TD : préparation au partiel

Typage

Exercice 1. Considérer les définitions suivantes :

```
# let somme (x,y) = x + y;;
# let g x = fun f -> f(x +. 0.);;
# let doublesingleton x = [(x,x)];;
# let rec chercher x =  function
   None -> -1
 | Some l ->
       ( match l with
         [] -> 0
       | (t::q) ->
           if (x = t) then 1 else
             let
               i = chercher x (Some q)
             in
              if (i = 0) then 0 else
                i + 1
       );;
```

Calculer les types des noms ainsi définis : `somme`, `g`, `doublesingleton`, `chercher`.

Solution. On laisse Ocaml trouver les types :

```
# let somme (x,y) = x + y;;
val somme : int * int -> int = <fun>
# let g x = fun f -> f(x +. 0.);;
val g : float -> (float -> 'a) -> 'a = <fun>
# let doublesingleton x = [(x,x)];;
val doublesingleton : 'a -> ('a * 'a) list = <fun>
# let rec chercher x =  ...
val chercher : 'a -> 'a list option -> int = <fun>
```

□

Évaluation

Exercice 2. Entre les expressions suivantes lesquelles sont des valeurs et lesquelles ne le sont pas :

1. `# 3 + 4;;`
2. `# fun x -> x + 3 ;;`
3. `# let x = 1. in x+.5.;;`
4. `# (fun x -> fun f -> f(x)) 33 ;;`

Solution.

```
# 3 + 4;;
- : int = 7
# fun x -> x + 3;;
- : int -> int = <fun>
# let x = 1. in x+.5.;;
- : float = 6.
# (fun x -> fun f -> f(x)) 33;;
- : (int -> '_a) -> '_a = <fun>
```

```
# (fun x -> fun f -> f(f(x)))
  ((fun y -> y + 1) 3)
  ((fun z -> (fun w -> (w+ z))) 4);;
    - : int = 12
```

Seulement la deuxième expression n'est pas évaluée par Ocaml (il s'agit d'une expression ayant la forme `fun ... -> ...`), et donc il s'agit du seul valeur. La troisième expression est bien évaluée :

```
(fun x -> fun f -> f(x)) 33 --->
fun f -> f(33) :  int -> 'a
```

□

Exercice 3. Donner les étapes de l'évaluation par valeur ET de l'évaluation par nom des expressions suivantes :

1. `# (fun x -> fun f -> f(f(x)))`
   `((fun y -> y + 1) 3)`
   `((fun z -> (fun w -> (w+ z))) 4) ;;`
2. `# (fun x -> (fun f -> x + f(x)))(3 + 5)((fun y -> (fun z-> z*y)) 4);;`

Solution. Rappelons d'abord que dans l'évaluation de `e1 e2 e3`, le parenthèses implicites sont `((e1 e2) e3)`.
Evaluation par VALEUR :
1) :

```
(fun x -> fun f -> f(f(x)))
((fun y -> y + 1) 3)
((fun z -> (fun w -> (w+ z))) 4)
----->
(fun x -> fun f -> f(f(x)))
(3 + 1)
((fun z -> (fun w -> (w+ z))) 4)
----->
(fun x -> fun f -> f(f(x)))
4
((fun z -> (fun w -> (w+ z))) 4)
----->
fun f -> f(f(4))
((fun z -> (fun w -> (w+ z))) 4)
----->
fun f -> f(f(4))
(fun w -> (w+ 4))
----->
(fun w -> (w+ 4)) ((fun w -> (w+ 4)) 4)
----->
(fun w -> (w+ 4)) (4+ 4)
----->
(fun w -> (w+ 4)) 8
----->
(8+ 4)
----->
12
```

2) :

```
(fun x -> (fun f -> x + f(x)))
(3 + 5)
((fun y -> (fun z-> z*y)) 4)
----->
(fun x -> (fun f -> x + f(x)))
8
((fun y -> (fun z-> z*y)) 4)
```

```
----->
(fun f -> 8 + f(8))
((fun y -> (fun z-> z*y)) 4)
----->
(fun f -> 8 + f(8))
(fun z-> z*4)
----->
8 + ((fun z-> z*4)(8))
----->
8 + (8*4)
----->
8 + 32
----->
40
```

Evaluation par NOM :
1) :

```
(fun x -> fun f -> f(f(x)))
((fun y -> y + 1) 3)
((fun z -> (fun w -> (w+ z))) 4)
----->
fun f -> f(f((fun y -> y + 1) 3))
((fun z -> (fun w -> (w+ z))) 4)
----->
((fun z -> (fun w -> (w+ z))) 4)
(((fun z -> (fun w -> (w+ z))) 4) ((fun y -> y + 1) 3))
----->
fun w -> (w+ 4)
(((fun z -> (fun w -> (w+ z))) 4) ((fun y -> y + 1) 3))
----->
(((fun z -> (fun w -> (w+ z))) 4) ((fun y -> y + 1) 3)) + 4
----->
((fun w -> (w+ 4)) ((fun y -> y + 1) 3)) + 4
----->
((((fun y -> y + 1) 3)+ 4))  + 4
----->
((3 + 1) + 4)  + 4
----->
...
----->
12
```

2) :

```
(fun x -> (fun f -> x + f(x)))
(3 + 5)
((fun y -> (fun z-> z*y)) 4)
----->
fun f -> (3+5) + f(3 + 5)
((fun y -> (fun z-> z*y)) 4)
----->
(3+5) + ((fun y -> (fun z-> z*y)) 4)(3 + 5)
----->
(3+5) + (fun z-> z*4)(3 + 5)
----->
(3+5) + (3 + 5)*4
...
----->
40
```

□

Exercice 4. Considérer la définition suivante :

```
# let sialorssinon x y z = if x then y else z;;
```

1. Trouver 3 expressions `e1`, `e2` et `e3` telles que l'évaluation par valeur de `sialorssinon e1 e2 e3` n'amène pas au même résultat que l'évaluation par valeur de `if e1 then e2 else e3`.
2. Définir une fonction `autresialorssinon`, de type est `bool -> (unit -> 'a) -> (unit -> 'a) -> 'a`,telle que :
   (*) le résultat de l'évaluation par valeur de `autresialorssinon e1 (fun () -> e2) (fun () -> e3)` est toujours égal au résultat de l'évaluation par valeur de `if e1 then e2 else e3`.
3. Justifier votre définition en expliquant la raison pour la quelle la propriété notée (*) est vraie.

Solution. 1)

```
# let rec fix f = f(fix f);;
val fix : ('a -> 'a) -> 'a = <fun>
# sialorssinon true 0 (fix (fun x -> x + 1));;
Stack overflow during evaluation (looping recursion?).
# if true then 0 else (fix (fun x -> x + 1));;
- : int = 0
```

2)

```
# let autresialorssinon x y z = if x then y ()  else z ();;
val autresialorssinon : bool -> (unit -> 'a) -> (unit -> 'a) -> 'a = <fun>
# autresialorssinon true (fun() -> 0) (fun() -> (fix (fun x -> x + 1)));;
- : int = 0
```

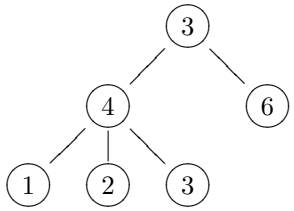
3) Soit `e : 'a` une expression dont l'évaluation par valeur entraîne des calculs qui ne se terminent pas. L'expression semblable `fun () -> e : unit -> 'a` est une valeur et donc cette expression n'est pas évaluée. Seulement une application de cette expression à l'unique objet de type `unit` – c-à-d. l'expression `(fun () ->  e) ()` – entraînera une réécriture en `e` et déclenchera l'évaluation de `e`. □

Types récursifs

Dans les exercices suivants nous allons considérer le type des arbres à branchement fini, défini comme il suit :

```
#type 'a arbre = Noeud of 'a * 'a arbre list ;;
```

Exercice 5. Considérer l'arbre suivante :



Représenter cet arbre en Caml comme un objet de type `int tree` (donc : utiliser la syntaxe Caml pour le expressions ayant ce type).

Solution.

```
# let arb = Noeud(3,[
                  Noeud(4,[
                          Noeud(1,[]);Noeud(2,[]);Noeud(3,[])
                        ]);
                     Noeud(6,[])
                ]);;
val arb : int arbre = ...
```

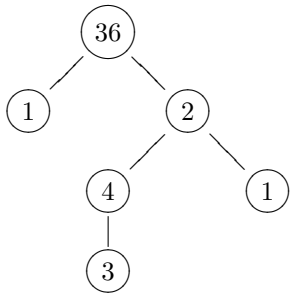
□

Exercice 6. Vice-versa, considérer l'expression suivante Caml, ayant type `int tree` :

```
# Noeud(36,[
   Noeud(1,[]);
   Noeud(2,[
      Noeud(4,[Noeud(3,[])]);
      Noeud(1,[])
     ])
   ]) ;;
```

Faire un dessin de cet objet.

Solution.



□

On définit l'itérateur `iter_arbre` comme il suit :

```
# let rec iter_arbre funetiquette premier coller =
  function Noeud(e,fils) ->
  let
    recur arb = iter_arbre funetiquette premier coller arb
  in
  let
    funcoller x arb = coller x (recur arb)
  in
    funetiquette e (List.fold_left funcoller premier fils) ;;
```

Exercice 7. Considérer la définition suivante :

```
# let algox arb =
  let
    funetiquette e n = e + n
  and
    premier = 0
  and
    coller x y = if x < y then y else x
  in
    iter_arbre funetiquette premier coller arb;;
```

1. Calculer la valeur de `algox arb` quand l'expression `arb` est évalué a) à l'arbre de l'exercice 5, b) à l'arbre de l'exercice 6.
2. Décrire, en français, ce que la fonction `algox` calcule en général.

Solution. 1.a) : on obtient l'entier 10, 1.b) on obtient l'entier 45.
2) Une branche d'un arbre est un chemin de la racine vers une feuille. Pour une telle branche π, écrivons $n \in \pi$ si n est un noeud sur cette branche, et pour un tel noeud soit $e(n)$ son étiquette. Posons

$$\Sigma(\pi) = \sum_{n \in \pi} e(n)$$

La valeur de `algox arb` est le maximum des entiers $\Sigma(\pi)$ pour π est une branche de `arb` :

$$algox\ \ arb = \max\{\, \Sigma(\pi) \,|\, \pi \text{ est une branche de } arb \,\}\,.$$

□

Exercice 8. Dans les questions suivantes, il faudra instancier les paramètres `funetiquette`, `premier` et `coller` de la fonction `iter_arbre` par des fonctions définies par vous.

1. Se servir de l'itérateur `iter_arbre` pour définir une fonction `mult_arbre`, dont le type est`int arbre -> int`, qui calcule le produit de tous les étiquettes de l'arbre.
2. Se servir de l'iterateur `iter_arbre` pour définir une fonction `contient`, dont le type est `int arbre -> int -> bool`, qui calcule si un entier est une étiquette d'un noeud de l'arbre.

Solution. 1)

```
# let mult_arbre arb =
  let
    funetiquette e n = e * n
  and
    premier = 1
  and
    coller x y = x * y
  in
    iter_arbre funetiquette premier coller arb;;
val mult_arbre : int arbre -> int = <fun>
# mult_arbre arb;;
- : int = 432
```

2)

```
# let contient arb x =
  let
    funetiquette e b = (e = x) or b
  and
    premier = false
  and
    coller x y = x or y
  in
    iter_arbre funetiquette premier coller arb;;
val contient : 'a arbre -> 'a -> bool = <fun>
# contient arb 2;;
- : bool = true
# contient arb 33;;
- : bool = false
```

□

Exercice 9. Considérer les deux définitions de la fonction `reverse` :

reverse.ml

```
 1 : let rec reverse1 = function
 2 :     [] -> []
 3 :   | t::q -> reverse1 q @ [t] ;;
 4 :
 5 : let reverse2 liste =
 6 :   let rec revacc liste acc =
 7 :     match liste with
 8 :         [] -> acc
 9 :       | t::q -> revacc q (t::acc)
10 :   in
11 :     revacc liste [];;
```

1. Laquelle, selon vous, est la définition plus performante ? Justifiez votre réponse.
2. Démontrer que pour toute liste `l` on a `reverse1 l = reverse2 l`. Suggestion : démontrer la rélation

```
revacc l acc = (reverse1 l)@acc
```

Solution.

1. La deuxième définition est meilleure car le calcul de la valeur de `reverse2 l` s'achève en temps linéaire dans la longueur dans la liste. On remarque par contre que l'ajout en queue d'un élément à un liste (expression `(reverse1 q)@[t]` demande temps linéaire dans la longueur de la liste. Étant donné que cet ajout en queue se fait un nombre de fois proportionnel à la longueur de la liste, on obtient que l'évaluation de `reverse1 l` se fait en temps quadratique par rapport à la longueur de la liste `l`. On remarque aussi que la fonction `revacc` est récursif terminale.
2. Démontrons que

   `revacc [x1;x2; ... xn] acc = (reverse1 [x1;x2; ... xn])@acc`

   Si `l = []` :
   `revacc [] acc -> []`, `reverse1 [] - []`, et donc `(reverse1 [])@acc -> []@acc -> acc`.
   Si `l = t : :q` :
   `revacc t : :q acc -> revacc q t : :acc` ce qui donne la même valeur (par hypothèse d'induction) de `(reverse1 q)@(t : :acc)`. D'ailleurs `reverse1 t : :q -> (reverse1 q)@[t]` et donc

   `(reverse1 t : :q)@acc -> (reverse1 q)@[t]@acc = (reverse1 q)@(t : :acc)`

   par des propriétés bien connue de la concaténation des listes.

□

Programmation

Exercice 10. Implementer un type des ensemble, en accord avec la signature suivante :

```
type 'a ens
(* l'ensemble vide *)
empty : 'a ens
(* ajout d'un élément à un ensemble *)
ajouter : 'a ens -> 'a -> 'a ens
(* substraction d'un élément d'un ensemble *)
enlever : 'a ens -> 'a -> 'a ens
(* appartenance d'un élément à un ensemble *)
mem : 'a ens -> 'a -> bool
(* comme enlever, mais *)
(* enlever_force retournera l'exception NotFound *)
(* si l'élément n'est pas dans l'ensemble *)
enlever_force : 'a ens -> 'a -> 'a ens
(* intersection, comme d'habitude *)
intersection : 'a ens - 'a ens -> 'a ens
(* conversion de et vers les listes *)
to_list : 'a ens -> 'a list
of_list : 'a list -> 'a ens
```

Solution.

ens.ml

```
 1 : type 'a ens = 'a list;;
 2 :
 3 : let empty = [];;
 4 :
 5 : let rec ajouter ens el =
 6 :   match ens with
 7 :       [] -> [el]
 8 :     | t::q -> if(t = el) then ens
 9 :       else t::(ajouter q el) ;;
10 :
11 : let rec enlever ens el =
12 :   match ens with
13 :       [] -> []
14 :     | t::q -> if(t = el) then q
15 :       else t::(enlever q el) ;;
16 :
17 : let mem ens el = List.mem el ens;;
18 :
19 : exception NotFound;;
20 :
21 : let rec enlever_force ens el =
22 :   match ens with
23 :       [] -> raise NotFound
24 :     | t::q -> if(t = el) then q
25 :       else t::(enlever_force q el) ;;
26 :
27 : let intersection ens1 ens2 =
28 :   List.filter (fun x -> List.mem x ens2) ens1;;
29 :
30 : let to_list ens = ens;;
31 :
32 : let rec of_list = function
33 :     [] -> []
34 :   | t::q -> ajouter (of_list q) t;;
35 :
36 : let e1 = ajouter empty 1;;
37 : let e2 = ajouter e1 1;;
38 : let e3 = enlever (ajouter e2 3) 1;;
39 : mem e1 1;;
40 : let e3 = enlever_force (ajouter e2 3) 4;;
41 : intersection e1 e3;;
42 : to_list e1;;
43 : of_list [1;2;4;3;2;1;5;5;5;7];;
```

□