# Programación Declarativa 2020-2
# Facultad de Ciencias UNAM
# Tarea 3: Bringing you into the fold

Favio E. Miranda Perea  Javier Enríquez Mendoza

Fecha de entrega: 12 de marzo de 2020

## Parte Teórica

1. Demuestra las siguientes propiedades de los operadores de plegado:

   (a) `foldr f e . map g = foldr (f . g) e`

   (b) `foldl f e xs = foldr (flip f)  e (reverse xs)`

   (c) `foldr f e (xs ++ ys)= foldr f (foldr f e ys)xs`

2. Considera el siguiente tipo de dato algebraico en Haskell para definir árboles binarios.

   ```
   data Tree a = Void | Node (Tree a) a (Tree a)
   ```

   Y la función `foldT` que define el operador de plegado para la estructura `Tree`, definido como sigue:

   ```
   foldT :: (b -> a -> b -> b) -> b -> Tree a -> b
   foldT _ v Void = v
   foldT f v (Node t1 r t2) = f t1' r t2'
    where t1' = foldT f v t1
          t2' = foldT f v t2
   ```

   (a) Da en términos de una función `h` el patrón encapsulado por el operador `foldT`.

   (b) Enuncia y demuestra la propiedad Universal del operador `foldT`, basándote en la Propiedad Universal vista en clase sobre el operador `foldr` de listas.

3. Calcula una definición eficiente para `scanr` partiendo de la siguiente:

   ```
   scan r f e = map (foldr f e) . tails
   ```

4. Considera la siguiente definición de la función `cp` que calcula el producto cartesiano.

   ```
   cp :: [[a]] -> [[a]]
   cp = foldr f e
   ```

(a) En la definición anterior ¿Quiénes son `f` y `e`?

(b) Dada la siguiente ecuación

```
length . cp = product . map length
```

en donde `length` calcula la longitud de una lista y `product` regresa el resultado de la multiplicación de todos los elementos de una lista. Demuestra que la ecuación es cierta, para esto es necesario reescribir ambos lados de la ecuación como instancias de `foldr` y ver que son idénticas.

**Extra** *Hasta 0.5 puntos sobre la tarea* En una granja con mucho folklore se discute acerca del siguiente razonamiento. El día que nace un becerro, cualquiera lo puede cargar con facilidad. Y los becerros no crecen demasiado en un día, entonces si puedes cargar a un becerro un día, lo puedes cargar también al día siguiente, siguiendo con este razonamiento entonces también debería serte posible cargar al becerro el día siguiente y el siguiente y así sucesivamente.

Pero después de un año, el becerro se va a convertir en una vaca adulta de 1000kg algo que claramente ya no puedes cargar.

Este es un razonamiento inductivo, la base es el día que el becerro nace, suponemos cierto que se puede cargar en el día $n$ de vida del becerro y si se puede cargar ese día, como no crece mucho en un día entonces también se puede cargar en el día $n+1$. Pero claramente la conclusión es falsa.

Para este ejercicio hay dos posibles soluciones, la primera es indicar en donde está el error en el razonamiento inductivo o la segunda es cargar una vaca adulta así demostrando que el argumento es correcto.

## Parte Práctica

1. Reescribe las siguientes funciones usando la técnica de programación origami, es decir como instancias de `foldr`

   (a) `concat :: [[a]] -> [a]`

   (b) `minimum :: (Ord a)=> [a] -> a`

   (c) `reverse :: [a] -> [a]`

   (d) `filter :: (a -> Bool)-> [a] -> [a]`

   (e) `inits :: [a] -> [[a]]`

2. Podemos definir el operador de plegado para el tipo `Int` de la siguiente manera:

   ```
   foldi :: (a -> a) -> a -> Int -> a
   foldi f q 0 = q
   foldi f q i = f (foldi f q (pred i))
   ```

   Define las funciones `add`, `mult` y `expt` como instancias de `foldi`.

3. Utiliza las funciones de orden superior `foldr`, `foldl`, `scanr` o `scanl` para definir las siguientes funciones

   (a) `sumq :: Int -> Int` que recibe un entero $n$ y regresa la suma de los cuadrados de los primeros $n$ naturales. Es decir `sumq` $n = 0^2 + 1^2 + ... + (n-1)^2$

(b) `remove :: (Eq a)=> [a] -> [a] -> [a]` que es una función que toma dos listas como parámetros y elimina todos los elementos de la segunda lista que aparecían en la primera. Por ejemplo `remove "first""second"= "econd"`

(c) `remdups :: (Eq a)=> [a] -> [a]` que elimina los elementos adyacentes duplicados en una lista, por ejemplo `remdups [1,2,2,3,3,3,1,1] = [1,2,3,1]`

(d) `rotate :: [a] -> [[a]]` que produce todas las posibles rotaciones de una lista, ejemplo `rotate [1, 2, 3] = [[1, 2, 3], [2, 3, 1], [3, 1, 2]]`. **Hint:** Definir primero una función `shift` (también en origami) que calcula una rotación de la lista.

4. Completa la siguiente definición de la función `unmerge` de tal forma que sea equivalente a la vista en clase.

```
unmerge :: (Ord a) => [a] -> [([a],[a])]
unmerge xs = [(ys,zs) | merge ys zs == xs ...
```

## Entrega

- La tarea puede entregarse de forma individual o en parejas.
- Se tomará en cuenta tanto el estilo de programación como la complejidad de las soluciones para la calificación de la sección práctica.
- La parte práctica de la tarea se entrega a través de Slack:
  - Todos los archivos deben tener al principio como comentario los nombres de los alumnos.
  - Si la tarea se realizo en parejas crear un canal privado con el ayudante y los miembros del equipo para la entrega.
  - Si se realizo de forma individual enviarla en un mensaje directo al ayudante.
  - **No** deben comprimir los archivos, en caso de tener mas de uno enviarlos por separado.
- La parte teórica se puede entregar en PDF (obtenido a partir de LaTeX) junto con la parte práctica o a mano en la **ayudantia** previa a la fecha de entrega.