Agregats

```
cumule :: Num a => (a -> a -> a) -> a -> [a] -> a
cumule f i [] = i
cumule f i (x:xs) = f x (cumule f i xs)

somme2 :: [Int] -> Int
somme2 = cumule (+) 0

somme :: Num a => [a] -> a
somme = cumule (+) 0
```

Agregeable

```
module Agregeable where

class Agregeable a where
 neutre :: a
 operation :: a -> a -> a

instance Agregeable Int where
 neutre = 0
 operation = (+)
--il faut forcer le type Int dans la definition de la liste, sous peine d'avoir une erreur "ambigous type variable"

instance Agregeable Bool where
 neutre = False
 operation = (||)

instance Agregeable [a] where
 neutre = []
 operation = (++)
--l'operation sur les listes pour faire de [a] une instance de Agregeable est la concatenation

cumule :: Agregeable a => [a] -> a
cumule [] = neutre
cumule [x] = operation neutre x
cumule (x:xs) = operation x (cumule xs)
--Si on passe une liste vide d'entiers, cumule renvoie 0 (il faut forcer le type avec ::[Int] sous peine de type ambigu)
--Si on passe une liste de listes vides, cumule renvoie une liste vide (sans besoin de forcer le type)
--Si on passe une liste non-vide, on realise l'operation avec l'element neutre et le premier element, puis ce resultat avec le second element, etc
```

IE

```
module IE where

import Agregeable

data Int_ex = Moins_inf | I Int deriving (Show)
```

```
max_ie :: Int_ex -> Int_ex -> Int_ex
max_ie Moins_inf x = x
max_ie x Moins_inf = x
max_ie (I x) (I y) | x >= y = I x
                   | otherwise = I y
--si on donne a max_ie Moins_inf et n'importe quel nombre, il renvoie ce nombre. Sinon, il compare les deux nombres et renvoie le plus grand.


instance Agregeable Int_ex where
 neutre = Moins_inf
 operation = (max_ie)

--L'élément neutre est Moins_inf car tout calcul de maximum entre Moins_inf et n'importe quel autre nombre renvoie le nombre sus-mentionne

--map I transforme des Int en Int_ex (Int extended)

max_liste :: [Int] -> Int_ex
max_liste [] = Moins_inf
max_liste l = cumule (map I l)

--Sur une liste vide, max_liste renvoie l'element neutre
```

Set

```
module Set where

data Set a = Set [a] deriving Show

set :: [a] -> Set a
set l = Set l

set_difference :: (Eq a) => Set a -> Set a -> Set a
set_difference s1 (Set []) = s1
set_difference s1 (Set (x:xs)) = remove x (set_difference s1 (Set xs))

--On ne verifie pas le presence de doublons, ni dans la construction de l'ensemble (avec set), ni dans set_difference
--on pourrait s'occuper des doublons des la construction de l'ensemble, et les verifier dans set_difference

remove :: (Eq a) => a -> Set a -> Set a
remove x (Set []) = Set []
remove x (Set (u:us)) | x == u = Set us
                      | otherwise = Set (u:r)
                        where Set r = remove x (Set us)

--remove retire d'un ensemble le premier element egal a un autre element passe en parametre

grab :: Set a -> a
grab (Set (x:xs)) = x
```

```
remove_duplicates  ::  (Eq a) =>  Set a -> Set a
remove_duplicates (Set l) = Set (remove_duplicates_aux l)
  where
    remove_duplicates_aux e@[] = e
    remove_duplicates_aux (a:reste) | elem a reste = remove_duplicates_aux reste
                                    | otherwise = a:(remove_duplicates_aux reste)
```

Hanoi

```
module Hanoi where

import Set

data Pos = A | B | C  deriving (Show, Eq)
--un element de type Pos indique la position d'un disque, sur un des trois piquets

data Movement = Mvt Int Pos Pos deriving Show
--un element de type Movement est un deplacement d'un piquet a un autre, d'un disque identifie par l'element
Int

hanoi :: Int -> [Movement]
hanoi k = hanoi_aux k A B

other :: Pos -> Pos -> Pos
other p1 p2 = grab (set_difference (set [A,B,C]) (set [p1,p2]))


hanoi_aux :: Int -> Pos -> Pos -> [Movement]
hanoi_aux 1 p1 p2 = [Mvt 1 p1 p2]
hanoi_aux k p1 p2 = (hanoi_aux (k-1) p1 (other p1 p2)) ++ (hanoi_aux 1 p1 p2) ++ (hanoi_aux (k-1)
(other p1 p2) p2)
--On deplace l'element au dessus sur un autre piquet, puis les autres disques sur le piquet restant
--Pour realiser cette deuxieme operation, on repete l'operation ci-dessus avec un disque de moins
```