Facultad de Ciencias, UNAM

Práctica 2

Alejandro Hernández Mora Pablo Camacho González Luis Manuel Martínez Dámaso
Edith Araceli Reyes López Adrián Felipe Vélez Rivera

Fecha de entrega: 27 de Noviembre

Introducción.

En esta práctica aprenderemos a hacer definiciones recursivas y a utilizar la estructura de datos lista, en *Haskell*.

1.1. Recursión.

La **Recursión** es uno de los conceptos más importantes que veremos a lo largo del curso.

Decimos que una defición es recursiva si en dicha definición utilizamos la misma definición. Por ejemplo, la función **Factorial** esta definida de la siguiente forma en haskell:

```
factorial :: Int -> Int
factorial 0 = 1 -- Caso base o cláusula de escape
factorial n = n * factorial (n-1) -- Caso recursivo
```

Podemos ver que llamamos a la función factorial en la definición de la misma función. En general, todas las funciones recursivas que implementemos tendrán al menos dos casos:

El caso base, que es en el que la definición no se llama a si misma, se cumple una condición específica[*] y la función es evaluada a un resultado final. Y el caso recursivo, que es en el que se llama de nuevo a la función que se está definiendo. Por lo general en el caso recursivo, se contribuye a un cálculo parcial del resultado que se va a obtener.

1.2. Listas.

El siguiente tipo que usaremos en Haskell son las **Listas**. Una lista es una colección de elementos del mismo tipo. Por ejemplo:

```
[1,2,3,4] -- Lista de tipo Int.
[1.0,2.0,3.0] -- Lista de tipo Float.
['a','b','c'] -- Lista de tipo Char
```

La representación de una lista de tipo **a** en haskell es la siguiente:

```
[a] -- Lista de elementos de tipo a.
[Int] --Lista de tipo Int.
[Char] -- Lista de tipo Char.
```

Por ejemplo, si queremos crear una función que nos devuelva el primer elemento de una lista de tipo **a**, la firma de la función nos quedaría así:

```
primerElemento :: [a] -> a
```

[*] a la que llamamos cláusula de escape

1.2.1. Construcción de una lista en haskell.

Las listas son elementos que están construidos recursivamente. Considera la siguiente definición de listas:

```
[] es una lista.
Si x es un elemento y xs es una lista, entonces (x:xs) es una lista.
Son todas
```

Decimos que [] es la lista vacía. En el caso recursivo, decimos que x es la cabeza de la lista y que xs es la cola de la lista. Observemos que xs también es una lista.

En Haskell se abrevia la construcción de listas de manera que podemos representarla de manera más amigable como un conjunto de elementos entre corchetes, sin embargo lo que esto representa va de acuerdo con la definición anterior. Esto quiere decir que indistintamente podemos utilizar cualquiera de las siguientes representaciones.

```
[1,2,3] = (1:(2:(3:[])))
```

La representación de lista que usaremos dependerá de lo que más nos convenga a la hora de programar y ejecutar funciones.

La primer representación nos es muy útil cuando ya estamos usando la función que requiere una lista como parámetro, o sea, a nivel ejecución. La segunda representación nos servirá cuando tengamos que trabajar con listas en las funciones.

1.3. Listas por comprensión

Otra forma de construir una lista es utilizando la notación de conjuntos matemáticos. Por ejemplo, si queremos una lista que tenga por elementos las parejas (x, z), con x en la lista xs y z en la lista zs, definimos la función de la siguiente forma:

```
-- La funcion productoCruz calcula el producto entre las listas xs y zs.
-- [(x,z) | x <- xs, z <- zs] es una lista con elementos (x,z)
-- tal que x pertenece a xs y z pertenece a zs.
productoCruz :: [a] -> [b] ->[(a,b)]
productoCruz xs zs = [(x,z) | x <- xs, y <- ys]
```

Ajuste de Patrones.

Dada la definición de lista, podemos observar que cualquier lista cumple con uno de los dos posibles patrones de construción. Es decir que cualquier lista, es una lista vacía o es una lista que tiene cabeza y cola.

Veamos un ejemplo de una función que obtiene el primer elemento de una lista:

```
primerElemento :: [a] -> a
primerElemento (x:xs) = x
```

Esto quiere decir que si nuestra función recibe una lista que tenga cabeza y cola, devolvemos la cabeza. Por otro lado, también estamos limitando a que la función sólo acepte listas que tengan cabeza y cola. Si ejecutamos la función con la lista vacía, el intérprete nos dirá que la lista vacía no se puede ajustar al patron de la función.

Veamos otro ejemplo. Queremos una función recursiva, que devuelva el último elemento de una lista. Vamos a definir la función así:

```
ultimoElemento :: [a] -> a
ultimoElemento [x] = x
ultimoElemento (x:(z:zs)) = ultimoElemento (z:zs)
```

Podemos observar que si nuestra función recibe una lista con un sólo elemento, podemos devolver ese elemento, ya que al ser el único elemento de la lista también es el último. Ahora, si nuestra lista tiene por lo menos dos elementos x y z al principio, z puede ser el último si la cola es una lista vacía. La otra opción es caer en el mismo patrón, por esa razón usamos recursión y obtenemos el último elemento de la lista con cabeza z y cola zs, con la misma función.

Ejercicios.

Para esta práctica programarás algunas funciones en Haskell. Debes usar las firmas que están en cada ejercicio.

1. **Máximo común Divisor**. La función debe calcular el MCD de dos números enteros positivos.Puedes asumir que la función siempre recibirá enteros positivos **(1 punto)**

   ```
   mcd :: Int -> Int -> Int
   ```

2. **Mínimo común Múltiplo**. La función debe calcular el MCM de dos números enteros positivos.Puedes asumir que la función siempre recibirá enteros positivos **(1 punto)**

   ```
   mcm :: Int -> Int -> Int
   ```

3. **Longitud de una lista**.Función recursiva que calcula la longitud de una lista. **(1 punto)**

   ```
   longitud :: [a] -> Int
   ```

4. **Máximo de una lista**. La función devuelve el máximo elemento de una lista de tipo numérico. **(1 punto)**

   ```
   maximo :: Num a => [a] -> a
   ```

5. **Reversa**. La función calcula recursivamente la reversa de una lista. **(1 punto)**

   ```
   reversa :: [a] -> [a]
   ```

6. **Palíndromo**. La función verifica si una lista es palíndromo. **(1 punto)**

   ```
   palindromo :: [a] -> Bool
   ```

7. **Divisores**. La función devuelve una lista con todos los divisores positivos de un número entero positivo. Puedes asumir que la función siempre va a recibir enteros positivos. **(1 punto)**

   ```
   divisores :: Int -> [Int]
   ```

8. **Diferencia Simétrica**. La función debe calcular la diferencia simétrica entre dos listas. **(1 punto)**

   ```
   diferenciaSimetrica :: [a] -> [a] -> [a]
   ```

9. **Multiplicación de matrices**. La función debe calcular la multiplicación de las matrices M y N de dimension nxm y mxk respectivamente. Para representar una matriz en Haskell usaremos un lista de listas. Puedes asumir que siempre vas a recibir matrices válidas para la multiplicación. **(2 puntos)**

   ```
   multMatriz ::Num a => [[a]] -> [[a]] -> [[a]]
   ```

10. **Conjunto potencias**. La función debe calcular el conjunto potencia de una lista. **(2 puntos)**

    ```
    conjuntoPotencia :: [a] -> [[a]]
    ```

Entrega

Deberás entregar un archivo llamado **practica2.hs** (respetando el nombre y la extensión del archivo) junto con el archivo **ReadMe.txt** conforme a los lineamientos de entrega de prácticas. La entrega es a través de la plataforma *classroom*, en la actividad asignada a la tarea. Se tiene como límite las **23:59** de la fecha de entrega, cada día de retraso se penalizará con puntos menos.

¡Que tengas éxito!