# Taller de Programación Funcional

Verano 2018

Fecha de entrega: 14 de Febrero del 2018

## 1. Introducción

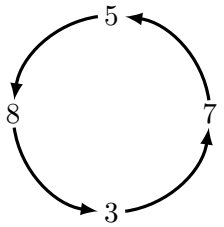
Durante este taller trabajaremos con una estructura que permite representar anillos en Haskell. Un *anillo* es lo que en Teoría de Grafos se conoce como un ciclo simple. Es decir, un conjunto de vértices y ejes que forman un camino cerrado que no repite ejes ni nodos a excepción del vértice inicial. En este caso, la estructura de datos no contendrá elementos repetidos. La misma está definida como:

```
data Anillo t = A t (t -> Maybe t)
```

Contamos con las funciones:

- `actual::Anillo t -> t`, que proyecta el primer argumento del constructor (el elemento actualmente en consideración);
- `siguiente::Anillo t -> t -> Maybe t`, que proyecta el segundo argumento del constructor (el cual define cómo están vinculados los elementos).

Consideremos el anillo de enteros.




Este anillo puede definirse como:

```
anilloEjemplo::Anillo Integer
anilloEjemplo = A 5 proximo
       where proximo n | n == 5 = Just 8
                       | n == 8 = Just 3
                       | n == 3 = Just 7
                       | n == 7 = Just 5
                       | otherwise = Nothing
```

Notar que este anillo puede ser definido también como se muestra a continuación, dado que el elemento `9` no es alcanzable desde el nodo actual.

```
anilloEjemplo'::Anillo Integer
anilloEjemplo' = A 5 proximo
       where proximo n | n == 5 = Just 8
                       | n == 8 = Just 3
                       | n == 3 = Just 7
                       | n == 7 = Just 5
                       | n == 9 = Just 9
                       | otherwise = Nothing
```

Recordar que el módulo `Data.Maybe` provee la función `fromJust::Maybe t-> t` que proyecta el valor `v` cuando es aplicada a `Just v`, y no está definida para `Nothing`.

El taller contiene dos archivos:

- `Util.hs`: contiene todo necesario para poder trabajar con anillos (el tipo de dato, las funciones, y el `anilloEjemplo`). Este archivo no podrá ser modificado.
- `Main.hs`: contiene las funciones para implementar.

# 2. Implementación

## Ejercicio 1

(a) `singleton::Eq t => t -> Anillo t`, que genera un anillo unitario a partir de un elemento dado.

(b) `insertar::Eq t => t -> Anillo t -> Anillo t` que, a partir de un elemento `e` y un anillo `a`, genera un nuevo anillo que se obtiene agregando `e` inmediatamente a continuación del elemento actual de `a`. El elemento actual del nuevo anillo coincide con el actual de `a`. Puede asumirse que `e` no pertenece a `a`.

(c) `avanzar::Anillo t -> Anillo t`, que permite "pasar" al siguiente elemento del anillo.

## Ejercicio 2

Definir la función `enAnillo::Eq t => t -> Anillo t -> Bool` que dados un elemento `e` y un anillo `a`, `enAnillo e a` evalúa `True` si y sólo si `e` pertenece a `a`. Por ejemplo:

`enAnillo 3 anilloEjemplo` $\rightsquigarrow$ `True` | `enAnillo 12 anilloEjemplo` $\rightsquigarrow$ `False`

**Aclaración:** no basta con que `siguiente a e` evalúe distinto de `Nothing`. Hace falta verificar que `e` sea accesible desde el elemento actual de `a`. Notar que `enAnillo` debe definirse de manera tal que `enAnillo 9 anilloEjemplo'` $\rightsquigarrow$ `False`

## Ejercicio 3

Escribir la función `filterAnillo:: (t -> Bool) -> Anillo t -> Maybe (Anillo t)`. Esta función recibe como argumentos un anillo `a` y un predicado `p` y devuelve el anillo que se obtiene a partir de `a` manteniendo sólo los elementos que satisfacen `p`. Notar que el resultado puede ser Nothing si ningún elemento del anillo original pasa el filtro.

`enAnillo 8 (fromJust (filterAnillo (>5) anilloEjemplo))` $\rightsquigarrow$ `True`
`enAnillo 3 (fromJust (filterAnillo (>5) anilloEjemplo))` $\rightsquigarrow$ `False`

## Ejercicio 4

Dar la función `mapAnillo::Eq a => Eq b => (a ->b) -> Anillo a -> Anillo b`, que aplique una función a todos los elementos de un anillo. Puede asumir que la función a aplicar es inyectiva sobre los elementos del anillo, es decir, a elementos distintos le corresponden distintas imágenes. Por ejemplo:

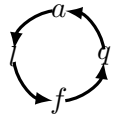
```
enAnillo 5 (mapAnillo (‘mod‘ 6) anilloEjemplo)  ⇝  True
enAnillo 2 (mapAnillo (‘mod‘ 6) anilloEjemplo)  ⇝  True
enAnillo 3 (mapAnillo (‘mod‘ 6) anilloEjemplo)  ⇝  True
enAnillo 1 (mapAnillo (‘mod‘ 6) anilloEjemplo)  ⇝  True
actual (mapAnillo (*2) anilloEjemplo)           ⇝  10
```

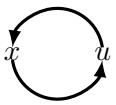
**Importante:** La solución no debe convertir al anillo en una lista y aplicar el map sobre listas.

## Ejercicio 5

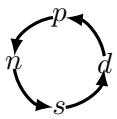
Definir la función `palabraFormable::String -> [Anillo Char] -> Bool`. A partir de una lista `as` de anillos de caracteres y una cadena de caracteres `s`, esta función determina si existe alguna manera de posicionar cada anillo en `as` de forma tal que `s` quede expresada en término de los actuales de cada anillo. Asumir que `s` y `as` tienen la misma longitud.

Por ejemplo, si consideramos la lista `as = [a1, a2, a3, a4]` con los siguientes anillos:

Anillo a1: 


Anillo a2: 


Anillo a3: 


Anillo a4: 

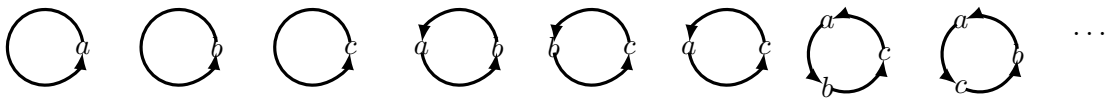

```
palabraFormable ’’luis’’ as ⇝ True
palabraFormable ’’flan’’ as ⇝ False
palabraFormable ’’quid’’ as ⇝ True
```

## Ejercicio 6

Definir la función `anillos::[a] -> [Anillo a]`. A partir de una lista `xs` de elementos, devuelve la lista de todos los anillos que contienen un subconjunto de elementos de `xs`.

**Aclaración:** queda a criterio de los integrantes, distinguir o no a dos anillos que difieren solo en su elemento actual.

Por ejemplo, `anillos ’’abc’’` ⇝



## Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;Taller-PF]` seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: Es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del Taller es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

**Importante:** se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Para instalar HUnit usar: `>cabal install hunit`

Para instalar cabal ver: `https://wiki.haskell.org/Cabal-Install`

## Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en: http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos parciales, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquellos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.