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 `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

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)))
```

perimetre:: Figure -> Double

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 $

slide 38 fonctions anonymes

```
main = print $ map (\x -> x+1) [100]  -- output [101]

add:: Integer -> Integer -> Integer
add  = (\x y -> x+y)
main = print $ add 10 11  -- output 21
```

Récursion sur les listes : map

41 Ex en utilisant map, écrire une fonction an qui donne la liste de toutes les anagrammes d'un mot

```
import Data.List -- pour delete
an::String->[String]
an "" = [""]
an xs = concat $ map (\c -> map (c:) (an (delete c xs))) xs
-- c est une lettre

main = print(an "abc")
-- output ["abc","acb","bac","bca","cab","cba"]
```

concat

delete

42 Récursion sur les listes : filter

```
divise n k = mod n k == 0 -- True if k|n
diviseurs n = filter (divise n) [1..n]
diviseurs n = filter ((==0).(mod n)) [1..n] -- Le point est la composition mathématique

premiers n = filter (\x -> length (diviseurs x) == 2) [2..n]

main = print(premiers 15)
```

43 Récursion sur les listes : folds

46 EX

```
sumX xs = foldl (+) 0 xs
main = print $ sumX [1,23]

maxL::Ord a => [a] -> a
maxL xs = foldl1 max xs
main = print(maxL [1,5,3])  -- output 5

-- indique si tous les éléments sont vrais
andL = foldr (&&) True
main = print(andL [True, False, True])

-- qui indique si au moins un élément de la liste satisfait un prédicat donné
f x = x == 0
anyL f xs =  or $ map f xs
-- or xs = foldl (||) False xs
main = print $ anyL f [1,1,1]

-- concat qui concatène une liste de listes
concatL  = foldl (++) []
main = print $ concatL [[1,2], [3,4,5], [6]]
-- ou main = print $ concat [[1,2], [3,4,5], [6]]
```

IV. Méthodes d'évaluation

49

-- le 1234567e nombre de Fibonacci

Evaluation non-stricte et foldS 51

foldr --> liste infinie

V.Entrée, sortie

Le type paramétré `IO` monade

ex 51

```
-- entrer coin affiche coin nioc
main::IO ()
main = getLine >>= (\x->putStrLn(x ++ " " ++ reverse x))
-- E macc S macc ccam

main = getLine >>= (\x -> getLine >>= (\y -> putStrLn(y++ " "++ x)))
```

ex slide 57

une fonction ioLength qui donne la longueur d'une chaîne de caractères lue au clavier

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

Autre solution

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

ex 58 Le chiffre de César

```
import Data.Char
-- ord :: Char -> Int
-- chr :: Int -> Char
cesar n mot = map ( \x -> chr $ (ord 'a' + ( mod (ord  x - ord 'a' + n )  26 ) ))  mot
main = print (cesar 13 "coin") --output "pbva"

-- la fonction main permettant l'acquisition de l'entier et de la chaîne de caractères
main = do
    x <- getLine
    let n = (read x) :: Int  -- pour être sûr que le type de conversation est bien
    mot <- getLine
    putStrLn $ cesar n mot

-- la fonction main pour forcer l'utilisateur à entrer un entier compris entre 1 et 25
getIntBorne :: IO Int
getIntBorne = do
    putStrLn "Entrez une valeur "
    x <- getLine
    let n =(read x) ::Int
    if n > 0 && n <= 25 then return n
                        else getIntBorne
main = do
    x <- getLine
    mot <- getLine
    putStrLn $ cesar n mot
```

```
ess = [print "Coin", print "Meuh", print "Miaou"]
main = do
        ess!!2          -- output  Miaou
        sequence_ ess   -- output tout
```

VI.Généricité avancée

Typeclass

```
-- pour tout type numérique a
product::Num a=>[a]->a
product = foldl (*) 1
```

ex 64

数列 <-

concatMap f == concat.(map f)

Il y aura une questions sur les monades et les foncteurs

Maybe, muni de mfmap, est appelé un foncteur

```haskell
-- incrément 增量
-- décrément 递减
-- inversion 倒数
import Data.Maybe
data Expr a = Val a |  Inc (Expr a) | Dec (Expr a) | Inv (Expr a) | Neg (Expr a)

evaluate::(Fractional a)  => Expr  a -> a
evaluate (Val a) = a
evaluate (Inc e) = evaluate e + 1
evaluate (Dec e) = evaluate e - 1
evaluate (Inv e ) = 1 / (evaluate e)
evaluate (Neg e) = - (evaluate e)
-- main = print $ evaluate $ Inc $  Val  4
-- main = print $ evaluate $ Inv 0

isZero::(Eq a,Num a)=> (Maybe a) -> Bool
isZero Nothing = False      -- Nothing  et Just viennnet de Data.Maybe
isZero (Just x) = (x == 0)

mevaluate::(Eq a, Fractional a) => Expr a -> Maybe a
mevaluate (Val a) = Just a
mevaluate (Inc e) = let res = mevaluate e in
                    if isNothing res then Nothing
                    else Just (fromJust res + 1)
mevaluate (Dec e) = let res = mevaluate e in
                    if isNothing res then Nothing
                    else Just (fromJust res - 1)
mevaluate (Inv e) = let res = mevaluate e in
                    if (isNothing res) || (isZero res) then Nothing
                    else Just (1 / (fromJust  res))
mevaluate (Neg e) = let res = mevaluate e in
                    if isNothing res then Nothing
                    else Just (negate (fromJust res))

-- main = print $ mevaluate $ Inv(Val(0))

-- ordre supérieur mfmap évolution
mfmap::(a->b) -> Maybe a -> Maybe b
mfmap _ Nothing = Nothing
mfmap op (Just x) = Just (op x)

mevaluate:: (Eq a, Fractional a) => Expr a -> Maybe a
mevaluate (Val a) = Just a
mevaluate (Inc e) = mfmap (+1) (mevaluate e)
mevaluate (Dec e) = mfmap (subtract 1) (mevaluate e)
mevaluate (Inv e )= let res = mevaluate e in
```

```
                    if isZero res then Nothing
                    else mfmap (\x -> 1 / x) res
main = print $ mevaluate $ Inv(Val(0))
```

ex 65

fmap doit vérifier les propriétés suivantes :

1. fmap id == id
2. fmap (f.g) == (fmap f).(fmap g)

```
-- la fonction fmap pour le constructeur de type []
myfmap:: (a->b) -> [a] -> [b]
myfmap f [] = []
myfmap f (x:xs) = (f x) : (fmap f xs)

--  la fonction fmap pour le constructeur de type r->
myfmap :: (a->b) -> (r->a) -> (r->b)
myfmap f g = f . g
```

```
import Data.Maybe
data Expr a = Val a | Add (Expr a) (Expr a) | Sub (Expr a) (Expr a) | Mul (Expr a) (Expr a) | Div

mliftA2::(a->b->c)->Maybe a->Maybe b->Maybe c
mliftA2 _ Nothing _ = Nothing
mliftA2 _ _ Nothing = Nothing
mliftA2 f (Just a) (Just b) = Just $ f a b

isZero::(Eq a,Num a) => (Maybe a) -> Bool
isZero Nothing = False
isZero (Just x) = (x == 0)

mevaluate::(Eq a,Fractional a) => Expr a -> Maybe a
mevaluate (Val x) = Just x
mevaluate (Add e1 e2) = mliftA2 (+) (mevaluate e1) (mevaluate e2)
mevaluate (Sub e1 e2) = mliftA2 (-) (mevaluate e1) (mevaluate e2)
mevaluate (Mul e1 e2) = mliftA2 (*) (mevaluate e1) (mevaluate e2)
mevaluate (Div e1 e2) = let res2 = mevaluate e2 in
                        if isZero res2 then Nothing
                        else mliftA2 (/) (mevaluate e1) res2
```

ex 68 Foncteur applicatif (applicative functor)

```
-- définir apm
-- apm :: Maybe (a -> b) -> Maybe a -> Maybe b
apm::Maybe(a->b)->Maybe a->Maybe b
apm Nothing _ = Nothing
apm _ Nothing = Nothing
apm (Just f) (Just a) = Just $ f a

-- remplacer mliftA2 par mfmap et apm
import Data.Maybe
data Expr a = Val a | Add (Expr a) (Expr a) | Sub (Expr a) (Expr a) | Mul (Expr a) (Expr a) | Div

mfmap::(a->b) -> Maybe a -> Maybe b
mfmap _ Nothing = Nothing
mfmap op (Just x) = Just $ op x

mevaluate (Add e1 e2) = apm $ mfmap ((+) (mevaluate e1)) (mevaluate e2)

-- un opérateur d'addition à 3 éléments
add3 x y z = x + y + z
mevaluate (Add3 e1 e2 e3) = mfmap add3 e1 `apm` e2 `apm` e3
```

```
mfmap (+) (Maybe 1) ‘apm‘ Nothing
```

ex 70

```
-- Définir fmap en fonction de pure et <*>
fmap::(a->b) -> m a -> m b
pure::a -> m a
ap::m (a->b) -> m a -> m b
fmap f x = ap (pure f) x -- pure f <*> x


pure::a->(r->a)
pure x = (\_ -> x)

(<*>)::(r->(a->b))->(r->a)->(r->b)
-- exemple : f <*> u = (\x -> f x (u x))

--Donner une expression sans variable ni λ (point-free) de la fonction f x = (cos x) * x
f = (*) <*> cos
g = pure (*) <*> cos <*> sin -- redécomposer les fonctions pour comprendre
```

ex 75

Définir fmap à partir de >>= et return

```
return::a-> ma
(>>=)::m a -> (a -> mb) -> m b

fmap::(a->b)->(m a)->(m b)
fmap f x = x >>= return.f
```