Principes de Programmation
TD1

16 janvier 2019

Exercice 1 (Paresse et Listes).

1. Que fait l'opérateur `(!!)` défini dans haskell par :

```
(!!) :: [a] -> Int -> a
(x:_) !! 0  =  x
(_:l) !! n  =  l !! (n-1)
[]    !! _  =  undefined
```

   Reponse:
   Cet opérateurs traite la liste en entrée comme un tableau "à la C" de sorte que `l!!0` rende le premier élément de `l` et, en général, `l!!n` rende le $(n+1)$ ieme élément. Remarquez :
   — qu'un opérateur comme `!!` peut être mis en notation préfixe en mettant des parenthèse. Par exemple `((!!) l x) = (l !! x)`,
   — que l'on traite les diférentes entrées possibles directement dans les arguments ; ainsi, on écrit des codes différents pour `l !! 0` et `l !! n` lorsque $n \neq 0$, on sépare aussi le cas où la liste est vide des autres.
   — que l'on fait un appel récursif à la fonction `(!!)`, mais le second argument (l'entier) est décroissant, cette fonction à donc un sens (tant que l'on utilise avec un nombre positif et une liste fufisement grande),
   — on utilise souvant l'*underscore* `"_"` à la place d'une variable, is s'agit d'une bonne pratique pour indiquer (au compilo et au lecteur) que la variable n'est pas utilisée,
   — le `undefined` à la fin est là pour relever une erreur en cas d'utilisation annormal, on verra plus tard comment relever des erreurs qui font plus de sens.

2. Décrivez l'évaluation de `[0,1,2,3]!!3`.
   Reponse:

```
[0,1,2,3]!!3 ~= (0:[1,2,3]) !! 3
             ~> [1,2,3] !! (3-1)
             ~> [1,2,3] !! 2
```

```
~= (1:[2,3]) !! 2
~> [2,3] !! (2-1)
~> [2,3] !! 1
~= (2:[3]) !! 1
~> [3] !! (1-1)
~> [3] !! 0
~= (3:[]) !! 0
~> 3
```

3. Que font les programmes suivants ? Donner leur évaluation

```
un :: Int
un = un
```

Reponse:
Ce programme boucle : il s'appel lui même et n'a pas de cas d'arrêt

```
un ~> un ~> un ~> un ~> .....
```

```
deux :: Int
deux = (\x -> 2) un
```

Reponse:
Ce programme rend `2` car le programe `un` qui boucle n'est jamais appellé.

```
deux ~> (\x -> 2) un ~> 2{x:=un} ~= 2
```

```
trois :: Int
trois = (\x -> x+2) un
```

Reponse:
Ce programme boucle car le programe `un` est appelé.

```
trois ~> (\x -> x+2) un ~> (x+2){x:=un} ~= un+2 ~> un+2 ~> un+2 ~> ......
```

```
quatre :: Int
quatre = [1, 2, 3, 4] !! 3
```

Reponse:
Ce programme rend `4` car on passe sur le premier élément de la liste sans l'évaluer

```
quatre ~> [un, 2, 3, 4] !! 3
       ~= (un:[2,3,4]) !! 3
       ~> [2,3,4] !! (3-1)
       ~> [2,3,4] !! 2
       ~= (2:[3,4]) !! 2
       ~> [3,4] !! (2-1)
       ~> [3,4] !! 1
       ~= (3:[4]) !! 1
       ~> [4] !! (1-1)
```

```
      ~> [4] !! 0
      ~= (4:[]) !! 0
      ~> 4
```

```
cinq  :: Int
cinq  = [un, 2, 3, 4, 5] !! 4
```

Reponse:
Ce programme rend `5` car le `un` n'est jamais évalué :

```
cinq  ~> [un,2,3,4,5] !! 4
      ~> (un:[2,3,4,5]) !! 4
      ~> [2,3,4,5] !! (4-1)
      ~> [2,3,4,5] !! 3
      ~> (2:[3,4,5]) !! 3
      ~> [3,4,5] !! 2
      ~> (3:[4,5]) !! 2
      ~> [4,5] !! 1
      ~> (4:[5]) !! 1
      ~> [5] !! 0
      ~> (5:[]) !! 0
      ~> 5
```

```
six   :: Int
six   = [4..] !! 2
```

Reponse:
Ce programme rend `4`, car `[2..]` est la liste (infinie) de tous les entiers supérieurs à `2`

```
six   ~> [4..] !! 2
      ~> (4:[(4+1)..]) !! 2
      ~> [(4+1)..] !! (2-1)
      ~> ((4+1):[(4+1+1)..] !! (2-1)
      ~> ((4+1):[(4+1+1)..] !! 1
      ~> [(4+1+1)..] !! (1-1)
      ~> ((4+1+1):[(4+1+1+1)..] !! (1-1)
      ~> ((4+1+1):[(4+1+1+1)..] !! 0
      ~> 4+1+1
      ~> 5+1
      ~> 6
```

Dans la suite on trichera lorsqu'il y a des additions : on les fera imédiatement alors que Haskell atend pour les faire.

```
sept :: Int
sept = [2..]  !!  (-1)
```

Reponse:
Ce programme diverge car on appellera `[3..]!!(-2)`, puis `[4..]!!(-3)`, puis `[5..]!!(-4)` ...etc...

```
sept ~> [2..]  !!  (-1)
     ~> (2:[3..]) !! (-1)
     ~> [3..] !! (-2)
     ~> (3:[4..] !! (-2)
     ~> [4..] !! (-2)
     ~> ...
```

```
huit   :: Int
huit   = (+2) ([4..6]  !!  2)
```

Reponse:
Ce programme rend `8` car on va chercher 6 dans la liste et on lui ajoute 2 :

```
sept ~> (+2) ([4..6]  !!  2)
     ~>  ([4..6]  !!  2)     + 2
     ~>  ((4:[5..6])  !!  2) + 2
     ~>  ([5..6]  !!  1)     + 2
     ~>  ((5:[6..6])  !!  1) + 2
     ~>  ([6..6]  !!  0)     + 2
     ~>  ((6:[])  !!  0)     + 2
     ~>  6                   + 2
     ~>  8
```

```
neuf   :: Int
neuf   = [5..9]  !!  5
```

Reponse:
On rend "undefined" ici, car `[[5..9]` est la liste des entier de 5 à 9, il y en a que 5, et on demande le 6ieme

```
sept ~> [5..9]  !!  5
     ~> (5:[6..9])  !!  5
     ~> [6..9]  !!  4
     ~> (6:[7..9])  !!  4
     ~> [7..9]  !!  3
     ~> (7:[8..9])  !!  3
     ~> [8..9]  !!  2
     ~> (8:[9..9])  !!  2
     ~> [9..9]  !!  1
     ~> (9:[])  !!  1
     ~> []  !!  0
     ~> undefined
```

```
dix :: Int
dix = map (*2) [4..]  !!  1
```

Reponse:
Ce programme rend `6` car on multiplie *virtuellement* les éléments de

la liste infinie par 2, puis on prend le second élément, et c'est seulment ensuite qu'on applique vraiment la multiplication de cet élément (et pas de l'infinité d'autres éléments que l'on n'utilise pas).

```
dix ~> map (*2) [4..]   !!  1
    ~> map (*2) (4:[5..])   !! 1
    ~> ((*2) 4) : (map (*2) [5..])   !! 1
    ~> map (*2) [5..]  !! 0
    ~> map (*2) (5:[6..])  !! 0
    ~> ((*2) 5) : (map (*2) [6..])  !! 0
    ~> (*2) 5
    ~= 5 * 2
    ~> 6
```

Exercice 2 (Typage et Fonctionnel).

```
flip x y z      :: (a -> b -> c) -> b -> a -> c
flip x y z      =   x z y
```

Reponse:
Pour tout programme `f` avec deux arguments, `(flip f)` est la même fonction avec ses entrées échangées.

```
id              ::  a -> a
id              =   \x -> x
```

Reponse:
C'est la fonction iddentité qui prend un argument et le retourne sans même le regarder.

```
(||)            ::  Bool -> Bool -> Bool
(||) x y        =   if x then x else y
```

Reponse:
Il s'agit du "ou" logique entre deux bouléens.

```
(&&)            ::  Bool -> Bool -> Bool
False && _    =   False
True  && x    =   x
```

Reponse:
Il s'agit du "et" logique entre deux bouléens.

```
($)             ::  (a -> b) -> a -> b
($)             =   \x -> \y -> x y
```

Reponse:
Il s'agit de l'application : elle prend une fonction et son argument puis elle aplique la première au second.

```
(.)             :: (b -> c) -> (a -> b) -> a -> c
(.)             =   \f -> \g -> \x -> f(g x)
```

Reponse:
Il s'agit de la composition ; moralement, `f.g` est la même chose que la composition mathématique $f \circ g$.

```
owl             ::  (a -> b -> c) -> (d-> b) -> a -> d -> c
owl             =   (.) $ (.)
```

Reponse:
Ce programme (qui n'a pas beaucoup d'intéret) peut être redéfinit ainsi :

```
owl f g x y  = f x (g y)
```

```
jackpot         ::  (a -> b) -> (a -> b)
jackpot         =   ($) . ($)
```

Reponse:
Ce programme se comporte comme lapplication `($)`, sauf qu'il y a plus d'étapes de calcul...

```
dots            ::  (a -> b) -> (c -> d -> a) -> c -> d -> b
dots            =   (.) . (.)
```

Reponse:
Ce programme (qui n'a pas beaucoup d'intéret) peut être redéfinit ainsi :

```
      dots f g x y = f (g x y)
```

```
swing           ::  (a -> b) -> (b -> c) -> a -> c
swing           =   flip . (. flip id)
```

Reponse:
Ce programme est la composition inversé, qui se définit plus lisiblement comme :

```
swing f g    =  g . f
```

```
bigmap          ::  [Int -> b] -> [b]
bigmap          =   map ($3)
```

Reponse:
Ce programme [1] est plus intéressant, il s'agit d'appliquer l'argument `3` à chacune des fonctions de la liste prise en argument. Remarquez que :

- `map :: (a -> b) -> [a] -> [b]` applique une fonctions à tous les éléments d'une liste,
- `($3) :: ((Int -> b) -> b)` va apliquer l'entier `3` à son argument.

1. Si vous testez ce programme sur *ghci*, is vous indiquera quelque chose com $Numa \Rightarrow [a \to b] \to b$. On vera dans le prochain cours que 3 n'est pas forcément un entier, mais peut être n'import quel "Num", d'où ce type abscond.