Lenguajes de Programación, 2021-1
Práctica 1: Introducción a Haskell λ

Favio E. Miranda Perea
favio@ciencias.unam.mx

Javier Enríquez Mendoza
javierem_94@ciencias.unam.mx

L. Fernando Loyola Cruz
loyola@ciencias.unam.mx

Fecha de entrega: *9 de Octubre de 2020*

Objetivo

Que el alumno se familiarice con conceptos básicos del lenguaje de programación funcional `Haskell` como lo son las listas, *pattern matching*, recursión, etc. así como hacer uso de estos para definir funciones con la finalidad de resolver un problema en particular.

Números Naturales

El cielo nocturno contiene más estrellas de las que puedo contar, aunque menos de cinco mil son visibles a simple vista. El universo observable contiene alrededor de setenta billones de estrellas.
Pero el número de estrellas es finito, mientras que los números naturales son infinitos.
Cuente todas las estrellas y aún le quedarán tantos números naturales como al principio.

Todo el mundo está familiarizado con los números naturales.

```
0, 1, 2, 3, ...
```

Usualmente usamos el símbolo $\mathbb{N}$ para hacer referencia al tipo de números naturales, y decimos que 0, 1, 2, 3, etc. son valores de tipo $\mathbb{N}$, lo cual se puede representar escribiendo $0 : \mathbb{N}, 1 : \mathbb{N}, \ldots$

El conjunto de números naturales es infinito, pero podemos escribir su definición en unas pocas líneas. Aquí está la definición como un par de reglas de inferencia:

```
                                          m : ℕ
        --------                      -----------
        cero : ℕ                      suc m : ℕ
```

Lo anterior se puede ver como una definición inductiva que nos dice lo siguiente:

- Caso base: El `cero` es un número natural.
- Caso inductivo: Si `m` es un número natural entonces el sucesor de `m` es un número natural.

Además, estas dos reglas nos permiten crear números naturales de manera única. Por tanto, los posibles números naturales son:

```
cero
suc cero
suc (suc cero)
...
```

Escribimos `0` como abreviatura de *cero*; y 1 es la abreviatura de *suc cero*, el sucesor del cero, es decir, el natural que viene después del cero; y 2 es la abreviatura de *suc (suc cero)*, que es lo mismo que *suc 1*, el sucesor de uno; y asi sucesivamente.

Ejercicios

1. Da la definición del tipo de dato `Nat` que corresponde a la definición de los números naturales.
2. Define la función `suma` que recibe dos números naturales `n` y `m` y regresa un número natural `k` que representa la suma de `n` y `m`.

```
suma :: Nat -> Nat -> Nat

{- Ejemplo -}
$ suma (suc cero) (suc (suc cero))
    suc ( suc ( suc ( cero ) ) )
```

3. Define la función `sub` que recibe dos números naturales `n` y `m` y regresa `cero` si sucede que `n` $\leq$ `m` o un número natural `k` que representa la resta de `n` y `m`.

```
sub :: Nat -> Nat -> Nat

{- Ejemplo -}
$ sub (suc (suc (suc cero)))  (suc (suc cero))
    suc (cero)
```

4. Define la función `mul` que recibe dos números naturales `n` y `m` y regresa un número natural `k` que representa la multiplicación de `n` y `m`.

```
mul :: Nat -> Nat -> Nat

{- Ejemplo -}
$ mul suc (suc cero)  (suc (suc cero))
    suc (suc (suc (suc (cero))))
```

5. Define la función `menorQue` que recibe dos números naturales `n` y `m` y regresa un booleano que indica si el primer argumento es menor al segundo.

```
menorQue :: Nat -> Nat -> Bool

{- Ejemplo -}
$ menorQue (suc cero)  (suc (suc cero))
    True
```

6. Define la función `eq` que recibe dos números naturales `n` y `m` y regresa un booleano que indica si `n` representa el mismo número que `m`.

```
eq :: Nat -> Nat -> Bool

{- Ejemplo -}
$ eq (suc (suc cero))  (suc (suc cero))
    True
```

7. Define la función `par` que recibe un número natural `n` y regresa un booleano que indica si `n` es par.

```
par :: Nat -> Bool

{- Ejemplo -}
$ par (suc (suc cero))
    True
```

8. Define la función `impar` que recibe un número natural `n` y regresa un booleano que indica si `n` es impar.

```
impar :: Nat -> Bool

{- Ejemplo -}
$ impar (suc cero)
    True
```

9. Define la función `toInt` que transforma un número natural a un número entero de Haskell.

```
toInt :: Nat -> Int

{- Ejemplo -}
$ toInt (suc (suc (suc cero)))
    3
```

10. Una función parcial es aquella que no esta definida sobre todo su dominio. La función que transforme un número entero a un número natural es una función parcial pues no hay manera de transformar un número negativo a un número natural.

    10.1. Define la función `toNat` que transforma un número entero de Haskell a un número natural. Haz uso de la función predefinida `error` para manejar el caso cuando la entrada es un número negativo.

```
toNatError :: Int -> Nat

{- Ejemplo -}
$ toNatError -1
    *** Exception: Hubo un error.

$ toNatError 2
    suc (suc cero)
```

10.2. El problema de usar la función `error` es que provoca que la evaluación se detenga. Una manera de evitar esto es usando el tipo de dato `Maybe`. Modifica la función `Nat` usando el tipo de dato `Maybe` para manejar el caso de los números negativos.

```
toNatMaybe :: Int -> Maybe Nat

{- Ejemplo -}
$ toNatMaybe -1
    Nothing

$ toNatMaybe 2
    Maybe (suc (suc cero))
```

Conjuntos

Olvídese de todo lo que sabe sobre números.
De hecho, olvídese de que sabe lo que es un número.
Aquí es donde comienzan las matemáticas.
En lugar de matemáticas con números, ahora pensaremos en matemáticas con "cosas".

En computación un *conjunto* es un tipo de dato abstracto que puede almacenar valores únicos sin ningún orden en particular. Muchos lenguajes de programación que pertenecen al paradigma imperativo ya traen incluida esta estructura de forma nativa y en muchos de los casos la forma en como se implementa un conjunto es con ayuda de los *arreglos* permitiendo realizar ciertas operaciones a un costo computacional muy bajo.
Dado que `Haskell` es un lenguaje de programación funcional, tratar de implementar un conjunto de manera similar a como se haría en un lenguaje imperativo es casi imposible, sin embargo podemos optar por una implementación usando listas.

En `Haskell` podemos definir el tipo de dato `Set` como un sinónimo del tipo de dato `Lista` de la siguiente manera:

```
type Set a = [a]
```

Nota: Debemos especificar que los elementos con los que vamos a trabajar puede ser comparados entre si. Para eso hacemos uso de *clases de tipo* como `Eq`.

Ejercicios

1. Define la función `pertenece` que recibe un elemento y regresa un booleano indicando si el elemento existe en el conjunto.

```
pertenece :: (Eq a) => a -> Set a -> Bool

{- Ejemplo -}
$ pertenece 2 [1,2,3,4]
    True
$ pertenece 10 [1,2,3]
    False
```

2. Dado que estamos definiendo los conjuntos con listas, no hay manera de asegurar "estáticamente" que lo que decimos que es un conjunto en efecto lo sea.
   Define una función `esConjunto` que determina si un elemento del tipo `Set` es un conjunto según la definición dada.

```
esConjunto :: (Eq a) => Set a -> Bool

{- Ejemplo -}
$ esConjunto [1,2,3,4,5]
    True
$ esConjunto [1,2,3,4,1]
    False
```

3. Define la función `toSet` que reciba una lista y devuelve un conjunto.

```
toSet :: (Eq a) => [a] -> Set a

{- Ejemplo -}
$ toSet [1,1,2,2,3,3]
    [1,2,3]
```

4. Define la función `eq` que reciba una dos conjuntos y determine si son iguales

```
eq :: (Eq a) => Set a -> Set a -> Bool

{- Ejemplo -}
$ eq [5,3,1] [1,5,3]
    True
$ eq [10, 20, 30] [10, 20, 40]
    False
```

5. En `Haskell` las funciones son de primer orden, es decir, podemos recibir funciones como parámetros y regresar funciones.
   Define la función `todos` que recibe una función `f` y un conjunto `A` y determina si todos los elementos de `A` cumplen `f`.

```
todos :: (Eq a) => (a -> Bool) -> Set a -> Bool

{- Ejemplo -}
$ todos (>0) [1,2,3]
    True
```

6. Define la función `alguno` que recibe un conjunto `A`, una función `f` y determina existe un elemento de `A` que cumple `f`.

```
alguno :: (Eq a) => Set a -> (a -> Bool) -> Bool

{- Ejemplo -}
$ alguno (<0) [1,2,-3]
    True
$ alguno (==0) [1,2,3]
```

7. Define la función `agrega` que agrega un elemento a un conjunto. Si el elemento ya existe entonces no es necesario hacer nada.

```
agrega :: (Eq a) => a -> Set a -> Set a

{- Ejemplo -}
$ agrega 10 [1,7,3] -- Dado que el orden no importa, una respuesta
   valida puede ser [1,7,3,10]
    [10,1,7,3]
```

8. Define la función `union` que recibe dos conjuntos y regresa la unión de ellos.

```
union :: (Eq a) => Set a -> Set a -> Set a

{- Ejemplo -}
$ union [1,2,3] [3,4,5]
    [1,2,3,4,5]
```

9. Define la función `interseccion` que recibe dos conjuntos y regresa la intersección de ellos.

```
interseccion :: (Eq a) => Set a -> Set a -> Set a

{- Ejemplo -}
$ interseccion [1,2,3] [2,3,4]
    [2,3]

$ interseccion [1,2,3] [4,5,6]
    []
```

10. Define la función `diferencia` que recibe un conjunto `A`, un conjunto `B` y regresa un conjunto `C` que contiene los elementos de `A` que no están en `B`.

```
diferencia :: (Eq a) => Set a -> Set a -> Set a

{- Ejemplo -}
$ diferencia [1,2,3,4,5], [1,3,5]
    [2,4]
```

11. Define la función `subconjunto` que recibe un conjunto `S` y un conjunto `C` y verifica si `S` es subconjunto de `C`.

```
esSubconjunto :: (Eq a) => Set a -> Set a -> Bool

{- Ejemplo -}
$ esSubconjunto [1,2,3] [5,4,3,2,1]
    True
$ esSubconjunto [] [1,2,3]
    True
$ esSubconjunto [1,3] [2,4]
    False
```