Mots

```
#!/usr/bin/env ghci

espace :: Char -> Bool
espace ' ' = True
espace _ = False

separe :: (Char -> Bool) -> [Char] -> ([Char], [Char])
separe f xs = separe_aux f xs ([],[])
 where separe_aux f [] (a,b) = (a,b)
       separe_aux f (x:xs) (a,b) | (f x) = separe_aux f xs (a++[x],b)
                                 | otherwise = (a,x:xs)

grignote_espace :: String -> String
grignote_espace xs = snd (separe espace xs)

un_mot :: String -> (String, String)
un mot [] = ([],[])
un_mot xs = (a, grignote_espace b)
 where (a,b) = separe (not.espace) (grignote_espace xs)

mots :: String -> [String]
mots [] = []
mots xs = a:(mots b)
 where (a,b) = un_mot xs
```

Listes

```
#!/usr/bin/env ghci
data Liste = Vide | Cons Int Liste deriving Show

vide :: Liste -> Bool
vide Vide = True
vide _ = False

premier :: Liste -> Int
premier (Cons a _) = a

reste :: Liste -> Liste
reste (Cons _ liste) = liste

longueur :: Liste -> Int
longueur Vide = 0
longueur xs = (+) 1 (longueur (reste xs))

dernier :: Liste -> Int
dernier (Cons a Vide) = a
dernier (Cons a liste) = dernier liste

applique :: (Int -> Int) -> Liste -> Liste
applique f Vide = Vide
applique f (Cons a liste) = Cons (f a) (applique f liste)

ajoute :: Int -> Liste -> Liste
ajoute b Vide = Cons b Vide
ajoute b (Cons a liste) = Cons a (ajoute b liste)
```

```
renverse :: Liste -> Liste
renverse Vide = Vide
renverse (Cons a liste) = ajoute a (renverse liste)
```

Trifusion

```
#!/usr/bin/env ghci

halve :: [ Int ] -> ([ Int ] , [ Int ])
halve [] = ([],[])
halve [x] = ([x],[])
halve (x:y:zs)=((x:xs) , (y:ys))
  where (xs,ys)=halve(zs)

combine :: [ Int ] -> [ Int ] -> [ Int ]
combine [] [] = []
combine x [] = x
combine [] y = y
combine (x:xs) (y:ys) | x<=y = x:(combine xs (y:ys))
                      | otherwise = y: (combine (x:xs) ys)

tri_fusion :: [ Int ] -> [ Int ]
tri_fusion [] = []
tri_fusion [x] = [x]
tri_fusion xs = combine (tri_fusion a) (tri_fusion b)
 where (a,b) = halve xs
```

Cumul

```
#!/usr/bin/env ghci

somme :: [Int] -> Int
somme [] = 0
somme (x:xs) = x + somme xs

produit :: [Int] -> Int
produit [] = 1
produit (x:xs) = (*) x (produit xs)

cumule :: (Int -> Int -> Int) -> Int -> [Int] -> Int
cumule f i [] = i
cumule f i (x:xs) = f x (cumule f i xs)

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

produit2 :: [Int] -> Int
produit2 liste = cumule (*) 1 liste

maxi :: [Int] -> Int
maxi (x:xs) = maxi_aux x xs
 where maxi_aux a [] = a
       maxi_aux a (x:xs) | a<=x = maxi_aux x xs
                         | otherwise = maxi_aux a xs
```