# Miércoles 20/08/2014

## ESTRUCTURAS DE DATOS : Introducción a la materia

### Roadmap

1. **Modelo Puro (mención de Haskell)**
   - tipos algebraicos
   - tipos abstractos
   - Herramientas Conceptuales -> Metodología:
     - Contrato (Propósito y Precondiciones) : Es fundamental documentar para que el código sea claro. Es importante que el contrato se documente ANTES de implementarse la solución.
     - Recorrido : Es una manera de procesar una lista de a un elemento a la vez incorporando ese elemento al resultado final.
       - Iterativo
       - Recursivo (Estructural)
2. **Modelo Destructivo (mención de híbrido de C/C++)**
   - Memoria
   - Tipos abstractos, listas, etc...
   - Estructuras imperativas
3. **Temas adicionales**: Sorting, paneo rápido de grafos

Se aconseja fuertemente hacer función las soluciones en una computadora.
Se habla de los parciales: el primero apenas termina el modelo puro,el segundo es un TP en Cpp.

Se reforzó la parte ética, acerca de no copiarse y sus consecuencias si lo hacen.
También sobre la ABSOLUTA NECESIDAD de hacer las prácticas cuando se las da para estar bien preparados, y la importancia de ENTENDER LAS IDEAS que se intentan transmitir, no solo resolver los problemas.
Lo mismo para el TP final. NO COLGAR!
Mencionan la bibliografía y las intenciones de actualizarla. Se recomienda aprender sintaxis Haskell POR SU CUENTA!

Página de la materia: https://sites.google.com/site/tpiedunq/home.

---

### Repaso de Recorrido de Listas

Diferencia constructores entre CGobstones y XGobstones

| CGobstones | XGobstones |
|---|---|
| Nil | [] |
| Cons(1, Cons(2, Cons(3, Nil))) | [1,2,3] |

*Aclaración para la gente que viene de XGobstones: Cons es una operación que agrega un elemento adelante de otra lista.*

Implementación en XGbostones:

```
        function cons(x, xs)
        { return ( [x] ++ xs ) }
```

Nil devuelve una lista vacía.

```
        function nil()
        { return ( [] ) }
```

Cálculo de la longitud de una lista con recorrido:

```
    function len(xs)
        /*
        PROPOSITO: retorna la cantidad de elementos de la lista xs
        PRECONDICION: Ninguna
        OBS: usamos un recorrido sobre los elementos
         */
    {
        // Inicializacion del recorrido
        cant := 0
        aRecorrer := xs
        while (not isNil(aRecorrer))        // Hay mas elementos
        {
            cant := cant + 1                // Procesar el elemento actual
            aRecorrer := tail(aRecorrer)    // Pasar al siguiente
        }
        // Finalizacion del recorrido
        return (cant)
    }
```

/* Promesa de probar que aplicando el contador no le va a función si lo pone primero (en referencia a que primero pensamos la estructura del problema) */

**Ejercicio para entregar (pre-break):**
aumentarEn10(xs) // Devuelve una lista con cada elemento de una lista xs multiplicado por 10 (varios se dan cuenta que la lista va a quedar al revés y consultan, los ayudantes se los dejamos pasar)

**Revisión de problemas encontrados en las soluciones:**

1. se olvidó la operación de "Pasar al siguiente". Refleja que no se pensó en la estructura del problema.
2. nombres de las variables no describen correctamente lo que deberían.
3. no se divide en subtareas. Si bien no es un problema necesariamente en este caso, pero hacerlo ayuda a separar diferentes problemas en diferentes funciones.
4. al intentar agregar el elemento procesado a la lista a devolver lo hacían concatenando ese elemento a una lista vacía. O sea: aDevolver := nil() ++ [ procesarElemento( head(aRecorrer) ) ]

**Se plantea una solución adecuada al problema:**

```
function aumentarEn10(xs)
{
    // Empezar el recorrido
    aDevolver := nil()
    aRecorrer := xs
    while ( not isNil(aRecorrer) )         // hay mas elementos
    {
        aDevolver := aDevolver ++ [ procesarElemento( head(aRecorrer) ) ]
        aRecorrer := tail( aRecorrer )     // Pasar al siguiente
    }
    // Finalizar el recorrido
    return (aDevolver);
}
```

---

## Introducción al Recorrido Recursivo

Se plantea una lista y solo la posibilidad de usar head y tail para re-hacer aumentarEn10.

**** Dibujo del Genio ****

**Progresión de la introducción al pensamiento recursivo:**

Si tengo una lista de elementos, solo pienso de a un elemento a la vez y asumo que el problema ya viene resuelto para el resto de los elementos. Hay un **Genio** que es el que se encarga de darme eso que "ya viene resuelto".
Ej. Para calcular la suma de los números de una lista de números [1,2,3] tomo el primer elemento (1) de la lista, y se lo sumo al resultado que me entrega el genio (que es la suma del resto de los números (2+3) ). De esa manera, al implementarlo quedaría.

```
function sumRec(xs)
{
    return (head(xs) + genio( tail(xs) ));  // 1 + (2+3)
}
```

Si se le presta atención en detenimiento, se puede ver que lo que se espera que haga el genio es exactamente lo que hace sumRec(xs): devolver la suma de todos los elementos de una lista. Por lo tanto se presenta por primera vez la idea de recursión como tal, ya sumRec(xs) deja de ser:

```
function sumRec(xs)
{
    return ( head(xs) + genio( tail(xs) ) )
}
```

para ser:

```
function sumRec(xs)
{
    return ( head(xs) + sumRec( tail(xs) ) )
}
```

O sea, **sumRec(xs) SE LLAMA A SI MISMA!!**. A eso se le llama recursión. Recorrer una lista de elementos con recursión es realizar un recorrido recursivo.
Pero hace falta hacer un ajuste. Como el genio tiene que saber manejar tail(xs), y tail es una operación parcial, si xs fuese vacía habría un problema. Entonces tenemos que distinguir el caso en que xs es vacía y considerarlo por separado.
Por lo tanto sumRec debe contemplar este caso, que llamamos, el CASO BASE: si la lista viene vacía, sumRec(xs) debe devolver 0. Resumiendo

- Si el genio recibe [] entonces devolverá 0 (que es el neutro de la suma) ya que en esta lista no hay elementos pero aún así se espera que genio devuelva un valor que pueda sumarse a head(xs).
- Si el genio recibe una lista con elementos debería entonces devolver la suma de todos estos. (como ya está implementado arriba)

```
function sumRec(xs)
{
    if ( isNil(xs) )
    {
        return (0)          // CASO BASE
    } else {
        return (head(xs) + sumRec( tail(xs) ))    // RECURSION
    }
}
```

---

## HASKELL

Si queremos ahora pasar a sintaxis de Haskell este código podemos limpiarlo y hacer unos ajustes mínimo sobre el mismo.

```
sumRec xs = if ( isNil(xs) )
            then 0                                -- CASO BASE
            else   (head xs) + sumRec (tail xs)   -- CASO DE RECURSION
```

Observaciones:

- se quitaron los return, ya que lo que viene despues del "then" o del "else" es efectivamente lo que la función va a devolver.
- no hay apertura y cierre de bloques con las llaves.
- Los comentarios en linea ya no son // sino --
- Los parámetros que se pasan a la función en Haskell no van entre paréntesis a no ser que el parámetro sea la llamada a otra función que tiene parámetros.
  Por ejemplo :
  - head xs // no necesita paréntesis
  - sumRec (tail xs) // necesita los paréntesis para que Haskell entienda que la lista resultante de (tail xs) va a ser el argumento de sumRec. De otra manera, si se hiciera "sumRec tail xs" a secas Haskell interpretaría que se le están pasando 2 argumentos a sumRec el primero "tail" y el segundo "xs"

Ejercicio aumentarEn10 xs (similar al enunciado anterior) usando recorrido recursivo.

```
aumentarEn10 xs = if ( isNil xs )
                  then []                                          -- CASO BASE
                  else ((head xs) * 10) : (aumentarEn10 (tail xs))     -- CASO DE RECURSION
                  -- otras opciones para el caso de recursión es no usar dos puntos (x:xs) sino ++ [x] ++ [xs]
```

Porque usamos recursión?
Porque es más fácil pensar las soluciones de esta manera. Yo se que cuando tengo una lista, saco un elemento y me queda el resto.
La recursión soluciona muchos tipos de problemas, siempre de la misma forma.

Constructores de Listas en Haskell:

| CGobstones | XGobstones | Haskell |
|---|---|---|
| Nil | [] | [] |
| Cons(1, Cons(2, Cons(3, Nil))) | [1,2,3] | **[1] ++ [2,3]** ó **[1,2] ++ [3]** ó **[1,2,3] ++ []** ó **1:[2,3]** ó **1:2:[3]** ó **1:2:3:[]** |

**Observaciones**

- O sea que ++ es un append al igual que en XGobstones y Cons es **:** (dos puntos)
- En Haskell toda nomeclatura de listas se traduce internamente a la notación cons (dos puntos). Por ejemplo, [1,2] se traduce a (1:(2:[])).
- Lo que delimita un bloque en Haskell es su indentación. Si un bloque de código está más a la derecha que la posición de un if o del nombre de la función entonces ese código esta en el bloque de esa operacion.

## Pattern Matching

Como se había dicho antes, internamente Haskell entiende las listas de la siguiente manera:

- una lista vacía es []
- una lista que tiene elementos es el primer elemento y el resto, o sea que la lista [1,2,3] es 1 y la lista restante (1:[2,3]). En otras palabras, si x es el primer elemento y xs es la lista restante, se puede decir que x:xs representa a cualquiera lista que tenga elementos.
  Por tanto se sabe que:
  head (x:xs) = x
  tail (x:xs) = xs

O sea que como se ve, la estructura de la lista puede descomponerse e identificarse mediante sus constructores más primitivos (:) y [].
Haskell provee una herramienta muy poderosa que permite identificar y acceder a los elementos de una estructura de datos algebraica llamada Pattern Matching. Pattern Matching como lo dice el nombre, es una manera de identificar una estructura utilizando un patrón. Por ejemplo:

```
sum [] = 0                 -- En caso de recibir una lista vacía retornar 0 (CASO BASE)
sum (x:xs) = x + sum xs
{- En caso de recibir una lista que contiene elementos, se identifican "x" como el 1er
   elemento y "xs" como la lista de elementos restantes (que puede ser vacía) y
   se realiza alguna acción sobre esos datos.
   En el caso específico de sum, se realiza el recorrido recursivo.
 -}

mult10 [] = []
mult10 (x:xs) = (x*10) : (mult10 xs)
```

**Implementación de funciones constructores y accesores de Listas en Haskell**

```
snoc xs x = xs ++ [x]
nil = []
head (x:xs) = x
tail (x:xs) = xs
isNil [] = True
isNil (x:xs) = False
```