Programmation fonctionelle Haskell

I.Constructions de base

Evaluation conditionnel

```
f :: Integer -> Integer
f n =  if n == 0 then 1
       else n * f (n-1)
main :: IO ()
main = print (f 5)
```

Output `120`

Filtrage par motif (Pattern matching)

```
f 0 = 1
f n = n * f (n-1)
```

```
f n = case n of
    0 -> 1
    _ -> n * f (n-1)
main = print(f 5)
```

Les noms de fonctions et variables commencent par une minuscule

```
square x = x*x
inc x = x + 1

f x = (inc.square) x  -- ou juste f x = inc (square x)
main = print( f 3 )
```

Output `10`

Evaluation gradée

```
signe n
    | n < 0     = -1
    | n > 0     = 1
    | otherwise = 0
main = print(signe 10)
```

Output `1`

Fonction récursive terminale

```
f' 0 r = r
f' n r = f' (n-1) (n*r)
f n = f' n 1
main = print (f 3)
```

Output `6`

Ex slide 15

```
puiss  x 0 = 1
puiss  x n = x * ( puiss  x (n-1))
main = print (puiss  4 4 )
```

Output `256`

```
p x n = case n of
   0 -> 1
   _ -> if (even n)
       then p ( x*x ) (div n 2)
       else x* p (x*x) (div n 2)
main = print (p 4 3)
```

Output `64`

```
fibo 0 = 1
fibo 1 = 1
fibo n = fibo(n-1) + fibo(n-2)
main = print (fibo 4)
```

Output `5`

Ex slide 16

La hauteur palindromique(123->321)

```
r ::Integer->Integer    -- fonction donnée
raux n x = if n == 0
    then x
    else raux (div n 10) (10*x + (mod n 10))
r n = raux n 0
main = print (r 123)
```

Output `123`

```
r ::Integer->Integer    -- fonction donnée
raux n x = if n == 0
    then x
    else raux (div n 10) (10*x + (mod n 10))
r n = raux n 0

hpal n = if (r n) == n  -- calcule de fois
    then 0
    else 1 + hpal(n + r n)
main = print (hpal 454)
```

Output `0`

Délcaration locale

```
sommeCube x y = let  z = x + y in z*z*z
main = print(sommeCube 2 1)
```

```
sommeCube x y = let  add u v = u +v
                     z = x +  y
                in z*z*z
main = print(sommeCube 2 1)
```

Output `27`

Plus de pattern matching

```haskell
data PossibleReel = Rien | Valeur Double deriving Show
-- le nouveau type PossibleReel
-- constructeur de donnée Rien et Valeur
-- le constructeur Rien qui permet de construire un PossibleReel sans donner d'argument
-- le constructeur Valeur qui permet de construire un PossibleReel à partir d'un double

-- fonction: inversion
inversion :: Double -> PossibleReel
inversion 0 = Rien
inversion x = Valeur (1 / x)

-- fonction: opposePR
opposePR :: PossibleReel -> PossibleReel
opposePR Rien       = Rien
opposePR (Valeur x) = Valeur (-x)
```

```haskell
main = print (inversion 2.9)
```

Output `Valeur 0.3448275862068966`

```haskell
main = print (opposePR (Valeur 2.2))
```

output `Valeur (-2.2)`

Ex slide 22

Ecrire une fonction et qui réalise le et logique entre deux Bool

```haskell
et :: Bool -> Bool -> Bool
et True True = True
et _ _ = False
main = print(et True True)
```

Output `True`

II.Type

Ex : Type algébrique slide 23

Nom de type commerce par MAJUSCULE

```
data Point = Coord Double Double
distance :: Point -> Point -> Double
distance (Coord x1 y1 ) (Coord x2 y2 ) = sqrt( (x1-x2)^2 + (y1-y2)^2 )
main = let p1 = Coord 1.1 1.1 in print (distance p1 (Coord 3.1 3.1))
```

Output `2.8284271247461903`

```
data Point = Coord Double Double
distance :: Point -> Point -> Double
distance (Coord x1 y1 ) (Coord x2 y2 ) = sqrt( (x1-x2)^2 + (y1-y2)^2 )
--constructeur de donnée
data Figure =  FigP Point | FigC  Double| FigCa Point Point
-- type de sortie de donnée est Double
perimetre(FigP _ ) = 0.0
permetre(FigC r) = 2*3.14*r
permetre(FigCa p1 p2) = (distance p1 p2 )* 2 * sqrt 2
main = print (perimetre(FigP (Coord 2.1 2.1)))
```

Output `0.0`

```
main = print (permetre(FigC 2.1))
```

Output `13.188`

```
main = print (permetre (FigCa (Coord 2.1 2.1)  (Coord 3.1 3.1)))
```

Output `4.000000000000001`

erreur 26 test.hs ???

Les tuples

fsd et snd pour les couples
fst est la fonction qui pour un couple donne la première composante, et snd donne la 2e

composante : par exemple fst (1,3) vaut 1 et snd (1,3) vaut 3.

```
maxi ::(Integer, Integer)-> Integer
maxi x = let a = fst x
             b = snd x in
             if a > b then a else b
main =print (maxi (20,3))
```

pattern matching

```
maxi ::(Integer, Integer)-> Integer
maxi (a,b) = if a > b then a else b
main =print (maxi (20,3))
```

Output `20`

Pas que pour les couples, aussi pour plusieurs paramètres

```
ror :: (Char, Char,Char) ->(Char, Char,Char)
ror(a,b,c) = (c,b,a)
```

Type: les enregistrements

```
data Canard = Coin {nom::String, enverg:: Double}

info::Canard-> String
info c= "Oh le beau canard " ++ (nom c) ++
        " d'envergure " ++ show (enverg c) ++ "    "

main = let c1 = Coin "Coin" 0.5
           c2 = Coin {nom = "Gaga" , enverg = 0.2}
           c3 = c2 {enverg= 0.3}
        in
            print(info c1 ++ info c3)
```

Output

```
Oh le beau canard Coin d'envergure 0.5 Oh le beau canard Gaga d'envergure 0.3
```

Types algébriques récursifs

??? commet tester

```
data Nat = Zero | Succ Nat
intVal :: Nat -> Integer
intVal Zero = 0
intVal (Succ x) = (intVal x) + 1

addition :: Nat -> Nat -> Nat
additoin (Succ x) y = addition x (Succ y)  -- ou Succ(additon x y )
addition (Succ x) Zero = Succ x
addition (Succ x) (Succ y) = addition x (Succ (Succ y))
main = print (intVal(addition (Succ Zero) (Succ Zero) ))
```

Ex liste er arbres slide 29

```
data Liste = Vide | Cons Integer Liste
somme :: Liste -> Integer
somme Vide = 0
somme (Head x l) = x + somme l --permettre de parcourir tous les élément dans la liste

data ArbreBinaire = VideA |  Head Integer ArbreBinaire ArbreBinaire
hauteur VideA = 0
hauteur (Noeud x g d)= 1 + max (hauteur g) (hauteur d) --gauche et droit
```

Types en Haskell : les listes

```
-- les nombres paris inférieurs à n
evens n = [ x+1 | x <- [1..(n-2)], odd x]
```

Ex Construire la liste des triplets slide 31

comment présenter la sortie ?

```
triple = [(a,b,c) | a<-[0..333], b<-[(a+1)..500], c<-[(b+1)..1000], a+b+c == 1000 , a^2+b^2 == c^2
main = print triple
```

Ex ecrire une fonction reverse qui inverse une liste ? slide 32

J listfct.hs ?

```
reverseAux [] acc = acc
reverseAux (x:xs) acc = reverseAux xs (x:acc)

rev x = reverseAux x []

main = print (rev [1..10])
```

ex 1 delete

x is the first element (head) and xs is the rest of the list (tail).

```
deleteL [] e = []
deleteL (x:xs) e = if x==e then xs else x:(deleteL xs e)
main = print(deleteL [1..10] 11)
```

ex 2 maximum

```
maxL_aux [] x = x
maxL_aux (x:xs) m = if x > m then maxL_aux xs x else maxL_aux xs m

maxL:: [Integer] -> Integer -- obligé de spécifier le type pour que maxL [] marche
maxL [] = error "maxL: La liste est vide"
maxL (x : xs) = maxL_aux xs x

main = print(maxL [1,5,3,2])
```

ex 3 ?

Une focntion trimax qui réalise le tri par extraction du maximum dans une liste d'éntier

```
deleteL [] e = []
deleteL (x:xs) e = if x==e then xs else x:(deleteL xs e)

maxL_aux [] x = x
maxL_aux (x:xs) m = if x > m then maxL_aux xs x else maxL_aux xs m

maxL::[Integer]->Integer-- obligé de spécifier le type pour que maxL [] marche
maxL [] = error "maxL: La liste est vide"
maxL (x:xs) = maxL_aux xs x

? trimax_aux [] acc = acc
? trimax_aux x acc = let m = maxL x in trimax_aux (deleteL x m) (m:acc)
trimax x = trimax_aux x []

main = print(trimax [1,5,3,2,7,8,6])
```

Output `[1,2,3,5,6,7,8]`

Des focntions génériques sur des types algébriques génériques

```
-- pour tout tyoe a
head :: [a] -> a
head [] = error ("head: Empty list")
head (x:xs) = x
-- dans Data.Maybe
listToMaybe::[a]->Maybe a
lsitToMaybe [] = Nothing
lsitToMaybe (x:xs) = Just x
```

III.Fonctions d'ordre supérieur

Currying

-Currying: all functions in Haskell really take only one argument. (one by one)
-The process of creating intermediate functions when feeding arguments into a complex function is called currying

? slide 37

```
curry::((a,b)->c)->(a->(b->c))
uncurry::(a->(b->c))->((a,b)->c)
```

Ex slide 37

ex 1 Quel le type de map ?

```
-- applique f à tous les éléments
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

```
map::(a->b)->[a]->[b]
```

ex 2 une fonction flip, avec son type

```
myflip::(a->b->c)->(b->a->c)

myflip' f x y = f y x
myflip f = myflip' f  -- inverse

main = print(myflip (-) 1 3) -- output = 2
```

Application de fonction $

IV.

Evaluation non-stricte et foldS 51

foldr --> liste infinie

V.Entrée, sortie

Le type paramétré `IO`

ex slide 57

```
ioLength :: IO Int
ioLength = do
    x <- getLine
    return $ length x
main = do
    x<- ioLength
    print x
```

```
ioLength = getLine >>= return.length
main = ioLength >>= print
```

VI.Généricité avancée