# Trabajo Práctico 1
# Programación Funcional

Paradigmas de Lenguajes de Programación — $2^{do}$ cuat. 2022

Fecha de entrega: 13 de septiembre

## 1. Introducción

Un diccionario, mapa o array asociativo es una estructura de datos que asocia *claves* o *keys* de cierto tipo $A$ con *valores* o *values* de otro tipo $B$.

Por ejemplo, el siguiente diccionario asocia claves de tipo `String` con valores de tipo `Int`:

```
{
 "a" : 1,
 "b" : 2,
 ...
}
```

El estándar *JSON*[1] nos permite anidar diccionarios de la siguiente manera:

```
{
 "a": 1,
 "b": {
      "a": 2,
      "d": 4
     },
 "c": 3
}
```

```
{
 "a": 1,
 "b": {
      "b": 1,
      "c": {
           "c": 2,
           "d": 4
           }
     }
}
```

De este modo definimos los *multidiccionarios*, que permiten asociar a cada clave, o bien un valor, o bien otro multidiccionario.

En este trabajo práctico, implementaremos un *multidiccionario* en *Haskell*.

## 2. Implementación

### 2.1. Tipos de datos utilizados

Definimos el siguiente tipo de datos algebraico para nuestro multidiccionario:

```
data MultiDict a b = Nil |
  Entry a b (MultiDict a b) |
  Multi a (MultiDict a b) (MultiDict a b) deriving Eq
```

donde `Nil` es el multidiccionario vacío, `Entry k v m` es el multidiccionario que resulta de definir la clave `k` con valor `v` en el multidiccionario `m`, y `Multi k s m` es el resultado de agregar a `m` la clave `k`, asociándole como valor el multidiccionario `s`.

[1] https://www.json.org/

Por ejemplo, los multidiccionarios representados arriba se construyen como:
`Entry "a" 1 $ Multi "b" (Entry "a" 2 $ Entry "d" 4 Nil) $ Entry "c" 3 Nil` y
`Entry "a" 1 $ Multi "b" (Entry "b" 1 $ Multi "c" (Entry "c" 2 $ Entry"d" 4 Nil) Nil) Nil`.

## Ejercicio 1

Definir las siguientes funciones:

- `foldMD`, que representa el esquema de recursión estructural *foldr* sobre *MultiDict*. Dar el tipo de esta función.
- `recMD`, que representa el esquema de recursión primitiva *recr* sobre *MultiDict*.

Por tratarse de esquemas de recursión, para definir estas funciones se permite utilizar **recursión explícita**.

## Ejercicio 2

Definir las siguientes funciones utilizando alguno de los esquemas definidos en el punto anterior:

- `profundidad :: MultiDict a b -> Integer`, que dado un *MultiDict* nos devuelve la cantidad máxima de niveles que tiene.

```
> profundidad Nil
0
> profundidad $ Entry 'a' 1 Nil
1
> profundidad $ Entry 'a' 1 (Entry 'b' 2 Nil)
1
> let izq = (Entry 'a' 1 (Entry 'b' 2 Nil))
> let der = (Entry 'c' 3 (Entry 'd' 4 Nil))
> profundidad $ Multi 'x' izq Nil
2
> profundidad $ Multi 'x' izq der
2
```

- `tamaño :: MultiDict a b -> Integer` que dado un *MultiDict* nos devuelve cuántas claves tiene definidas en todos los niveles.

```
> tamaño Nil
0
> tamaño $ Entry 'a' 1 Nil
1
> tamaño $ Entry 'a' 1 $ Entry 'b' 2 Nil
2
> let izq = (Entry 'a' 1 (Entry 'b' 2 Nil))
> let der = (Entry 'c' 3 (Entry 'd' 4 Nil))
> tamaño $ Multi 'x' izq der
5
```

## Ejercicio 3

Definir la función `tablas :: Integer -> MultiDict Integer Integer`, que dado un entero `n`, devuelve un diccionario de profundidad 2, siendo las claves todos los naturales mayores o iguales que `n`. A cada número se asocia un diccionario con su tabla de multiplicar; es decir, para el número `i`, obtenemos un diccionario cuyas claves son los naturales de 1 en adelante, donde el valor para cada clave `j` es el producto de `i*j`.

Dado que se requiere generar una estructura infinita, para este ejercicio se permite usar **recursión explícita**.

Para poder observar los resultados de esta función y otros multidiccionarios infinitos, les proveemos `podar`, que dados una `longitud`, una `profundidad` y un *MultiDict*, nos devuelve otro *MultiDict* dejando sólo `longitud` claves por cada nivel y `profundidad` niveles.

Ejemplos:

```
> podar 5 2 (tablas 3)
  {
   3: {
        1: 3,
        2: 6,
        3: 9,
        4: 12,
        5: 15
       },
   4: {
        1: 4,
        2: 8,
        3: 12,
        4: 16,
        5: 20
       },
   5: {
        1: 5,
        2: 10,
        3: 15,
        4: 20,
        5: 25
       },
   6: {
        1: 6,
        2: 12,
        3: 18,
        4: 24,
        5: 30
       },
   7: {
        1: 7,
        2: 14,
        3: 21,
        4: 28,
        5: 35
       }
  }
```

```
> podar 5 3 $ tablas 10
{
 10: {
      1: 10,
      2: 20,
      3: 30,
      4: 40,
      5: 50
     },
 11: {
      1: 11,
      2: 22,
      3: 33,
      4: 44,
      5: 55
     },
 12: {
      1: 12,
      2: 24,
      3: 36,
      4: 48,
      5: 60
     },
 13: {
      1: 13,
      2: 26,
      3: 39,
      4: 52,
      5: 65
     },
 14: {
      1: 14,
      2: 28,
      3: 42,
      4: 56,
      5: 70
     }
}
```

## Ejercicio 4

Definir la función `serialize`, que dado un *MultiDict* nos devuelve un `String` que contiene una representación serializada de este de la siguiente manera, asumiendo que los tipos `a` y `b` implementan la clase `Show`.

- `Nil` se serializa como `[ ]`.
- `Entry a b md` se serializa como `[sa: sb, sm]` donde `sa` y `sb` son las representaciones de `a` y `b` respectivamente, y `sm` es la serialización de `md`.
- `Multi a md1 md2` se serializa como `[sa: sm1, sm2]` donde `sa` se define como en el punto anterior, y `sm1` y `sm2` son las serializaciones de `md1` y `md2`.

Ejemplo:

```
> nodo = Entry 'a' 1 $ Entry 'b' 2 Nil
> doble = Multi 'k' nodo $ Multi 'w' nodo Nil
> serialize nodo
"['a': 1, ['b': 2, [ ]]]"
> serialize doble
"['k': ['a': 1, ['b': 2, [ ]]], ['w': ['a': 1, ['b': 2, [ ]]], [ ]]]"
```

## Ejercicio 5

Implementar las funciones:

- `mapMD`, que toma dos funciones
  - `f :: a -> c`
  - `g :: b -> d`

  y traduce un `MultiDict a b` a un `MultiDict c d`.
- `filterMD` que toma un predicado `p :: a -> Bool` y un `MultiDic a b` y nos devuelve un *MultiDict* del mismo tipo con aquellas claves que cumplen el predicado (filtrando las claves de todos los niveles).
- `enLexicon`, que toma una lista de Strings en minúscula y un `MultiDic String b`, y devuelve el multidiccionario que resulta de pasar a minúsculas todas las claves del multidiccionario original, y descartar aquellas que no pertenezcan a la lista dada.

Ejemplos:

```
> let dic = Entry "A" 1 $ Multi "B" (Entry "Aa" 2 $ Entry "C" 3 $ Nil) $ Entry "D" 4 $ Nil
> mapMD (++ "a") (+ 1) dic
{
 "Aa": 2,
 "Ba": {
      "Aaa": 3,
      "Ca": 4
     },
 "Da": 5
}
> filterMD ((< 'C') . head) dic
{
 "A": 1,
 "B": {
      "Aa": 2
     }
}
```

```
> enLexicon ["aa", "aaa", "b", "d"] dic
{
 "b": {
      "aa": 2
     },
 "d": 4
}
```

### Ejercicio 6

Implementar la función `cadena`, que dados un elemento `v` de los valores y una cadena de claves `[c1, c2, ..., cn]`, devuelve un multidiccionario que resulta de agregar a cada nivel las sucesivas claves `c1, c2, ..., cn` hasta finalmente llegar a una entrada con el valor `v` seguida de `Nil`.

Ejemplos:

```
> cadena 1 "abc"
{
 'a': {
      'b': {
           'c': 1
               }
     }
}
```

```
> cadena "abc" [1,2,3]
{
 1: {
      2: {
           3: "abc"
         }
     }
}
```

### Ejercicio 7

Implementar la función `obtener` que dados una lista de claves `[c1, c2, ..., cn]` y un *MultiDict* `md` nos devuelve un `Maybe b` que consiste en el resultado de obtener el valor al final de la rama `c1, c2 ..., cn` en el caso de estar definido, o `Nothing` si no lo está.

La función `definir`, que permite asociar un valor a una rama (lista de claves), ya está provista por la cátedra.

Ejemplos:

```
>let dic = definir "abd" 1 $ cadena 2 "abc"
> obtener "a" dic
Nothing
> obtener "abd" dic
Just 1
> obtener "abc" dic
Just 2
> obtener "abcd" dic
Nothing
> obtener "af" dic
Nothing
```

## Tests

Parte de la evaluación de este Trabajo Práctico es la realización de tests. Tanto HUnit[2] como HSpec[3] permiten hacerlo con facilidad.

En el archivo de esqueleto que proveemos se encuentran tests básicos utilizando *HUnit*. Para correrlos, ejecutar dentro de *ghci*:

```
> :l tp1.hs
[1 of 2] Compiling MultiDict        ( MultiDict.hs, interpreted )
[2 of 2] Compiling Main             ( tp1.hs, interpreted )
Ok, modules loaded: MultiDict, Main.
> main
```

Para instalar HUnit usar: `> cabal install hunit` o bien `apt install libghc-hunit-dev`.

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

[2] https://hackage.haskell.org/package/HUnit
[3] https://hackage.haskell.org/package/hspec

## 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;TP-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 TP 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.

## 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.