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`.

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 ;;`

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);;`

Exercice 4. Considérer la définition suivante :

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

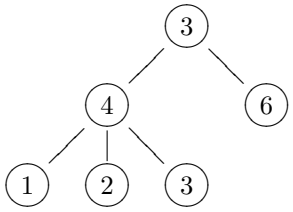
1. Trouver 3 expressions `e1`, `e2` et `e3` telles que l'évaluation par valeur de `sialorssinon e1 e2 e3` n'amène pas au même résultat que l'évaluation par valeur de `if e1 then e2 else e3`.
2. Définir une fonction `autresialorssinon`, de type est `bool -> (unit -> 'a) -> (unit -> 'a) -> 'a`,telle que :
   (*) 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.
```

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).

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.
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.

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.

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`

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
```