Programación Declarativa 2016-2
Práctica 4: OCaml

Selene Linares Arévalo Lourdes González Huesca

3 de mayo de 2016
Facultad de Ciencias UNAM

Instrucciones:
Una carpeta, con los archivos de las implementaciones de los siguientes ejercicios, debe llamarse `practica04Apellido` en donde el apellido de identificación es de algún miembro del equipo. La primer línea de cada archivo debe contener el nombre de los miembros del equipo.
La compresión de la carpeta, `practica04Apellido.tar.gz`, debe ser enviada a más tardar el **15 de mayo de 2016** a luglzhuesca@ciencias.unam.mx.

NOTA: Para todos los ejercicios de la práctica, utilizar el manejo de excepciones en los casos en donde consideren necesario. Además, cada función debe incluir un comentario explicando la idea que motivó la implementación dada.

1. **Map de listas**

   Suponer que se tienen dos tipos `t1,t2`. En la ausencia de elementos imperativos ocurre que para cualquier función `f : t1 -> t2` y cualquier lista `xs:t1 List` se satisface que

   $$\texttt{List.map f (List.rev xs)} \equiv \texttt{List.rev (List.map f xs)}$$

   Empleando elementos imperativos, define una función `crack : t1 -> t2` que invalide la equivalencia anterior.

2. **Iterar una función**

   Define una función `tabular : (int -> 'a) -> int -> 'a array` que toma como argumentos una función `f` y un entero `n`, y regresa un arreglo de longitud n donde el elemento en el índice i debe ser `f i`.

   Por ejemplo: `tabular (fun x -> x*x) 5 = [| 0; 1; 4; 9; 16|]`.

3. **Streams**

   Usando la definición de *streams* vista en ayudantía, define las siguientes funciones:

   - `zip_streams` que toma dos streams y regresa un stream de pares, donde la primer entrada de un par es un elemento del primer stream y la segunda del segundo stream.

- `merge_streams` que toma dos streams y regresa un stream que alterna elementos de ambos streams.

4. **Árboles infinitos**

   Defina el tipo de los árboles con ramas infinitas de manera perezosa. Tenga en cuenta que dado un árbol infinito t, se puede *destruir* y obtener el nodo raíz junto con una lista infinita con los subárboles infinitos que son *hijos* del nodo.

5. **Árboles mutables** [1]

   La representación de árboles con referencias permite obtener elementos mutables, es decir que pueden ser modificados en memoria sin necesidad de crear nuevas instancias de ellos.

   Implemente un módulo con la siguiente *signatura* para manipular árboles binarios de búsqueda:

```
module type SET = sig
 type t
 val create: unit -> t
 val add: int -> t -> unit
 val mem: int -> t -> bool
 val remove: int -> t -> unit
 val size: t -> int
 val height: t -> int
end
```

   El tipo `t` representará el tipo de árboles binarios de búsqueda con referencias, definido por:

```
type node = E | N of t * int * t
and t = node ref
```

6. **Árboles rojinegros**

   Usando el código en `btree.ml` proporcione dos funciones, una para calcular el tamaño de un árbol binario de búsqueda (`size`) y otra para decidir si un elemento pertenece a un árbol (`member`).

   A continuación modificará la implementación previa para obtener los *eficientes* árboles rojinegros cuya especificación es la siguiente:

   - ningún nodo rojo tiene un nodo *padre* de color rojo, es decir que cualquier nodo rojo tiene como descendientes a nodos de color negro;
   - cada camino desde la raíz hasta una hoja vacía tiene el mísmo número de nodos negros.

   La especificación anterior asegura que cualquier árbol de este estilo estará "razonablemente" balanceado. Por tanto, el tipo de dato para representar los árboles rojinegros es el siguiente:

```
type color =  R | B
type 'a rbtree = Leaf | Branch of color * 'a rbtree * 'a * 'a rbtree
```

[1] Inspirado en el trabajo de Jean-Christophe Filliâtre.

Modifique todas las funciones previamente implementadas, observe que para la inserción de un elemento es necesario definir una función `balance` que verifica la especificación.

7. **Colas doblemente terminadas "*deque*"**

   Una cola es un objeto en donde se pueden almacenar elementos con la restricción que los elementos son incluidos en un extremo y son retirados del otro extremo (*first in–first out*). Las colas doblemente terminadas permiten incluir y eliminar elementos en ambos lados de la cola. Esta estructura está determinada por la siguiente *signatura*

```
module type SET = sig
 type t
 val getHead : 'a t -> 'a
 val getTail : 'a t -> 'a
 val enqueueHead : 'a -> 'a t -> unit
 val enqueueTail : 'a -> 'a t -> unit
 val dequeueHead : 'a t -> 'a
 val dequeueTail : 'a t -> 'a
```

   De una implementación de las colas doblemente terminadas usando arreglos, considere los casos en donde la cola está vacía.

8. **Multiplicación de matrices**

   Dada una secuencia de dimensiones de matrices $[d_0, d_1, \ldots, d_n]$, donde una matriz M_i tiene dimensiones $d_{i-1} \times d_i$, determinar el menor costo para multiplicar las n matrices. El costo para multiplicar dos matrices con dimensiones $i \times j$ y $j \times k$ es $i * j * k$.

   Defina una función recursiva para resolver el problema anterior y después proponga otra función más eficiente utilizando memoización.