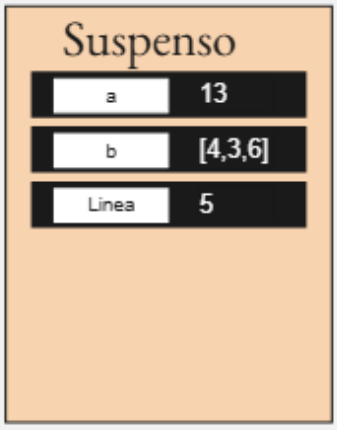
Usaremos X = 4 , Y = 3, Z = 6

```
1  def suspenso(a, b):
2      if b == []:
3          yield a
4      else:
5          yield a + b[0]
6          for x in suspenso(b[0], b[1:]):
7              yield x
8
9
10 for x in suspenso(4 + 3 + 6, [4, 3, 6]):
11     print(x)
```

Primera llamada a Suspenso

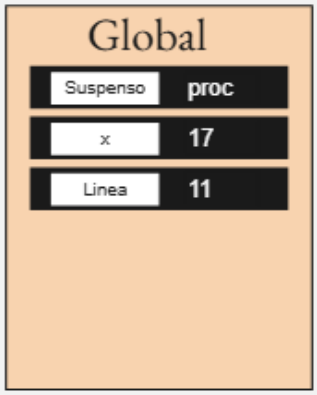
```
1  def suspenso(a, b):
2      if b == []:
3          yield a
4      else:
5          yield a + b[0]
6          for x in suspenso(b[0], b[1:]):
7              yield x
8
9
10 for x in suspenso(13, [4, 3, 6]):
11     print(x)
12
```

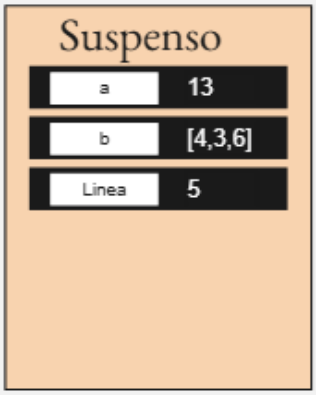




Hace **yield a+b[0] = 17**, se actualiza **x=17** del **for global** y se imprime por primera vez.

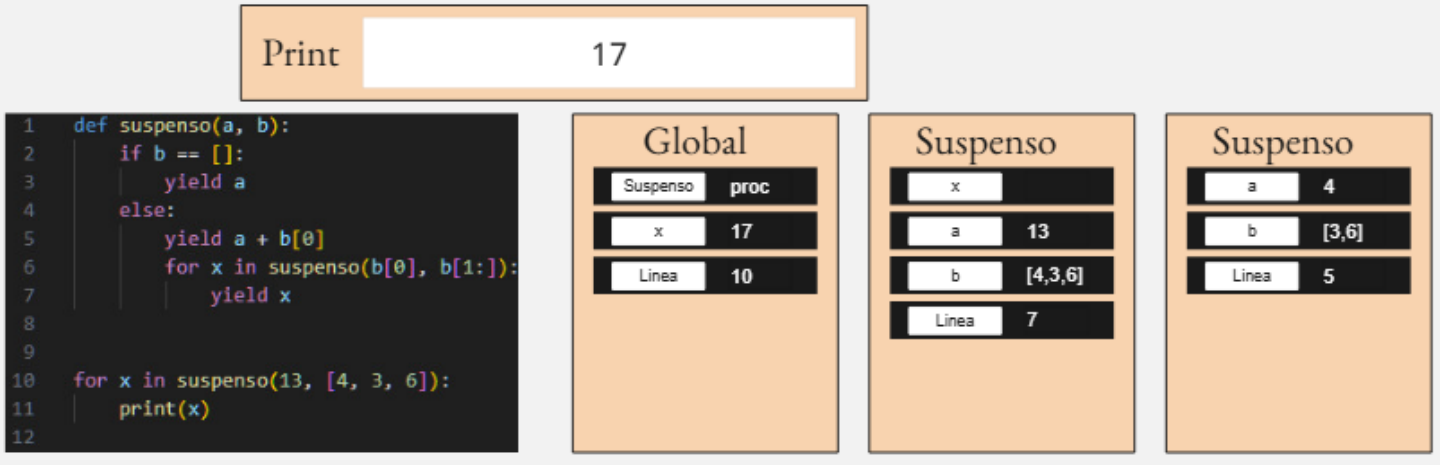

```
1  def suspenso(a, b):
2      if b == []:
3          yield a
4      else:
5          yield a + b[0]
6          for x in suspenso(b[0], b[1:]):
7              yield x
8
9
10 for x in suspenso(13, [4, 3, 6]):
11     print(x)
12
```





Primera llamada recursiva a Suspenso, se pasa **a = 4** y **b = [3,6]**

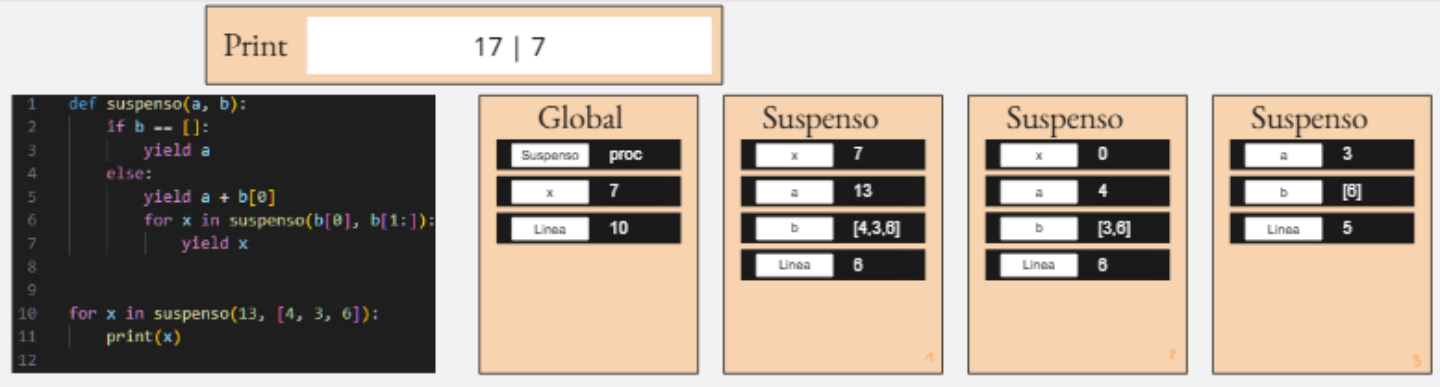
(a partir de la primera impresión mantendré el print hacia la derecha, pero en terminal se imprime con un salto de línea)



Se hace el yield de **x = a + b[0]**, entonces **x = 7**. Junto con la segunda llamada recursiva a Suspenso, se pasa **a = 3 y b = [6]**

(estas llamadas recursivas continuan hasta que **b = [ ]**)

Se actualiza el **for global** con el valor de X resuelto, luego se imprime **x = 7**.



Se hace el yield de **x = a + b[0]**, entonces **x = 9**. Junto con la tercera llamada recursiva a Suspenso, se pasa **a = 6 y b = [ ]**

Se actualiza el **for global** con el valor de X resuelto, luego se imprime **x = 9**.

Ahora el código entra por el **if b == []** y devuelve **a = 6**, esto actualiza al **x** global y se da la ultima impresión.

Finalmente obtenemos esto por terminal:

```
[Running] python -u "c:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta2\varios.py"
17
7
9
6

[Done] exited with code=0 in 0.073 seconds
```

b) Partimos del **for** que llama a **misterio**:

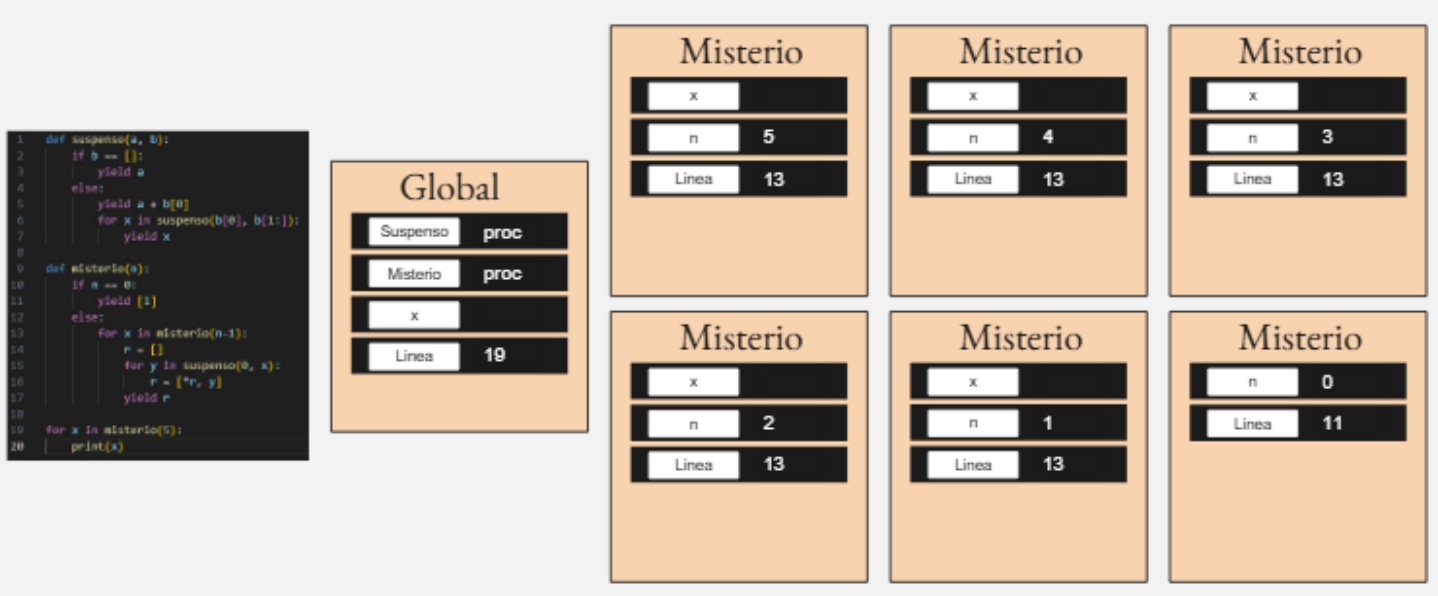
```
1   def suspenso(a, b):
2       if b == []:
3           yield a
4       else:
5           yield a + b[0]
6           for x in suspenso(b[0], b[1:]):
7               yield x
8
9   def misterio(n):
10      if n == 0:
11          yield [1]
12      else:
13          for x in misterio(n-1):
14              r = []
15              for y in suspenso(0, x):
16                  r = [*r, y]
17              yield r
18
19  for x in misterio(5):
20      print(x)
```

Se llama la primera vez a **misterio** con **n=5**

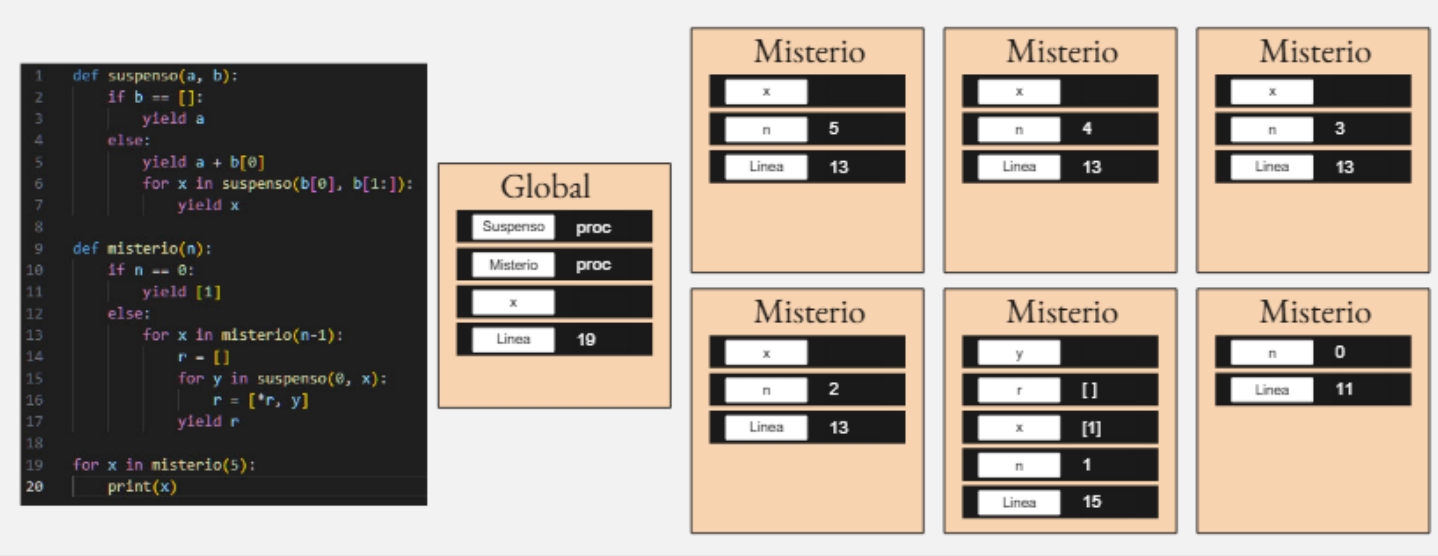
```
1   def suspenso(a, b):
2       if b == []:
3           yield a
4       else:
5           yield a + b[0]
6           for x in suspenso(b[0], b[1:]):
7               yield x
8
9   def misterio(n):
10      if n == 0:
11          yield [1]
12      else:
13          for x in misterio(n-1):
14              r = []
15              for y in suspenso(0, x):
16                  r = [*r, y]
17              yield r
18
19  for x in misterio(5):
20      print(x)
```

Esta llamada a **misterio** ocurre 5 veces más, dado que **n** se va disminuyendo en 1 cada vez que se hace la llamada, finalizando cuando **n == 0**, para administrar mejor el espacio saltaré directamente al momento en el que **n == 0,** justo antes que la función comience a devolverse.

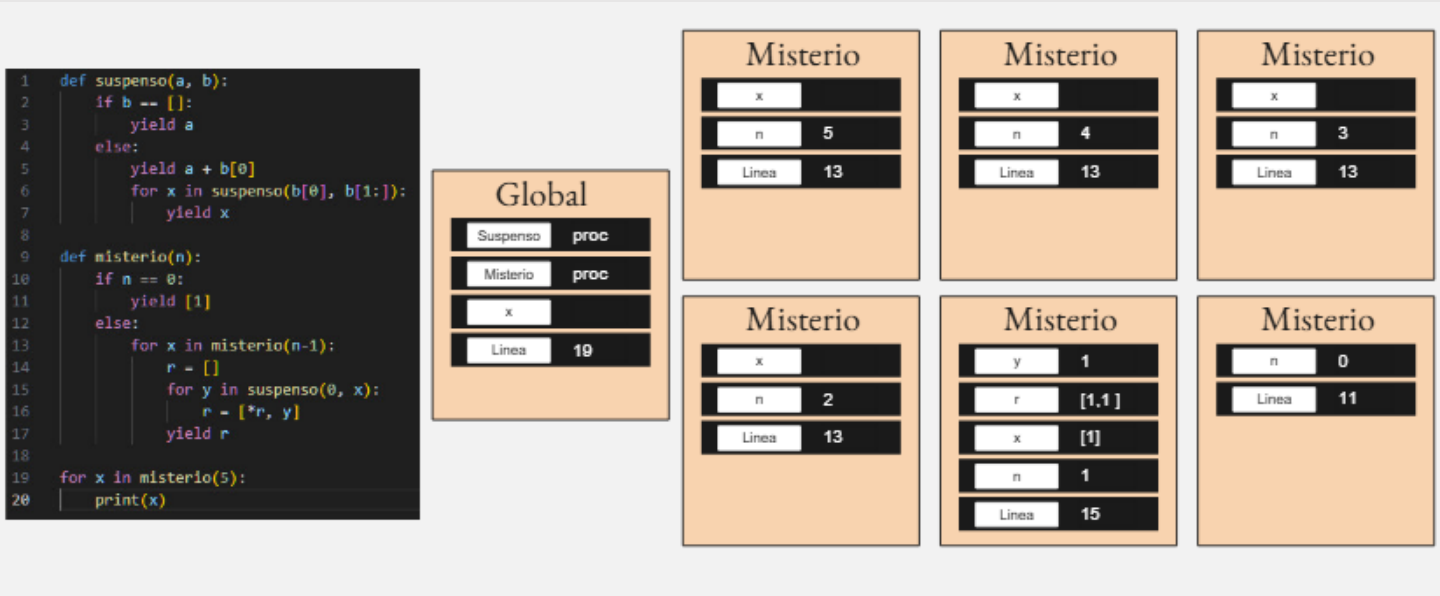
Aquí se pueden ver las 5 llamadas adicionales, de izquierda a derecha y de arriba hacia abajo, disminuyendo en 1 con cada llamada adicional, a partir de este momento la ejecución comienza a regresarse.



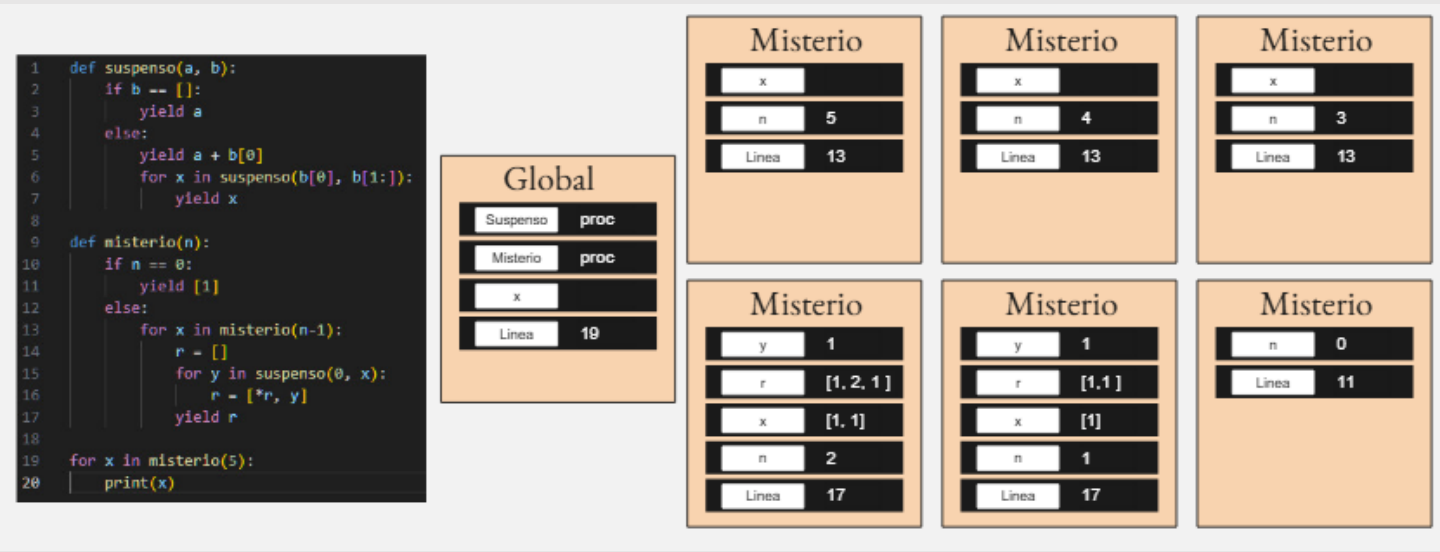
En este momento **n==0** y se pasa por el **if**, resultando en el **yield [1]**, esto actualiza a la penúltima llamada con **x = [1]** y **r =[ ],** además se declara **y**.



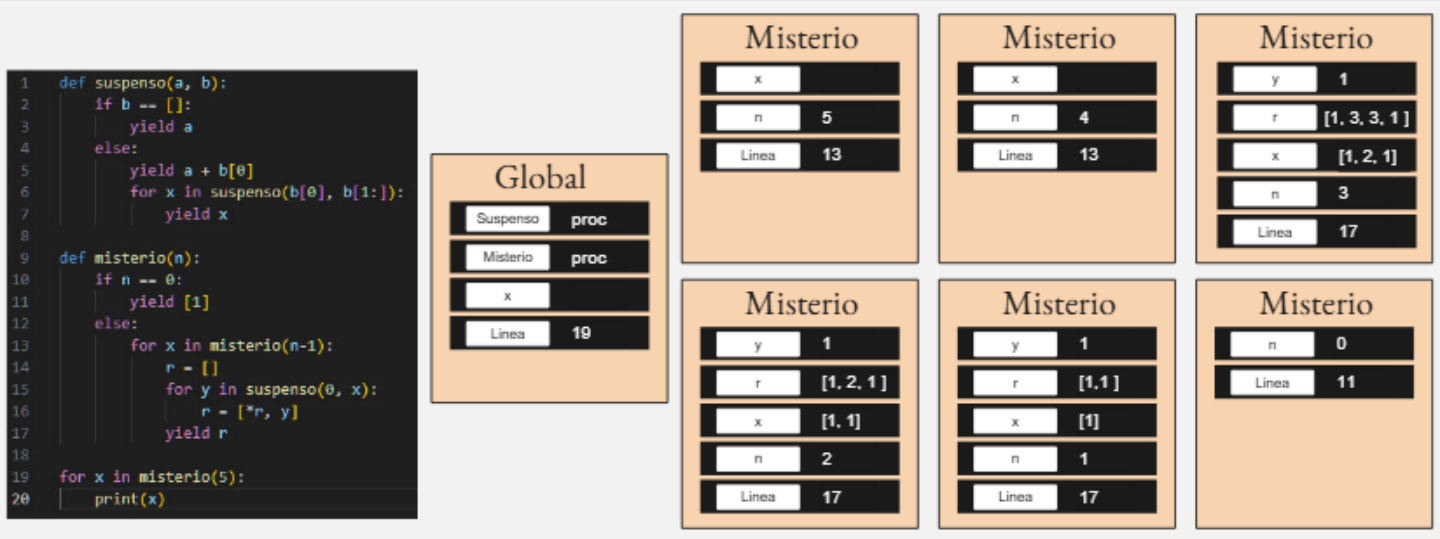
Se hace la llamada implícita a suspenso, y retorna **r = [1,1]** e **y = 1**. La **y** actualiza al arreglo **r**, se hará evidente con los valores de **x** a lo largo del resto de las ejecuciones que los arreglos siempre tendrán la forma **r = [1, … , 1]** esto se debe a que **y** siempre vale 1.



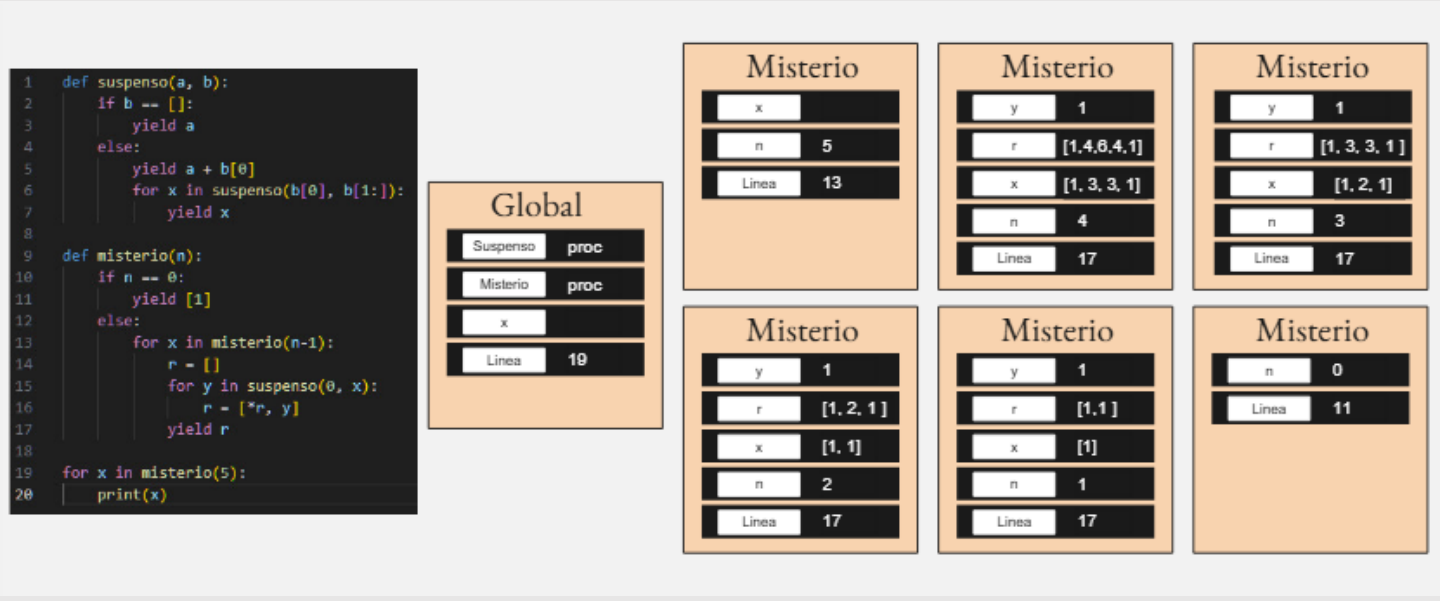
Se actualiza la **X** de la anterior llamada (la 4ta llamada), con el valor de la **r**, esto hace que **x = [1,1]**, con esta **x** se llama a **Suspenso**, retornando **r = [1,2,1]**



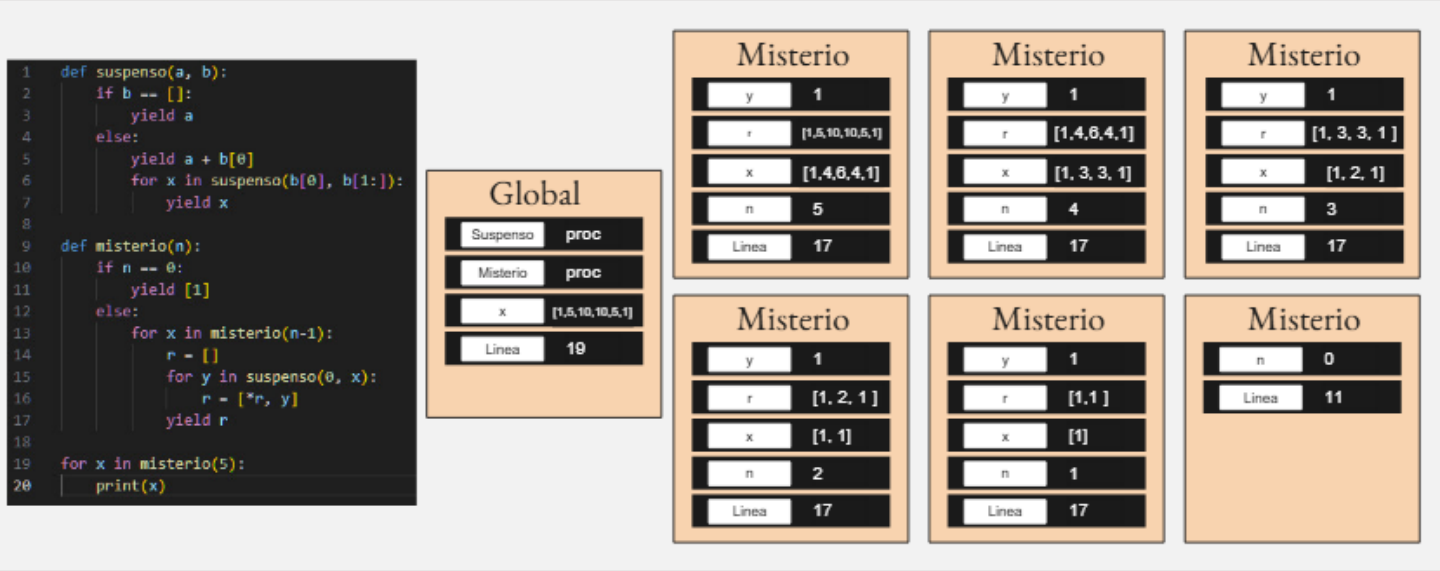
Se actualiza la **X** de la anterior llamada (la 3ra llamada), con el valor de la **r**, esto hace que **x = [1,2,1]**, con esta **x** se llama a **Suspenso**, retornando **r = [1,3,3,1]**



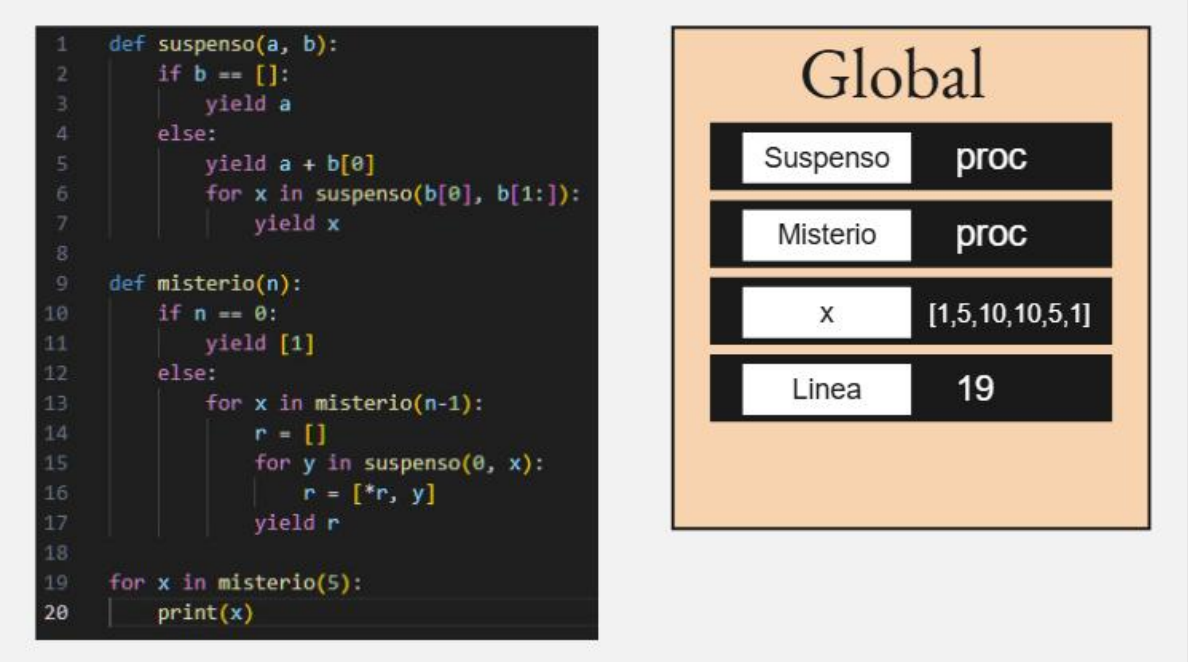
Se actualiza la **X** de la anterior llamada (la 2da llamada), con el valor de la **r**, esto hace que **x = [1,3,3,1]**, con esta **x** se llama a **Suspenso**, retornando **r = [1,4,6,4,1]**



Se actualiza la **X** de la anterior llamada (la 1ra llamada), con el valor de la **r**, esto hace que **x = [1,4,6,4,1]**, con esta **x** se llama a **Suspenso**, retornando **r = [1,5,10,10,5,1]**



Después de esto el programa empieza a eliminar procesos desde la 6ta llamada hacia la primera, uno a uno, finalmente al llegar a la primera llamada, **r = [1,5,10,10,5,1]** actualiza a la **x** global, siendo esto lo que se imprime por consola al final de la ejecución.



Esto se muestra por consola:

```
[Running] python -u "c:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta2\tempCodeRunnerFile.py"
[1, 5, 10, 10, 5, 1]

[Done] exited with code=0 in 0.061 seconds
```

c) El ejercicio que se planteó requiere un iterador que, sin organizar previamente a la impresión.

Para esto una solución sencilla es un algoritmo del estilo de **BubbleSort** pero iterativo, no tiene la mejor complejidad temporal siendo $O(n^2)$ pero resuelve el problema cumpliendo con los requisitos mencionados.

```
def BubbleIterMin(list):
    if not list:
        return
    else:
        while len(list) != 0:
            menor = min(list)
            list.remove(menor)
            yield menor
for x in BubbleIterMin([3, 20, 5, 1]):
    print(x)
```

Verifica si la lista está vacía, si lo está, no hace nada.

Si la lista no está vacía, entra en un bucle que busca el mínimo de la lista actual, almacena el mínimo en “menor”, elimina el mínimo de la lista y luego devuelve “menor” a la variable **x**, la cual lo imprime. Luego sigue dentro del bucle **while**, operando de la misma manera hasta que la lista esté vacía, para ese momento ya habrá impreso la lista ordenada.

Pregunta 4

Para la resolución del problema, se utilizó Python, un lenguaje imperativo que cuenta con todas las estructuras de control de flujo necesarias (condicionales, bucles, funciones, recursión y manejo de memoria).

Aplicando las fórmulas dadas:

$$\alpha = \big((4+3) mod 5\big) + 3 = 5,$$

$$\beta = \big((3+6) mod 5\big) + 3 = 7$$

Por tanto, se evaluó la función $F_{5,7}(n)$

Aquí dejo el link al GitHub donde se encuentra el código.

a) Recursión directa

Esta versión traduce literalmente la definición matemática al código.
Cuando $n < \alpha\beta$ devuelve n; de lo contrario, realiza α llamadas recursivas restando múltiplos de β.

```python
# misma recursión pero con memoización usando lru_cache
# esto convierte la exponencial en algo mucho más manejable
# nota: lru_cache guarda resultados por (n,a,b), si cambio a o b
# conviene limpiar la caché con fab_recursive_memo.cache_clear()
@lru_cache(maxsize=None)
def fab_recursive_memo(n: int, a: int, b: int) -> int:
    if n < a * b:
        return n
    # más compacto
    return sum(fab_recursive_memo(n - b * i, a, b) for i in range(1, a + 1))
```

b) Recursión de cola

En esta versión se mantiene una lista **dp** con los valores ya calculados.
Solo se hace una llamada recursiva por paso, por lo que la función se comporta como una recursión de cola.

```python
# versión recursiva de cola que construye la secuencia con una cola
# en realidad Python no optimiza tail calls, así que lo hago iterativo pero
# con una función helper recursiva que simplemente itera añadiendo a dp.
# dp guarda los valores F(0)..F(k-1) y voy calculando el siguiente.
def fab_tail(n: int, a: int, b: int) -> int:
    base_len = a * b
    # inicializo dp con la base F(0)=0, F(1)=1, ..., F(base_len-1)=base_len-1
    dp = [k for k in range(min(n + 1, base_len))]
    if n < base_len:
        return dp[n]

    def helper(k: int) -> int:
        # k es el índice que vamos a calcular ahora; si ya pasamos n, devolvemos
        if k > n:
            return dp[n]
        # calculo F(k) sumando las a entradas anteriores que corresponden
        # a los saltos de tamaño b: F(k-b), F(k-2b), ..., F(k-ab)
        val = 0
        for i in range(1, a + 1):
            val += dp[k - b * i]
        dp.append(val)
        # llamo de nuevo para el siguiente índice; comoun bucle
        return helper(k + 1)

    return helper(base_len)
```

c) Iterativa

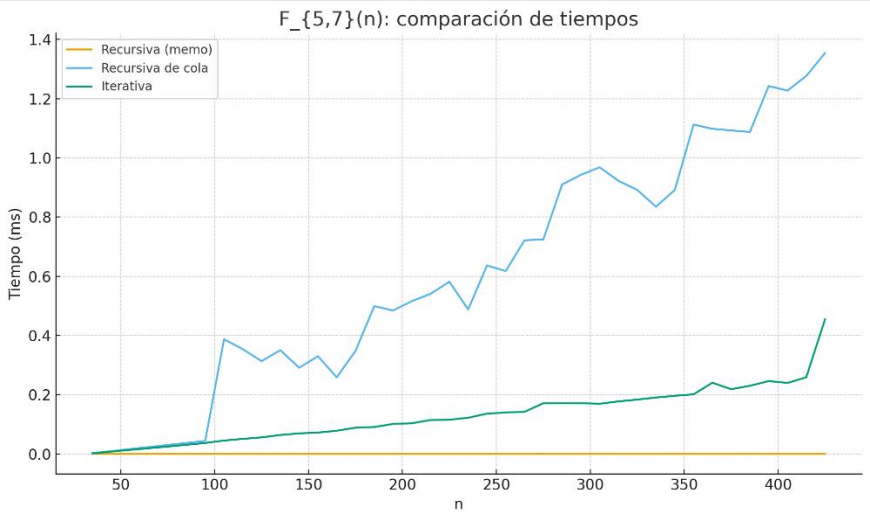
Se construyó un bucle **for** que recorre desde $\alpha\beta$ hasta n, aplicando la misma fórmula de transición que la versión de cola. Esta versión es equivalente en lógica, pero reemplaza la recursión por iteración.

```python
# versión iterativa con programación dinámica (bottom-up)
# esta es la más clara y eficiente en Python: construye dp[0..n]
def fab_iterative(n: int, a: int, b: int) -> int:
    base_len = a * b
    if n < base_len:
        return n
    # dp[k] guardará F(k)
    dp = [0] * (n + 1)
    # valores base
    for k in range(base_len):
        dp[k] = k
    # construyo hacia adelante usando la relación de recurrencia
    for k in range(base_len, n + 1):
        dp[k] = sum(dp[k - b * i] for i in range(1, a + 1))
    return dp[n]
```

Resultados observados:

La recursiva con memoización y la iterativa muestran un comportamiento lineal con n, mientras que la versión de cola presenta un leve aumento adicional por el costo de las llamadas recursivas.

- Las tres versiones devuelven los mismos resultados para todos los valores de entrada probados, confirmando la correcta implementación.
- En Python no existe optimización automática de recursión de cola, por lo que la versión iterativa es más eficiente en la práctica.



Correctitud: Todas las versiones producen los mismos valores de $F_{5,7}(n)$

Esto lo verifico con un pequeño script que compara resultados, estará adjunto en el GitHub también.

```python
from fab import compute_alpha_beta, fab_recursive_memo, fab_tail, fab_iterative

a,b = compute_alpha_beta(4,3,6)
print(f"Probando F_{{{a},{b}}}(n)")

for n in range(a*b, a*b+600, 10):
    fab_recursive_memo.cache_clear()
    v1 = fab_recursive_memo(n,a,b)
    v2 = fab_tail(n,a,b)
    v3 = fab_iterative(n,a,b)
    assert v1==v2==v3, f"Error en n={n}: {v1},{v2},{v3}"

print("Todos los n probados son iguales.")
```

Eficiencia:

- La versión iterativa es la más rápida y estable.
- La recursiva con memoización es clara y también eficiente para n moderados.
- La recursiva de cola es funcionalmente correcta, pero no mejora el rendimiento por las limitaciones de Python.

Complejidad: Todas las implementaciones con memo o DP son $O(\alpha \cdot n)$ en tiempo y $O(n)$ en memoria.

Conclusión general: La versión iterativa es la más recomendable para valores grandes de nnn, mientras que la recursiva con memo es útil para ilustrar el comportamiento teórico del modelo.

Pregunta 5

¡Reto extra POLIGLOTA!

```c
#ifdef __cplusplus ///Si el compilador detecta que el código se está compilando como C++,
  #include <cmath> // entonces incluye <cmath> en lugar de <math.h>.
using std::log;
  using std::floor;
#else
  #include <math.h> // Si no (o sea, si es C u Objective-C), usa <math.h> normalmente.
#endif
```

Aquí le dejo el enlace al GitHub del proyecto.

Este programa fue hecho en C, pero se escribió de una forma que lo hace compatible con C++ y también con Objective-C.
La idea fue no usar nada que fuera exclusivo de alguno de esos lenguajes, sino mantener solo funciones y estructuras básicas de C (como printf, scanf o math.h), que los otros dos lenguajes también entienden.

Para probarlo, se usaron tres compiladores diferentes:

- gcc para C,
- g++ para C++,
- y clang -ObjC en WSL para Objective-C.

En todos los casos el programa compiló sin errores, se ejecutó correctamente y dio el mismo resultado. Por ejemplo, al ingresar el número **8**, en los tres lenguajes el resultado fue **68.000000**.

Esto demuestra que el código puede funcionar igual en los tres lenguajes si se escribe de manera limpia y sin usar funciones propias de cada uno.
En resumen, al usar únicamente lo que comparten C, C++ y Objective-C, se logró que un mismo archivo (maldad.c) sea totalmente portable entre los tres.

```
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta5> wsl clang -ObjC -O2 maldad.c -lm -o maldad_objc
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta5> .\maldad_objc
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta5> wsl "/mnt/c/Users/PC/Desktop/Laboratorios/Lenguajes/Tarea2/Pregunta5/maldad_objc"
Ingresa un numero: 8
68.000000
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta5> gcc -O2 maldad.c -lm -o maldad_c
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta5> .\maldad_c.exe
Ingresa un numero: 8
68.000000
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta5> g++ -O2 maldad.c -lm -o maldad_cpp
PS C:\Users\PC\Desktop\Laboratorios\Lenguajes\Tarea2\Pregunta5> .\maldad_cpp.exe
Ingresa un numero: 8
68.000000
```