Université de Caen Normandie
UFR des Sciences
L2-MATH et L2-INFO

Lundi 9 Mars 2020

Autres Paradigmes
(Durée 1 heure - Notes et supports de CM/TP autorisés)

NOM :

Prénom : **Parcours :** Info Math

Consignes :
— *Pour chaque question, répondre obligatoirement dans l'espace réservé à cet effet.*
— *Rédiger les réponses au brouillon avant de les porter sur la feuille d'examen. On ne fournira pas d'exemplaire supplémentaire du sujet.*

---

1. Un ensemble est représenté par la liste de ses éléments dans laquelle aucun élément ne figure plus d'une fois. L'ensemble vide est représenté par la liste vide.

   Définir les fonctions `(intersect xs ys)` et `(union xs yx)` qui calculent l'intersection et l'union de 2 ensembles représentés respectivement par les listes `xs` et `ys`.
   (On pourra utiliser la fonction prédéfinie `(elem x xs)` *qui détermine si un élément* `x` *appartient ou non à une liste* `xs`*)*

```
> intersect [1, 5, 6] [7, 6, 9, 1]   ==> [1, 6]
> intersect [1, 5, 6] [7, 2, 9]   ==> []
> union [1, 5, 6] [7, 6, 9, 1]   ==> [5, 7, 6, 9, 1]
> union [1, 5, 6] [7, 2, 9]   ==> [1, 5, 6, 7, 2, 9]

> intersect "abcd" "gecfa"   ==> "ac"
> union "abcd" "gecfa"   ==> "bdgecfa"
```

   Compléter : **(4 pts)**

```
intersect, union :: (Eq a) =>

intersect []

intersect
     |

     |


union []

union
     |

     |
```

2. Définir la fonction `(suppriPremierVrai p xs)` qui, pour un prédicat `p` et une liste `xs`, supprime le premier élément de `xs` qui satisfait `p`. Si aucun élément de `xs` ne satisfait `p`, la fonction retourne `xs`.

```
> suppriPremierVrai (< 0) [1, 2, -3, 4, -5]   ==> [1, 2, 4, -5]
> suppriPremierVrai (< 0) [1..3]  ==> [1, 2, 3]
> suppriPremierVrai even [4..8]   ==> [5, 6, 7, 8]
> suppriPremierVrai odd [4..8]   ==> [4, 6, 7, 8]
> suppriPremierVrai (== 'a') "abac"   ==> "bac"
> suppriPremierVrai (/= 'a') "abac"  ==> "aac"
```

Compléter : **(1,5 pts)**

```
suppriPremierVrai ::

suppriPremierVrai _ [] =

suppriPremierVrai p (x:xs)

   |                    =

   | otherwise          =
```

3. Définir la fonction `(numerote xs)` qui numérote les éléments d'une liste `xs` à partir du rang 0.

```
> numerote "abc"   ==> [('a',0), ('b',1), ('c',2)]
> numerote [7, 5]   ==> [(7,0), (5,1)]
> numerote [0..4]   ==> [(0,0),(1,1),(2,2),(3,3),(4,4)]
> numerote [] ==> []
```

Compléter : **(3 pts)**

```
numerote ::

numerote []  =

numerote (x:xs) = inter (x:xs) 0

             where inter             =

                   inter             =
```

4. Comme dans le TP #2, on associe, à chaque entier naturel, sa décomposition en facteurs premiers représentée par les types suivants :

```
type Facteur = Int
type Exposant = Int
type Couple = (Facteur, Exposant)
type Decomposition = [Couple]
```

Définir la fonction `(produit xs ys)` qui calcule le produit de 2 entiers naturels représentés par leur décomposition

```
> produit [(2,3), (5,2)] [(7,3)]   ==> [(2,3), (5,2), (7,3)]
> produit [(2,3), (5,2)] [(2,2)]   ==> [(2,5), (5,2)]
> produit [(2,3), (5,2)] [(2,1), (5,1), (7,3)]   ==> [(2,4), (5,3), (7,3)]
```

Compléter : **(2,5 pts)**

```
produit ::

produit

produit

produit ((k1,d1):p1) ((k2,d2):p2)

      |

      |

      | otherwise =
```

5. (a) *En utilisant une liste en compréhension*, définir la fonction `(tableMult n)` qui, à un entier `n`, associe sa table de multiplication représentée par la liste des neuf premiers triplets :

```
> tableMult 2
    [(2,1,2),(2,2,4),(2,3,6),(2,4,8),(2,5,10),(2,6,12),(2,7,14),(2,8,16),(2,9,18)]
> tableMult 3
    [(3,1,3),(3,2,6),(3,3,9),(3,4,12),(3,5,15),(3,6,18),(3,7,21),(3,8,24),(3,9,27)]
```

Compléter : **(1 pt)**

```
tableMult ::

tableMult n = [                              x <- [1..9]]
```

(b) *De manière analogue au (a)*, définir la fonction `(tableOp op n)` qui retourne la table de l'entier `n` pour l'opérateur `op` :

```
> tableOp (+) 3
    [(3,1,4),(3,2,5),(3,3,6),(3,4,7),(3,5,8),(3,6,9),(3,7,10),(3,8,11),(3,9,12)]
> tableOp (*) 2
    [(2,1,2),(2,2,4),(2,3,6),(2,4,8),(2,5,10),(2,6,12),(2,7,14),(2,8,16),(2,9,18)]
```

Compléter : **(2 pts)**

```
tableOp :: (    -> Int ->     ) ->           ->

tableOp op n = [                                         ]
```

6. Comme dans le TP #3, on définit le type suivant pour représenter les arbres binaires :

```
data Btree a = Nil
             | Bin a (Btree a) (Btree a)
     deriving (Show, Ord, Eq)
```

On suppose définies les fonctions

- `(minArbre t)` qui détermine le plus petit élément d'un `(Btree a)`
- `(maxArbre t)` qui détermine le plus grand élément d'un `(Btree a)`

Définir la fonction `(estABR t)` qui détermine si un arbre binaire (non-vide) `t` est un arbre binaire de recherche.

```
a1 = (Bin 4 (Bin 3 (Bin 2 Nil Nil) Nil) (Bin 7 (Bin 6 Nil Nil) (Bin 8 Nil Nil)))
a2 = (Bin 4 (Bin 3 (Bin 1 Nil Nil) Nil) (Bin 8 (Bin 7 Nil Nil) (Bin 2 (Bin 9 Nil Nil) Nil)))
```

```
> estABR a1  ==> True
> estABR a2   ==> False
```

Compléter : **(3 pts)**

```
estABR :: (Ord a) =>

estABR (Bin x Nil Nil) =

estABR (Bin x t1 Nil)  =  (estABR t1)  &&

estABR (Bin x Nil t2)  =               &&

estABR (Bin x t1 t2)   =
```

7. On considère la représentation des arbres binaires non étiquetés étudiée en CM :

```
data Tree a = Tip a
            | Bin (Tree a) (Tree a)
      deriving Show
```

(a) Définir la fonction `(filterTree p t)` qui, pour un prédicat `p` et un arbre binaire non étiqueté `t`, retourne la liste des feuilles de `t` qui satisfont `p`.

```
> filterTree even (Bin (Bin (Tip 4) (Tip 5)) (Tip 2))  ==> [4,2]
> filterTree odd (Bin (Bin (Tip 4) (Tip 5)) (Tip 2))  ==> [5]
> filterTree (/=[]) (Bin (Bin (Tip [2]) (Tip [])) (Tip [1,2]))  ==> [[2], [1, 2]]
```

Compléter : **(2 pts)**

```
filterTree ::

filterTree p (Tip x)
     |

     | otherwise =

filterTree
```

(b) On considère la fonction `(unknown f t)` définie par : **(1 + Bonus 1 pt)**

```
unknown f (Tip x)     = Tip (f x)
unknown f (Bin t1 t2) = Bin (unknown f t1) (unknown f t2)
```

i. Compléter le type de cette fonction :

```
unknown ::
```

ii. Evaluer les expressions suivantes :

```
> unknown (+7)  (Bin (Bin (Tip 4) (Tip 5)) (Tip 2))
==>

> unknown even (Bin (Bin (Tip 4) (Tip 5)) (Tip 2))
 ==>
```

iii. De manière générale, que calcule cette fonction ?

```
-- 1.
intersect, union :: (Eq a) => [a] -> [a] -> [a]
intersect [] xs = []
intersect (x:xs) ys
    | elem x ys = x : (intersect xs ys)
    | otherwise = intersect xs ys
union [] xs = xs
union (x:xs) ys
    | elem x ys = union xs ys
    | otherwise = x : (union xs ys)

-- 2.
suppriPremierVrai :: (a -> Bool) -> [a] -> [a]
suppriPremierVrai _ [] = []
suppriPremierVrai p (x:xs)
    | p x       = xs
    | otherwise = x : (suppriPremierVrai p xs)

-- 3.
numerote :: [a] -> [(a, Int)]
numerote [] = []
numerote (x:xs) = inter (x:xs) 0
   where inter  []  _   = []
         inter (x:xs) n = (x, n) : (inter xs (n+1))

-- 4. (TP 02, exo 3 et exo 4)
produit :: Decomposition -> Decomposition -> Decomposition
produit [] ys = ys
produit xs [] = xs
produit ((k1,d1):p1) ((k2,d2):p2)
   | k1 == k2  = (k1,d1+d2) : (produit p1 p2)
   | k1 < k2   = (k1,d1): (produit p1 ((k2,d2):p2))
   | otherwise = (k2,d2): (produit ((k1,d1):p1) p2)

-- 5.
tableMult :: Int -> [(Int, Int, Int)]
tableMult n = [(n, x, n * x)| x <- [1..9]]
tableOp :: (a -> Int -> b) -> a -> [(a, Int, b)]
tableOp op n = [(n, x, op n x)| x <- [1..9]]

-- 6. (TP 03, exo 5 b)
estABR :: (Ord a) => Btree a -> Bool
estABR (Bin x Nil Nil) = True
estABR (Bin x t1 Nil)  = (estABR t1) && (x > maxArbre t1)
estABR (Bin x Nil t2)  = (estABR t2) && (x < minArbre t2)
estABR (Bin x t1 t2)   = (estABR t1) && (x > maxArbre t1) && (estABR t2) && (x < minArbre t2)

-- 7.
filterTree :: (a -> Bool) -> Tree a -> [a]
filterTree p (Tip x)
   | p x       = [x]
   | otherwise = []
filterTree p (Bin t1 t2) = (filterTree p t1) ++ (filterTree p t2)
-- 7.b. (CM 05, slide 14)
unknown :: (a -> b) -> Tree a -> Tree b
> unknown (+7)  (Bin (Bin (Tip 4) (Tip 5)) (Tip 2))  ==> Bin (Bin (Tip 11) (Tip 12)) (Tip 9)
> unknown even (Bin (Bin (Tip 4) (Tip 5)) (Tip 2))  ==> Bin (Bin (Tip True) (Tip False)) (Tip True)
unknown transforme un arbre binaire en un autre arbre binaire ayant exactement la meme structure,
mais ou chaque feuille etiquetee x est transformee en une feuille etiquetee (f x)
```