**Universidad San Francisco de Quito**
**Inteligencia Artificial**

**Nombre:** José Gabriel García
**Código:** 00211322
**NRC:** 3322

**Tarea Adicional**

1. **Explique en qué consiste el cálculo lambda. ¿Cómo funciona la notación de tipos de funciones?**

   En el área de la matemática, el cálculo lambda es definido como un modelo formal, lo que quiere decir que es un modelo matemático abstracto y se usa para investigar la definición de función, la aplicación de funciones y la recursión. Fue introducido por Alonzo Church y Stephen Kleene en 1930, y este tipo de cálculo posee únicamente 3 elementos: variables, funciones y la aplicación de funciones. De esta manera, en el área de la computación surgen los lenguajes de programación funcional teniendo como base al cálculo lambda.

   El cálculo lambda no maneja tipos de variables como int, bool, float, etc., y trata a todos los valores de manera uniforme creando, así, el conocido Untyped Lambda Calculus. Sin embargo, con el fin de producir programas más seguros, se creó el Typed Lambda Calculus el cual si posee tipos de variables.

   La notación del Untyped Lambda Calculus es la siguiente:

   $$(\lambda x.\, x)\ 3$$

   La función anterior es la más simple que existe en el cálculo lambda, y es conocida como la identidad. En esta función, $\lambda x.\, x$ es equivalente a decir $f(x) = x$, lo que en palabras se traduce a que el input de la función es igual al output, y es por eso es que se la conoce como identidad. Finalmente, el numero $3$ es el valor que reemplazará a $x$ que se encuentra sola, es decir, es el valor en el que se evalua la función. Así, la solución de la expresión anterior es la siguiente:

   $$(\lambda x.\, x)\ 3$$
   $$= 3$$

   Lo mismo que se obtendría al evaluar la función $f(x) = x$ en $x = 3$, de manera que se tendría $f(3) = 3$, llegando al mismo resultado.

   Otro ejemplo sencillo de una expresión lambda es el siguiente:

   $$(\lambda x.\, x + 1)$$

   Si en esta expresión se evalúa el 4 por ejemplo, entonces se tendría lo siguiente:

   $$(\lambda x.\, x + 1)\ 4$$
   $$= 4 + 1$$
   $$= 5$$

   Finalmente, para que este cálculo puedo ser aplicado a la computación, se debe buscar la manera de realizar bucles, y la solución a este problema es la siguiente:

$$\lambda x.\, x\ x$$

Donde se tiene una función que tiene un output igual a dos veces su input, de modo que si se evalúa esta función en si misma, se tiene un loop:

$$(\lambda x.\, x\ x)(\lambda x.\, x\ x)$$

$$= (\lambda x.\, x\ x)(\lambda x.\, x\ x)$$

Y, como se observa, la salida de esa expresión es si misma, pero para poder utilizar este principio para hacer tareas repetitivas se necesita una forma en la que se pueda realizar un proceso intermedio antes de que la función se llame así misma, y entonces aparece el Y-combinator que es la forma en la que se hacen funciones recursivas como en un lenguaje de programación imperativo:

$$\lambda f.\, ((\lambda x.\, f\,(x\ x))(\lambda x.\, f\,(x\ x)))$$

Así, al darle como input una función que representa la tarea que se quiere realizar, esta función ejecutará la función enviada, y luego se llamará así misma con la misma función de input permitiendo la recursividad.

Este mismo calculo lambda, como se mencionó anteriormente, también tiene una versión en la que se implementan el uso de tipo de variables, y funciona de la misma manera que el Untyped Lambda Calculus, pero con la definición de tipo de variables. Así, la notación del Typed Lambda Calculus es la siguiente:

$$\lambda x\colon Type.\, x$$

Como se puede observar, es básicamente la misma estructura, pero aparece el tipo de variable, y este corresponde al input de la función. Así, si se usa la siguiente función:

$$\lambda x\colon Int.\, 2 * x$$

Entonces quiere decir que el tipo de dato del input, y por consiguiente, del output de esta función debe ser Int.

**2. ¿Qué es una expresión lambda? (Incluya ejemplos en Haskell, Python y C++).**

Una expresión lambda es una forma de definir funciones, las cuales tienen un patrón que se debe aplicar a todos los argumentos para conseguir un output deseado, es decir, posee un bloque de instrucciones a seguir que permite alcanzar los resultados utilizando los argumentos enviados; sin embargo, esta expresión, en comparación con las funciones convencionales, no necesita ser definida con un nombre específico, es decir, es una función anónima. Es muy similar a una función de un lenguaje imperativo, pero no es definida antes o después del código principal, sino dentro del mismo. El uso de este tipo de expresiones se puede ver en el manejo de eventos, por ejemplo.

**Ejemplo de expresión lambda en Haskell**

```haskell
-- Sintaxis general: \argumento -> expresion
-- Ejemplo de una función que suma dos números
add :: Int -> Int
add = \x -> x + x

-- Uso de la función
resultado :: Int
resultado = add 3

-- Imprimir el resultado en la consola
main :: IO()
main = putStrLn $ "Resultado: " ++ show resultado
```

En este caso, VSCode nos indica que la expresión lambda puede ser reemplazada por una que no lo sea la cual se escribe como:

```haskell
add x = x + x
```

Pero ambas funcionan correctamente. Sin embargo, si se quiere sumar dos números, la sintaxis es la siguiente:

```haskell
-- Sintaxis general: \argumento -> expresion
-- Ejemplo de una función que suma dos números
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)

-- Uso de la función
resultado :: Int
resultado = add 3 4

-- Imprimir el resultado en la consola
main :: IO()
main = putStrLn $ "Resultado: " ++ show resultado
```

Así, se tiene una función que devuelve otra, y el resultado de esta última es la suma de los dos números. Esta expresión puede ser cambiada por lo siguiente:

```haskell
add x y = x + y
```

**Ejemplo de expresión lambda en Python**

```python
# Sintaxis general: lambda argumentos: expresion
# Ejemplo de una función que suma dos números
sumar = lambda x, y: x + y

# Uso de la función
resultado = sumar(3, 4) # Resultado es igual a 7.
```

Las expresiones lambda se pueden usar también en funciones que evalúan elementos como min(), max(), sorted(), y se las usa para personalizar el funcionamiento de estas funciones. Un ejemplo de esto es la siguiente línea de código.

```
best_solution = min(solutions, key=lambda x: calculate_total_distance(x, distances))
```

Esta línea es parte del algoritmo ABC usado para resolver el problema del TSP. Como se observa, se busca el valor en el arreglo “solutions” con la distancia más pequeña, y para evaluar la distancia de cada elemento de “solutions” se usa una expresión lambda, y se toma el elemento del arreglo “solutions” con la distancia mínima.

**Ejemplo de expresión lambda en C++**

```cpp
#include <iostream>
using namespace std;

int main() {
    // Sintaxis general: [](argumentos) -> tipo { expresion; }
    // Ejemplo de una función que suma dos números
    auto sumar = [](int x, int y) -> int { return x + y; };

    // Uso de la función
    int resultado = sumar(3, 4);
    cout << "Resultado: " << resultado << endl;

    return 0;
}
```

3. **Explique en qué consiste un list comprehension en Haskell. (Incluya ejemplos en Haskell y Python).**

El concepto de list comprehension viene del área de matemáticas donde se usa comprehension notation y sirve para crear conjuntos basados en conjuntos preexistentes. Un ejemplo de esto lo siguiente:

$$\{x^2 | x \in \{1...5\}\}$$

Lo que produce un conjunto con el cuadrado de los números del 1 al 5. Así, en Haskell se usa una notación similar, pero esta es llamada list comprehension.

$$[x\text{^}2 | x <- [1..5]]$$

El símbolo | se lee: tal que, y el símbolo <- indica de donde se van a extraer los datos, y el resultado de la línea de código anterior también produce una lista del cuadrado de los números del 1 al 5. En este ejemplo, se agregan todos los valores del conjunto inicial elevados al cuadrado, sin realizar ningún tipo de selección, pero también puede añadirse algún criterio para limitar los valores que se añaden a la lista resultante, un ejemplo de esto es:

```
primes :: Int -> [Int]

primes n = [x | x <- [2..n], prime x]
```

Donde se usa una función previa llamada prime la cual indica si un número $x$ es primo, y con esta función y un conjunto de valores desde 2 hasta n, se agregan solo los valores primos que se encuentran en el conjunto $[2, n]$. Así, se destacan 2 casos: uno en el que se añaden todos los números de un conjunto a otro, después de aplicarles alguna

modificación, y otro caso en el que solo se añadirán los valores de un conjunto a otro, si estos cumplen con alguna condición, pero también podrían aplicarse la dos condiciones, por ejemplo, añadir el cuadrado de los números primos dentro un conjunto. En conclusión, un list comprehension (en haskell y cualquier otro lenguaje con esta carácterística) es una forma de crear una lista que resulta de evaluar varios elementos con una o varias condiciones.

**Ejemplo de list comprehension en Haskell**

```haskell
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]

resultado :: [Int]
resultado = factors 15 -- El resultado es [1,3,5,15]

prime :: Int -> Bool
prime n = factors n == [1,n]

answer :: Bool
answer = prime 15 -- El resultado es false

primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x] -- El resultado es [2,3,5,7]

result :: [Int]
result = primes 9
```

En este ejemplo se usa varias veces el concepto de list comprehension para encontrar los números primos en el conjunto $[2, n]$.

**Ejemplo de list comprehension en Python**

```python
archivos_csv = [archivo for archivo in archivos if archivo.endswith('.csv')]
```

En este ejemplo se añade todos los string que terminan con ".csv" dentro del arreglo archivos a un nuevo arreglo llamado archivos_csv. La escritura de esto en Haskell sería de esta manera:

$$archivos\_csv :: [String] -> [String]$$

$$archivos\_csv\ archivos = [archivo \mid archivo <- archivos\_csv, archivo\ ends\ with\ ".csv"]$$

4. **Explique los diferentes tipos de recursión que existen. (Incluya ejemplos distintos a los del libro).**

En lenguajes de programación imperativa existe la recursión directa e indirecta. La recursión directa comprende tres tipos: lineal, binaria y múltiple, las cuales consisten en 1 (lineal), 2 (binaria) o 3 en adelante (múltiple) llamadas recursivas dentro de la función que implementa la recursividad. Y, por otro lado, la recursión indirecta se da cuando una función 1 realiza una llamada a la función 2, y, a su vez, la función 2 realiza una llamada a la función 1, de modo que se tiene un ciclo interminable.

De la misma manera, en los lenguajes de programación funcional se tienen tipos de recursividad similares, pero la manera en las que se las debe declarar es diferente. En este

tipo de lenguajes la recursividad es la forma en la que se pueden realizar tareas repetitivas y complejas, y simular la existencia de bucles, los cuales no existen explícitamente en la programación funcional. Así, a continuación, se muestra el tipo de recursividad y un ejemplo de la misma.

**Recursividad con 1 argumento:** La función recursiva se aplica sobre 1 parámetro en cada llamada de si misma, y el caso base debe estar relacionado a este parámetro ya que es la única información que se tiene para detener las llamadas recursivas consecutivas.

```haskell
-- Función para sumar los digitos de un numero
sumaDigitos :: Int -> Int
sumaDigitos 0 = 0  -- El caso base es el numero 0 y la suma de digitos de 0 es 0
sumaDigitos n = n `mod` 10 + sumaDigitos (n `div` 10) -- Recursividad: suma del último dígito
                                                      -- más la suma de los dígitos restantes

prueba :: Int
prueba  = sumaDigitos 23 -- Devuelve 5

main :: IO()
main = print prueba
```

**Recursividad con múltiples argumentos:** Estas funciones recursivas se aplican sobre 2 o más argumentos en cada llamada, y pueden o no tener un caso base por argumento, pero si se debe asegurar que el caso o casos base que se definan deben evitar una recursión infinita.

Ejemplo 1

```haskell
-- Funcion para encontrar la potencia de un numero b elevado a e
potencia :: Int -> Int -> Int
potencia _ 0 = 1 -- Caso base: Cualquier numero elevado a cero es 1
potencia b e = b * potencia b (e-1)  -- Llamada Recursiva: la potencia es
                                     -- la multiplicacion sucesiva de un
                                     -- numero b, e veces

prueba :: Int
prueba = potencia 2 4 -- Devuelve 2^4 = 16

main :: IO()
main = print prueba
```

Ejemplo 2

```haskell
-- Funcion para verificar si en una lista se encuentra un valor
isThere :: Eq a => a -> [a] -> Bool
isThere x [] = False -- Caso base: se tiene una lista vacia, por lo que el
                     -- elemento x no esta en la lista
isThere x (y:ys) | x == y = True
                 | otherwise = isThere x ys -- Llamada recursiva: se extrae el
                                            -- primer valor de la lista y se
                                            -- compara con el valor buscado. El
                                            -- valor de extraido se elimina.

prueba :: Bool
prueba = isThere 3 [1,2,3,4,5] -- Devuelve True

main :: IO()
main = print prueba
```

**Recursividad múltiple:** Se da cuando la llamada de la función recursiva así misma se da más de una vez en una misma instrucción.

```haskell
-- Funcion para encontrar la cantidad de formas de tomar k elementos de un total de n elementos
comb :: Int -> Int -> Int
comb _ 0 = 1  -- Caso base: La cantidad de formas de tomar 0 elementos de un total de n elementos
              -- es 1
comb n k | n == k = 1  -- Caso base: La cantidad de formas de tomar k elementos de un total de
                       -- de k = n elementos es 1
         | otherwise = comb (n-1) (k-1) + comb (n-1) k -- Llamada recursiva: propiedad de las
                                                       -- combinaciones

prueba :: Int
prueba = comb 15 5  -- Devuelve 6

main :: IO()
main = print prueba
```

**Recursividad mutua:** Se da cuando 2 o más funciones recursivas están definidas por la llamada recursiva entre sí; es decir, la función 1 llama a la función 2, y la función 2 llama a la función 1 hasta que alguna de las dos alcanza el caso base y entonces esta llamada consecutiva se detiene.

```haskell
-- Funcion para determinar si una vocal es mayuscula o minuscula
-- Arreglos de las vocales en mayusculas y minusculas para iterar sobre ellas
uppercase :: [Char]
uppercase = ['A', 'E', 'I', 'O', 'U']

lowercase :: [Char]
lowercase = ['a', 'e', 'i', 'o', 'u']

-- Se determina si la vocal es mayuscula
isUppercase :: Char -> [Char] -> [Char] -> Bool
-- Si es mayuscula se imprime True, si no se envia el arreglo de minusculas y se pregunta
-- si es minuscula
isUppercase c (x:xs) (y:ys) | c == x = True
                            | otherwise = isLowercase c xs (y:ys)
isUppercase _ _ _ = False -- Correccion de chatGPT: Caso que no coincide con nada

-- Se determina si la vocal es minuscula
isLowercase :: Char -> [Char] -> [Char] -> Bool
-- Si es minuscula se imprime True, si no se envia el arreglo de mayusculas y se pregunta
-- si es mayuscula
isLowercase c (x:xs) (y:ys) | c == y = True
                            | otherwise = isUppercase c (x:xs) ys
isLowercase _ _ _ = False -- Correccion de chatGPT: caso que no coincide con nada

prueba :: Bool
prueba = isUppercase 'E' uppercase lowercase

main :: IO()
main = print prueba
```

## Referencias


Computerphile (Dirección). (2017). *Essentials: Functional Programming's Y Combinator* [Película].

Computerphile (Dirección). (2017). *Lambda Calculus* [Película].

Hutton, G. (2007). *Programming in Haskell.* Cambridge University Pres: Cambridge University Press.

OpenAI. "ChatGPT." Modelo de lenguaje, OpenAI, 2023, https://openai.com/research/chatgpt.

Pierce, B. (2002). *Types and Programming Languages.* Cambridge, Massachusetts: The MIT Press.

S. Marlow, Ed., Haskell Language Report, 2010, available on the web from: https://www.haskell.org/definition/haskell2010.pdf.