# Ejercicio Semanal 1
Recordando `Haskell`

Favio E. Miranda Perea (favio@ciencias.unam.mx)
Diego Carrillo Verduzco (dixego@ciencias.unam.mx)
Pablo G. González López (pablog@ciencias.unam.mx)

Miércoles 8 de agosto de 2018

**Fecha de entrega: Viernes 17 de agosto de 2018 a las 23:59:59.**

## 1 (5 puntos) Listas `Snoc`

Las listas `Snoc` se construyen de manera similar a las listas que usamos comúnmente, pero a diferencia de ellas el operador primitivo para agregar un elemento lo hace por la derecha.

En *Haskell* se definen del siguiente modo:

```
data ListS a = NilS | Snoc (ListS a) a deriving Show
```

Donde `NilS` representa la lista vacía y `Snoc` representa el operador que agrega al final de la lista un elemento.

La lista `Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5` se ve de este modo:

```
((((<> :: 1) :: 2) :: 3) :: 4) :: 5
```

El elemento `1` es el primer elemento y el elemento `5` es el último.

Si se agrega un nuevo elemento (`10`), la lista quedaría como `Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5) 10` y se vería del siguiente modo:

```
(((((<> :: 1) :: 2) :: 3) :: 4) :: 5) :: 10
```

Implementa las siguientes funciones:

1. `headS`. Obtiene el primer elemento de la lista.

   ```
   headS :: ListS a -> a
   ```

   Ejemplo:

   ```
   *Main> headS NilS
   *** Exception: Empty list
   ```

```
*Main> headS (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
1
```

2. `tailS`. Obtiene la lista sin el primer elemento.

```
tailS :: ListS a -> ListS a
```

Ejemplo:

```
*Main> tailS NilS
*** Exception: Empty list

*Main> tailS (Snoc NilS 1)
NilS

*Main> tailS (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
Snoc (Snoc (Snoc (Snoc NilS 2) 3) 4) 5)
```

3. `initS`. Obtiene la lista sin el último elemento.

```
initS :: ListS a -> ListS a
```

Ejemplo:

```
*Main> initS NilS
*** Exception: Empty list

*Main> initS (Snoc NilS 1)
NilS

*Main> initS (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4)
```

4. `lastS`. Obtiene el último elemento de la lista.

```
lastS :: ListS a -> a
```

Ejemplo:

```
*Main> lastS NilS
*** Exception: Empty list

*Main> lastS (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
5
```

5. `nthElementS`. Obtiene el n-ésimo elemento de la lista.

```
nthElementS :: Int -> ListS a -> a
```

Ejemplo:

```
*Main> nthElementS 5 NilS
*** Exception: Invalid index

*Main> nthElementS 10 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
*** Exception: Invalid index

*Main> nthElementS (-1) (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
*** Exception: Invalid index

*Main> nthElementS 0 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
1

*Main> nthElementS 2 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
3
```

6. `deleteNthElementS`. Elimina el n-ésimo elemento de la lista.

```
deleteNthElementS :: Int -> ListS a -> ListS a
```

Ejemplo:

```
*Main> deleteNthElementS 5 NilS
NilS

*Main> deleteNthElementS 10 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
NilS

*Main> deleteNthElementS (-1) (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
*** Exception: Invalid index

*Main> deleteNthElementS 2 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1)
2) 3) 4) 5)
Snoc (Snoc (Snoc (Snoc NilS 1) 2) 4) 5
```

7. `addFirstS`. Obtiene la lista donde el primer elemento es el elemento dado.

```
addFirstS :: a -> ListS a -> ListS a
```

Ejemplo:

```
*Main> addFirstS 0 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
Snoc (Snoc (Snoc (Snoc (Snoc (Snoc NilS 0) 1) 2) 3) 4) 5
```

8. `addLastS`. Obtiene la lista donde el último elemento es el elemento dado.

```
addLastS :: a -> ListS a -> ListS a
```

Ejemplo:

```
*Main> addLastS 6 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
Snoc (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5) 6
```

9. `reverseS`. Obtiene la reversa de la lista.

```
reverseS :: ListS a -> ListS a
```

Ejemplo:

```
*Main> reverseS (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
Snoc (Snoc (Snoc (Snoc (Snoc NilS 5) 4) 3) 2) 1
```

10. `appendS`. Obtiene la concatenación de dos listas.

```
appendS :: ListS a -> ListS a -> ListS a
```

Ejemplo:

```
*Main> appendS (Snoc (Snoc (Snoc NilS 1) 2)
3) (Snoc (Snoc (Snoc NilS 6) 7) 8)
Snoc (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 6) 7) 8
```

11. `takeS`. Obtiene la lista con los primeros n elementos.

```
takeS :: Int -> ListS a -> ListS a
```

Ejemplo:

```
*Main> takeS 0 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
NilS

*Main> takeS 10 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5

*Main> takeS 2 (Snoc (Snoc (Snoc (Snoc (Snoc NilS 1) 2) 3) 4) 5)
Snoc (Snoc NilS 1) 2
```

## 2 (5 puntos) Números naturales

Los números naturales se pueden definir como:

- `Zero` Representa el número cero (0).
- `D`$x$ Representa al doble de $x$, con $x$ un número natural ($2x$).
- `O`$x$ Representa al sucesor del doble de $x$, con $x$ un número natural ($2x+1$).

Y su respectiva definición en *Haskell* es:

```
data Nat = Zero
         | D Nat
         | O Nat deriving Show
```

Usando esta definición, implementa las siguientes funciones:

1. `toNat`. Obtiene la representación en números `Nat` de un número entero.

   ```
   toNat :: Int -> Nat
   ```

   Ejemplo:

   ```
   *Main> toNat 616
   D (D (D (O (D (O (O (D (D (O (Zero)))))))))
   ```

2. `succ`. Obtiene el sucesor de un número `Nat`.

   ```
   succ :: Nat -> Nat
   ```

   Ejemplo:

   ```
   *Main> succ (D (O (D Zero)))
   O (O Zero)
   ```

3. `pred`. Obtiene el predecesor de un número `Nat`.

   ```
   pred :: Nat -> Nat
   ```

   Ejemplo:

   ```
   *Main> pred (D (O Zero))
   O (Zero)
   ```

4. `add`. Obtiene la suma de dos números `Nat`.

   ```
   add :: Nat -> Nat -> Nat
   ```

   Ejemplo:

   ```
   *Main> add (D (O (D Zero))) (D (O Zero))
   D (O (O Zero))
   ```

5. `prod`. Obtiene el producto de dos números `Nat`.

```
prod :: Nat -> Nat -> Nat
```

Ejemplo:

```
*Main> prod (D (O (D Zero))) (D (O Zero))
D (D (O Zero))
```

**¡Suerte!**