Jose Revete
19-10040

Examen 2

X=0, Y=4, Z=0

1. Lenguaje escogido: **RUBY**
    a. De una breve descripción del lenguaje escogido.
        i. Enumere y explique las estructuras de control de flujo que ofrece.

R: En Ruby encontramos la siguientes estructuras de de control de flujo:

- Condicionales:

  if, elsif, else: Permiten ejecutar bloques de código basándose en condiciones verdaderas o falsas.

  unless: Ejecuta un bloque de código sólo si la condición es falsa.

  Operador ternario (condicional ? true : false): Operador ternario para expresiones condicionales en una sola línea.
- Bucles:

  while: Ejecuta el bloque de código mientras la condición sea verdadera.

  until: Ejecuta el bloque de código hasta que la condición se vuelva verdadera.

  for: Itera sobre un rango o una colección.

  each: Itera sobre elementos de una colección (arreglos, rangos, hashes).

  loop do ... end: Ejecuta un bloque de código de forma indefinida hasta que se interrumpa con **break**.
- Control en Bucles

  break: Finaliza el bucle de inmediato.

  next: Salta la iteración actual y pasa a la siguiente.

  redo: Reinicia la iteración actual sin evaluar la condición.

  retry: Reinicia el bucle completo desde el principio.

  return: Salida de un método (puede retornar un valor).

  throw y catch: Similar a break, permite una salida del bucle (o bloques anidados) con una etiqueta específica.
- Expresiones Condicionales en Línea

  if en línea: Ejecuta una expresión si la condición es verdadera, en una sola línea.

  unless en línea: Ejecuta una expresión si la condición es falsa, en una sola línea.
- Case / When

  case / when / else: Evalúa una expresión y ejecuta un bloque de código basado en el valor de esa expresión.
- Estructuras de Excepciones

  begin / rescue / else / ensure: Estructura para manejo de excepciones.

  raise: Lanza una excepción que puede ser capturada con rescue.

  retry (en excepciones): Reintenta el bloque begin desde el inicio después de un rescue.
- Expresiones defined?

defined?: Evalúa si una variable, método o constante está definida, devolviendo un valor nil si no lo está.

- Bloques y Procs

Bloques: Son estructuras de código que pueden ser pasadas como argumento a métodos y se pueden ejecutar usando yield.

yield: Ejecuta el bloque pasado a un método.

Proc y lambda: Objetos que encapsulan bloques de código, permitiendo ejecutar lógica específica en cualquier lugar del programa.

return en lambda y Proc: En lambda retorna del lambda mientras que en Proc retorna del método completo donde se llama.

ii. Diga en qué orden evalúan expresiones y funciones.

R: En cuanto a las expresiones, Ruby utiliza la precedencia y asociatividad, basándose en el orden de operadores y, a su vez, de izquierda a derecha para operadores de igual precedencia. Por otro lado, para las funciones evalúa los argumentos antes de invocar esta. Lo que significa que se calculan todos los valores necesarios para ejecutar la función antes de que la función se ejecute.

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

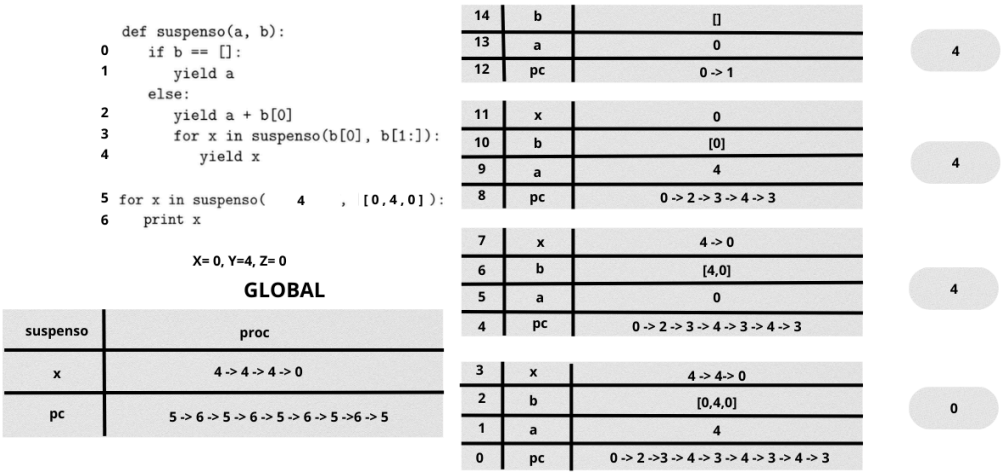
R: Ruby utiliza evaluación aplicativa, lo que significa que evalúa los argumentos de una función antes de llamar a la función en sí. Ruby no tiene evaluación perezosa de manera nativa. Todos los argumentos se evalúan antes de invocar la función; pero es posible imitarla usando **proc** y **lamba**

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

R: La evaluación en Ruby se realiza de izquierda a derecha, por lo que en una expresión con múltiples argumentos o en una función con varios parámetros, se evalúa primero el argumento a la izquierda y luego continua hacia la derecha.

3. Considere los siguientes iteradores, escritos en Python:

a.



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

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

X= 0, Y=4, Z= 0

GLOBAL

misterio	proc
x	-- -> [1, 5, 10, 10 , 5, 1]
pc	7 --> 8 --> 7

[1, 5, 10, 10 , 5, 1]

26	n	0
25	pc	0 --> 1

24	y	1 -> 1
23	r	[ ] --> [1] -> [1,1]
22	x	-- -> [1]
21	n	1
20	pc	0 -> 2 -> 3 -> 4 ->5 -> 4 ->5 -> 4 -> 6

19	y	1 -> 2 -> 1
18	r	[] --> [1] --> [1, 2] --> [1,2,1]
17	x	-- -> [1,1]
16	n	2
15	pc	0 -> 2 -> 3 -> 4 -> 5 -> 4 -> 5 -> 4 -> 5 -> 4-> 6

14	y	1 --> 3 --> 3 --> 1
13	r	[] -> [1] --> [1,3] --> [1,3,3] --> [1,3,3,1]
12	x	-- -> [1,2,1]
11	n	3
10	pc	0 -> 2 -> 3 -> 4 -> 5 ->4 -> 5 ->4 -> 5 ->4 -> 5 ->4 -> 6

9	y	1-> 4 -> 6 -> 4 -> 1
8	r	[] -> [1] -> [1,4] -> [1,4,6] -> [1,4,6,4] -> [1,4,6,4,1]
7	x	-- -> [1,3,3,1]
6	n	4
5	pc	0 -> 2 -> 3 -> 4 -> 5 ->4 -> 5 ->4 -> 5 ->4 -> 5 ->4 -> 5 ->4->6

4	y	1 -> 5 -> 10 -> 10 -> 5 -> 1
3	r	[] -> [1] -> [1,5] -> [1,5 10] -> [1,5,10,10] -> [1,5,10,10,5] -> [1, 5,10, 10, 5, 1]
2	x	-- -> [1,4,6,4,1]
1	n	5
0	pc	0 -> 2 -> 3 -> 4 -> 5 ->4 -> 5 ->4 -> 5 ->4 -> 5 ->4 -> 5 ->4-> 5 -> 4 -> 6

b.

4. Realice también un análisis comparativo entre las tres implementaciones realizadas, mostrando tiempos de ejecución para diversos valores de entrada y ofreciendo conclusiones sobre la eficiencia.

R: Es notable que dependiendo del valor de “n”, las funciones pueden ser más o menos eficientes, por ejemplo, con n <= 75, la función “recursion” es la más eficiente. mientras que “recursion_cola” e “iterativo” son menos eficientes. Pero con valores de n > 75, notamos que “recursion_cola” es más eficiente que “iterativo” y aún más que “recursion”.