Principes de Programmation
TD1

11 février 2019

Exercice 1 (Fold).
Les fonctions `foldl` et `foldr` sont définies sur les listes comme suit :

```
foldr   :: (a -> b -> b) -> b -> [a] -> b
foldr _ x []    = x
foldr f x (y:l) = f y (foldr f x l)

foldl   :: (b -> a -> b) -> b -> [a] -> b
foldl _ x []    = x
foldl f x (y:l) = foldl f (f x y) l
```

1. Que calculent les programmes suivant : [1]

   ```
   sum :: [Int] -> Int
   sum l = foldl (+) 0 l

   calculatrice :: Int
   calculatrice = foldr (\f x -> f x) 16 [(-9),(*9),(-3)]

   calculatriceL :: Int
   calculatriceL = foldl (\x f -> f x) 16 [(-9),(*9),(-3)]
   ```

2. Quelle est la diférence entre `foldl` et `foldr` ?
3. Prouvez que pour toute liste finie on a l'égalité :

   ```
   foldl (+) x l ~= foldr (+) x l
   ```

4. Prouvez que la traduction suivante est correcte.

   ```
   foldr' f x l = foldl (\g z -> g . (f z)) id l x
   ```

5. (Bonus) Prouvez que la traduction suivante de `foldl` à partire de `foldr` est correcte.

   ```
   foldl' f x l = foldr (\x g y -> g (f y x)) id l z
   ```

1. L'opérateur (++) : : [a]->[a]->[a] est la concaténation de listes et l'opérateur ($) :: (a->b)->a->b est l'application.

Exercice 2 (Bind (Bonus)).
L'operateur `=<<` et la fonction `foldlM` sont définis sur les listes comme suit :

```
(=<<) :: (a -> [b]) -> [a] -> [b]
_ =<< []    = []
f =<< (x:l) = (f x) ++ (f =<< l)

foldlM :: (b -> a -> [b]) -> b -> [a] -> [b]
foldlM f x []    = [x]
foldlM f x (y:l) = (\u -> foldM f u l) =<< (f x y)
```

1. Que calculent les programmes suivant :

   ```
   sansCarres :: [Int]
   sansCarres = (\n-> [((exp n 2) +1)..((exp (n+1) 2) -1)]) =<< [1..]

   concat :: [[a]] -> [a]
   concat = id =<<
   ```

2. Ecrivez `(=<<)` à l'aide de `foldl`.
3. Prouvez votre traduction.
4. Que font les programmes suivant :

   ```
   parties :: [a] -> [[a]]
   parties = foldlM (\l x -> [x:l,l]) []

   insertions :: a -> [a] -> [[a]]
   insertions x (y:l) = (x:y:l): (map (y:) (insert x l))
   insertions x []    = [x]

   shufle :: [a] -> [[a]]
   shufles = foldM (\l x -> insertions x l) []
   ```

5. Ecrivez `foldM` à l'aide de `(=<<)` et `foldl`.
6. Prouvez votre traduction.
7. Que font les programmes suivant :

   ```
   evalNDaux :: String -> EtatND -> Bool
   evalNDaux []    (NoeudND b f) = b
   evalNDaux (c:w) (NoeudND _ f) = evalND w (f c)

   evalND :: String -> AutomateND -> Bool
   evalND w a = forall (map (evalNDaux w) a)

   evalAPNDaux :: String -> EtatAPND a -> [a]
   evalAPNDaux []    (NoeudAPND l f) = l
   evalAPNDaux (c:w) (NoeudAPND _ f) = evalAPND w (f c)
   ```

```
evalAPND :: String -> ArbrePrefND a -> [a]
evalAPND w = (evalAPNDaux w) =<<
```

où l'arbre préfix non-déterministe `ArbrePrefND` sera définit dans un prochain TP

```
data EtatAPND a = NoeudAPND [a] (Char -> NoeudAPND a)
type ArbrePrefND = [EtatAPND]
```

8. Définissez `evalND` à l'aide de `=<<`.