PFITA – Cours/TD 3–4

UPS – L3 Info – Semestre 5

Rappels & Écriture de 2 fonctions à 3 arguments

• « Écrire une fonction permettant de connaître le min de 3 entiers, en utilisant une définition locale du min de 2 entiers » (même si cette fonction `min` est prédéfinie dans la librairie standard [1])

```
let min3 = fun x y z -> let min2 = fun a b -> if a < b then a else b
                        in min2 a (min2 b c) ;;

(* val min3 : 'a -> 'a -> 'a -> 'a = <fun> *)
```

Remarques : La fonction `min3` obtenue est polymorphe (1) (c.-à-d., son type comporte des variables de type, ici `'a`). Elle est ainsi plus "générale" que celle demandée (fonction sur les entiers). Si jamais l'on voulait "forcer" le type d'une fonction, on pourrait utiliser une annotation de type. [2]

• « Écrire une fonction permettant de connaître le nombre de racines réelles d'une équation du second degré à coefficients flottants, en utilisant une définition locale du discriminant. »

```
let nbrac a b c = let disc = b *. b. -. 4. *. a *. c in if det = 0 then 1 else
                  if disc > 0. then 2 else
                    if disc = 0. then 1 else 0 ;;

(* val nbrac : float -> float -> float -> int = <fun> *)
```

Retour sur l'ordre supérieur, le filtrage et la synthèse de type

On appelle fonctionnelle (2) une fonction qui prend en argument une autre fonction.

On appelle *fonction d'ordre supérieur* une fonction qui prend en argument une autre fonction, ou qui renvoie une autre fonction (c'est-à-dire qu'il s'agit d'une fonction curryfiée (3)).

Questions

1) Écrire la fonction `consTriplet` qui étant donné 2 entiers, construit le triplet constitué, du 2ème entier, du résultat de la comparaison des 2 entiers et du triple du premier. Donner son type.

```
let consTriplet = fun a b -> ((b, a=b, 3*a))
int -> int -> int * bool * int
```

(4)

2) Écrire la fonction `consCouple` qui étant donné 2 éléments construit le couple constitué de ces 2 éléments. Donner son type.

```
let consCouple = fun a b -> ((a, b), b)
'a -> 'b -> ('a * 'b) * 'b
```

(5)

1. https://ocaml.org/api/Stdlib.html
2. Non exigible en PFITA, mais voici un exemple :

```
let f1 x = x  and  f2 (x : int) = x  and  f3 x : int = x  and  f4 : int -> int = fun x -> x
(* val f1 : 'a -> 'a = <fun>
   val f2 : int -> int = <fun>  val f3 : int -> int = <fun>  val f4 : int -> int = <fun> *)
```

En pratique on mettrait plutôt ce genre d'annotations dans un fichier d'interface séparé (`.mli`)

3) Écrire la fonction `fst` qui étant donné un couple retourne la première comosante de ce couple. Donner son type.

```
let fst (a,_) -> a
'a * 'b -> 'a
```

4) Écrire la fonction `snd` qui étant donné un couple retourne la deuxième composante de ce couple. Donner son type.

```
let snd (a,b) = b   aussi  snd (_x, y) = y
'a * 'b -> 'b
```

Remarque : les fonctions '`fst`' et '`snd`' sont prédéfinies en OCaml.

5) Écrire une fonction correspondant à chaque type suivant :

```
val g1 : int * 'a -> int * 'a * 'a
val g2 : 'a * int * (int * 'b) -> int * 'b
```

```
let g1 (n,a) = (n+0, a, a)
let g2 (x, n, (m, y)) = (n+m, y)
```

(8)

Questions supplémentaires

Complétez la session OCaml suivante en donnant le type des fonctions :

```
let f1 x a = a x
val f1 : 'a -> ('a -> 'b) -> 'b
let f2 f x = f (f x)
val f2 : ('a -> 'a) -> 'a -> 'a
let f3 f = f 3 + 2
val f3 : (int -> int) -> int
```

(9)

Parmi toutes les fonctions précédentes, dites quelles fonctions sont d'ordre supérieur et pourquoi.

min3, mbRac, consTriplet, consCouple, f1, f2, f3
curryfiées — fonctionnelles

(10)

Exercice sur l'ordre supérieur : soit g une fonction prenant en argument un couple d'entiers. Pouvez-vous écrire la fonction `curry` la plus générale possible qui transforme g en fonction curryfiée ?

```
let curry g a b = g (a,b) ;; in g (a,b)
```

(11)

Pouvez-vous définir une telle fonction g et donner un exemple d'utilisation de votre fonction `curry` ?

```
let g (a,b) = a + b ;;
-> let gg = curry g ;;
```

(12)

```
-> gg 4 4 ;;
- : int = 8
```

Soit h une fonction prenant 2 arguments. Pouvez-vous écrire la fonction uncurry faisant l'inverse de la fonction curry ?

```
let uncurry h (a,b) = h a b
```

(13)

Pouvez-vous donner un exemple d'utilisation de votre fonction uncurry en choisissant pour h la multiplication entière ?

```
let h a b = a * b;;
let h2 = uncurry h
val h2 : int * int -> int
```

(14)

Exercice sur le filtrage : pouvez-vous ré-écrire la fonction suivante de la façon la plus concise possible ?

```
let s = fun z -> fun t -> match (z, t) with ((a, b), (c, d)) -> (a -. c, b -. d)
```

```
let s (a,b) (c,d) = (a -. c, b -. d);;
val s : f * f -> f * f -> f * f
```

(15)

Synthèse d'expressions

Donnez une expression du type `unit * (int * bool)` ~ `let a = (1), (2, true)` (16)

Donnez une expression du type `'a * 'b -> 'b * 'a` ~ `let f (a,b) = (b,a)` (17)

Donnez une expression du type `('a -> 'a) * int` ~ `let f1 = (fun a->a, 2)` (18)

Définissez des fonctions ayant les types ci-dessous.

```
val h1 : bool -> 'a -> 'a -> 'a

val h2 : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b

val h3 : ('a -> 'a -> 'b) -> 'a -> 'b * int

val h4 : ('a * 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c

val h5 : ('a -> 'b * 'c) -> ('b -> 'c -> 'd) -> ('d * bool -> 'e) -> 'a -> 'a -> 'e
```

```
let h1 a b c = if a then b else c;;
let h2 f1 f2 c = f1 (f2 c);;
let h3 f a = ((f a a), 1);;
let h4 f1 f2 a = f1 (a, (f2(a)))
let h5 f1 f2 f3 a1 a2 = let (b,c) = [if (a1=a2) then (f1 a1)
                                    else (f a2)] in f3 (f2 b c, true)
```

(19)

mieux

```
let h5 f1 f2 f3 a1 a2 = let (b,c) = f a1 in f3 (f2 b c, a1=a2);;
```

Un peu plus difficile, pour réviser

Définissez une fonction `f1` polymorphe mais pas d'ordre supérieur, et donnez son type :

```
let f1 a = 1 ;;
'a -> int
```

(20)

Définissez une fonction `f2` monomorphe (c'est-à-dire non polymorphe), non curryfiée, qui soit une fonctionnelle, et donnez son type :

```
let f2 g = g 1 + 2 ;;
(int -> int) -> int
```

(21)

Définissez une fonction `f3` monomorphe curryfiée, qui ne soit pas une fonctionnelle ; donnez son type :

```
let f3 = (+) ;;   (let f3 a b = a+b ;;)
int -> int -> int
```

(22)

La fonction `f4` ci-dessous a un défaut caractérisé ; pouvez-vous dire lequel ? puis ré-écrire la fonction pour l'éviter ?

```
let f4 f g x = g (fst (f x)) (snd (f x))
val f4 : ('a -> 'b * 'c) -> ('b -> 'c -> 'd) -> 'a -> 'd
```

```
calcul de f(x) en double.
let f4 f g x = let bc = f x in let (b, c) = bc in g b c ;;
```

(23)

Fonctions récursives sur les entiers (CTD 4) – Rappel 1

Voici le code de la fonction `fact` qui pour un entier `n`, calcule `n!` tel que `0! = 1` et `n! = n × (n-1)!`.

```
let fact n =
  let rec fac k =
    if k = 0 then 1 else k * fac (k - 1) in
  if n < 0 then failwith "fact"
  else fac n
```

Il s'agit ainsi d'une fonction *partielle*, définie uniquement sur les entiers positifs ou nuls.
Noter qu'il est également possible (et conforme à la définition mathématique[3]) d'étendre la fonction sur les négatifs en renvoyant aussi 1 dans ce cas, ce qui simplifie alors le code et donne ce "one-liner" :

```
let rec fact n = if n <= 0 then 1 else n * fact (n - 1)
```

Question 2

Écrire la fonction `sommeCarres` qui pour un entier `n`, donne la somme des carrés des entiers de 1 à `n`.

```
let sommeCarres n = let rec sumSqR n =
                    if n=0 then 0 else n*n + sumSqR (n-1)
                    in if n<0 then failwith "_" else sumSqR n;
```

(24)

```
# sommeCarres 5;;
- : int = 55
```

Question 3

(a) Écrire la fonction `sommeFonction` prenant en argument une fonction `f` et un entier `n`, et calculant la somme des résultats de l'application de `f` à chaque entier de 1 à `n`.

```
let sommeFonction f n = if n<=0 then 0
                        else f n + sommeFonction f (n-1);;
```

(25)

```
# sommeFonction (fun x -> 2 * x) 10;;
- : int = 110
```

(b) En déduire une nouvelle version pour `sommeCarres`.

```
let sommeCarres_v2 = sommeFonction (fun n -> n*n);;
```

(26)

Question 4

Écrire la fonction `sommeChiffres` prenant en argument un entier `n` et calculant la somme des chiffres contenus dans `n`.

```
let rec sommeChiffres n = if abs n < 10 then n
                          else let res = sommeChiffres (n/10) in
                               n mod 10 + res;;
```

(27)

3. Car $(-2)! = \prod_{k=1}^{-2} k = 1$ *(élément neutre de la multiplication)*.

```
# sommeChiffres 456;;
- : int = 15
```

Question 5

Écrire la fonction `sommeIteree` prenant en argument un entier `n` et itérant le calcul effectué par `sommeChiffres` jusqu'à-ce-que le résultat soit inférieur à 10.

```
let rec sommeIteree n = if n<0 then failwith "—" else
    if n<10 then n else sommeIteree (sommeChiffres n);;
```

(28)

```
# sommeIteree 456;;
- : int = 6
```

Question 6

Note : cet exercice peut éventuellement être commencé en TD ; il sera repris et terminé en TP.

Écrire les fonctions qui, étant donné un entier non nul :

1. Renvoie son dernier chiffre (`dernierChiffre`)
2. Renvoie cet entier privé de son dernier chiffre (`toutSaufDernier`)
3. Compte le nombre d'occurrence d'un chiffre dans l'écriture décimale d'un entier (`nombreOccs`)
4. Renvoie le premier chiffre d'un entier non nul (`premierChiffre`)
5. Renvoie son argument privé de son 1er chiffre (`toutSaufPremier`)
6. Teste si un entier est un palindrome (`estPalindrome`)

```
1 let dernierChiffre x = x mod 10;;
2 let toutSaufDernier x = (x - dernierChiffre x) / 10;;
3 let rec nombreOccs x n = if abs n<10 then if x=n then 1
      else 0 else nbOcc x (n/10) + if x = (n mod 10)
      then 1 else 0 ;;
4 let rec premierChiffre x = if x<10 then x else premierChiffre (x/10);;
5 let rec toutSaufPremier x = if x<10 then 0 else
      (dernierChiffre x) + (toutSaufPremier (x/10))*10;;
6 let rec estPalindrome x = (x<10) || ((premierChiffre x = dernierChiffre x)
      && estPalindrome (toutSaufPremier (toutSaufDernier x)));;
```

(29)