13 novembre 2006
Durée : 2 heures
Documents autorisés

Examen : Logique, Déduction et Programmation
Master 1 Informatique 2006-2007

Examen partiel

Typage

Exercice 1. (3 points) Considérez le code suivant :

```
 1 let str_concat l ouv fer sep =
 2   let rec interieur l =
 3     match l with
 4         [] -> failwith "str_concat : liste vide"
 5       | [t] -> t
 6       | t::q -> t^sep^(interieur q)
 7   in
 8     ouv@(interieur l)@fer;;

10 let rec pgcd x y =
11   let z = x - x/.y in
12     if z = 0 then y else (pgcd y) z ;;

14 let rec fold_left f a = function
15     [] -> a
16   | t::q -> fold_left f (f (a,t)) q ;;
```

Pour chaque fonction définie (`str_concat`, `pgcd`, `fold_left`) :

1. dites si la définition provoque un erreur de type. Si c'est le cas
   – expliquez pourquoi l'erreur se produit,
   – corrigez la définition (de façon à ne pas provoquer des erreurs de type).
   – calculez le type de la fonction une fois cette-ci a été corrigée.
2. expliquez, en langue française et par des exemples, ce que la fonction calcule.

Solution. On laisse OCaml nous expliquer les erreurs de type qui se produisent :

```
# let str_concat l ouv fer sep =
  let rec interieur l =
    match l with
        [] -> failwith "str_concat : liste vide"
      | [t] -> t
      | t::q -> t^sep^(interieur q)
  in
    ouv@(interieur l)@fer;;
              Characters 181-194:
      ouv@(interieur l)@fer;;
          ^^^^^^^^^^^^^
This expression has type string but is here used with type 'a list

# let rec pgcd x y =
    let
      z = x - x/.y
    in
      if z = 0 then y else (pgcd y) z;;
        Characters 42-43:
        z = x - x/.y
                ^
This expression has type int but is here used with type float
```

Observons que `fold_left` ne produise pas des erreurs de type :

```
# let rec fold_left f a = function
    [] -> a
  | t::q -> fold_left f (f (a,t)) q;;
    val fold_left : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

On peut corriger les deux premières définitions comme il suit, en laissant OCaml calculer les types :

```
# let str_concat l ouv fer sep =
  let rec interieur l =
    match l with
        [] -> failwith "str_concat : liste vide"
      | [t] -> t
      | t::q -> t^sep^(interieur q)
  in
    ouv^(interieur l)^fer;;
              val str_concat : string list -> string -> string -> string -> string =

# let rec pgcd x y =
    let
      z = x - x/y
    in
      if z = 0 then y else pgcd y z;;
        val pgcd : int -> int -> int = <fun>
```

.

La fonction `str_concat` formate une liste de chaînes de caractères en séparant les éléments de la liste par la chaîne de caractères `sep`. Un chaîne ouvrante et une fermante seront préfixes et postfixes. Par exemple,

```
str_concat ["a";"b";"c"] "(" ")" ","
```

sera évaluée à `"(a,b,c)"`.
La fonction `pgcd` calcule le plus grand diviseur commun de deux entiers.
Enfin, la fonction `fold_left` se comporte (presque exactement) comme la fonction `fold_left` du module `List`, sauf qu'elle prend en paramètre une fonction binaire de-currifiée. On aurait pu écrire

```
let fold_left f a l =
  let
    g x y = f (x,y)
  in
    List.fold_left g a l
```

□

Évaluation

Exercice 2. (2 points) Lesquelles des expressions suivantes sont des valeurs, lesquelles ne sont pas des valeurs ? Justifiez vos réponses.

1. `3 + 4`

   Solution. Pas un valeur car on peut réduire cette expression à la valeur `7`, □

2. `(fun x -> x + 2) 3`

   Solution. Pas un valeur car on peut réduire cette expression à l'expression `3 + 2`, □

3. `(fun x -> fun y -> x y)`

   Solution. Il s'agit d'un valeur, ca r on ne peut pas réduire cette expression. □

4. `(fun x -> fun y -> x y) (fun n -> n + 3)`

   Solution. Pas un valeur car on peut réduire cette expression à l'expression `fun y -> (fun n -> n + 3) y  3 + 2`, □

5. `fun y -> (fun n -> n + 3) y`

   Solution. Il s'agit d'un valeur, car par définition on n'opère pas de réductions à l'intérieur du corps d'une fonction (c.-à.-d. sous un `fun`). □

Exercice 3. (1 points) Expliquez qu'est-ce que la β-réduction.

Solution. Une β-réduction s'applique à une expression de la forme `v1 e2`, où `v1` est un valeur de type fonctionnel, à savoir de la forme `fun x -> c`.
La β-réduction transforme l'expression `(fun x -> c) e2` dans l'expression `c[e2/x]`, à savoir que l'on obtient à partir de `c` si l'on remplace toute occurrence de la variable `x` par l'expression `e2`. □

Exercice 4. (2 points) Soit `e1` l'expression `fun x -> 0` et `e2` l'expression `fun x -> x + x`. Soit `e3` une expression arbitraire de type `int`. Pour chacune des expressions `e1 e3` et `e2 e3`, proposez la stratégie d'évaluation plus performante. Justifiez vos réponses et apportez des exemples.

Solution. La stratégie par nom sera plus performante pour `(fun x -> 0)  e3` et la stratégie par valeur sera plus performante pour `(fun x -> x + x) e3`.
Par exemple, soit `e3` l'expression `1 + 1`.
Évaluation par nom de `(fun x -> 0) e3` :

```
(fun x -> 0) (1 + 1) -> 0
```

l'évaluation se fait en une seule étape par β-réduction.
Évaluation par valeur de `(fun x -> 0) e3` :

```
(fun x -> 0) (1 + 1) -> (fun x -> 0) 2 -> 0
```

l'évaluation se fait en plusieurs étapes.
Évaluation par valeur de `(fun x -> x + x) e3` :

```
(fun x -> x + x) (1 + 1) -> (fun x -> x + x) 2 -> 2 + 2 -> 4
```

Évaluation par nom de `(fun x -> x + x) e3` :

```
(fun x -> x + x) (1 + 1) -> (1 + 1) + (1 + 1)
                         -> 2 + (1 + 1)
                         -> 2 + 2
                         -> 4
```

On remarque que les calculs nécessaires pour réduire en valeur l'expression `1 + 1` ont été dupliqués dans l'évaluation par valeur. □

Fonctions récursives

Exercice 5. (4 points) Considérez les deux fonctions

```
1 let rec rev_append l1 l2 = match l1 with
2     [] -> l2
3   | t::q -> rev_append q (t::l2) ;;

5 let rec append l1 l2 = match l1 with
6     [] -> l2
7   | t::q -> t::(append q l2) ;;
```

1. Quelle fonction a une définition qui est récursive terminale ? Justifiez votre réponse.

   Solution. La fonction `rev_append` possède une définition récursive terminale car l'appel récursif n'est pas une sous-expression d'une autre expression. □

2. Démontrez que pour tout triplet de listes `l1,l2,l3` on a

   `rev_append l1 (append l2 l3) = append (rev_append l1 l2) l3` .

   Conseil : utiliser l'induction sur la structure de la liste `l1`.

   Solution. Nous démontrons cette propriété par induction sur la liste `l1`. Si `l1 = []`, alors

   ```
   rev_append [] (append l2 l3) = rev_append l2 l3
   append [] (rev_append l2 l3) = rev_append l2 l3 .
   ```

   Supposons que cette propriété est vraie pour la liste `q`, montrons qu'elle demeure vraie pour une liste de la forme `t : :q`.

   ```
   rev_append t::q (append l2 l3) = rev_append q t::(append l2 l3)
                                  = rev_append q (append t::l2 l3)
                                  = append (rev_append q t::l2) l3
                                  = append (rev_append t::q l2) l3 .
   ```

   □

Types récursifs

Considérez le type récursif `'a t` :

```
type 'a t = Vide | Noeud of int*'a*'a t *'a t
```

A l'aide de ce type (qui est une variante du type `arbre`) on se propose de stocker dans un valeur de ce type un ensemble d'informations (de type `'a`) à chercher à l'aide de clés associés (de type `int`). Chaque couple (clé,information) est l'étiquette d'un noeud.

Exercice 6. (2 points)

1. Appelons *feuille* un valeur de type `'a t` ayant la forme `Noeud(m,x,Vide,Vide)`. Écrivez une fonction

   `no_feuilles : 'a t -> int`

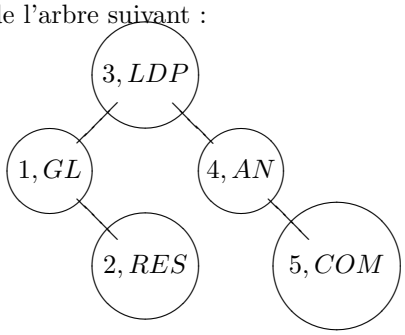
   qui calcule le nombre de feuilles dans une expression de type `'a t`.

   Solution.

```
3  let rec no_feuilles t = match t with
4      Vide -> 0
5    | Noeud(x,y,Vide,Vide) -> 1
6    | Noeud (x,y,fg,fd) -> (no_feuilles fg) + (no_feuilles fd);;
```

□

2. Écrivez une expression OCaml qui code l'arbre suivant :



Solution.

```
8  let a =
9    Noeud (3,"LDP",
10           Noeud(1,"GL",Vide,
11                 Noeud(2,"RES",Vide,Vide)),
12           Noeud(4,"AN",Vide,
13                 Noeud(5,"COM",Vide,Vide))
14         )
15 ;;
```

□

Exercice 7. (3 points) Considérez cet algorithme générique : étant donné `f`, `z`, et un valeur de type `'a t`, si cet valeur est `Vide` alors on retourne `z`, sinon, pour un valeur ayant la forme `Noeud(n,x,fg,fd)`, on applique l'algorithme de façon récursive sur `fg` et `fd` et on calcule deux valeurs ; on modifie ce deux valeurs à l'aide de la fonction `f` et de l'information `x` portée par le noeud.

1. Écrivez la définition d'un itérateur

```
iter  : ('a -> 'b -> 'b -> 'b) -> 'b -> 'a t -> b
```

qui implémente en OCaml cet algorithme.

Solution.

```
3  let rec iter f z = function
4      Vide -> z
5    | Noeud(n,x,fg,fd) ->
6        f x (iter f z fg) (iter f z fd);;
```

□

2. En utilisant l'itérateur `iter`, écrivez la définition d'une fonction qui calcule l'information maximum stockée dans un valeur de type `int t`.

Solution.

```
8  let max_info t =
9    let
10     f n1 n2 n3 = max (max n1 n2) n3
11   and
12     z = 0
13   in
14     iter f z t;;
```

□

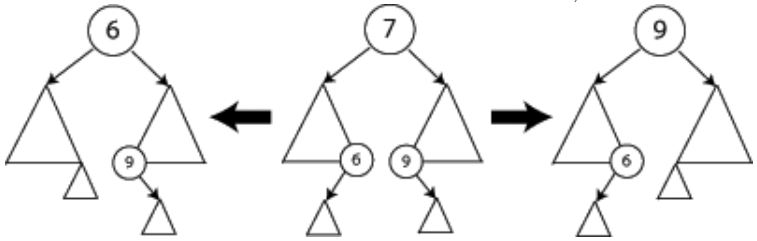
Programmation

Considérons de plus près le type des exercices 6 et 7. Le texte suivant est extrait de wikipedia.
Un arbre binaire de recherche (ABR) est un arbre binaire où chaque noeud possède une clé telle que chaque noeud du sous-arbre gauche aie une clé inférieure à celle du noeud considéré, et que chaque noeud du sous-arbre droit possède une clé supérieure à celle-ci. Les noeuds que l'on ajoute deviennent des feuilles de l'arbre. A un arbre binaire de recherche on peut appliquer les opérations suivantes.

Recherche. La recherche dans un arbre binaire d'un noeud ayant une clé particulière est un procédé récursif. On commence par examiner la racine. Si sa clé est la clé recherchée, l'algorithme termine et renvoie la racine. Si elle est strictement inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même si la clé de la racine est strictement supérieure à la clé recherchée la recherche continue sur le sous-arbre droit. Si on atteint une feuille dont la clé n'est pas celle recherchée, on sait alors que la clé recherchée n'appartient à aucun noeud.

Insertion. L'insertion d'un noeud commence par une recherche : on cherche la clé du noeud à insérer ; lorsqu'on arrive à une feuille, on ajoute le noeud comme fils de la feuille en comparant sa clé à celle de la feuille : si elle est inférieure, le nouveau noeud sera à gauche ; sinon il sera à droite.

Suppression. Plusieurs cas sont à considérer, une fois que le noeud à supprimer a été trouvé à partir de sa clé :
- Suppression d'une feuille : Il suffit de l'enlever de l'arbre vu qu'elle n'a pas de fils.
- Suppression d'un noeud avec un enfant : Il faut l'enlever de l'arbre en le remplaçant par son fils.
- Suppression d'un noeud avec deux enfants : Supposons que le noeud à supprimer soit appelé N. On l'échange alors par son successeur le plus proche (le noeud le plus à gauche du sous-arbre droit) ou son plus proche prédécesseur (le noeud le plus à droite du sous-arbre gauche). Cela permet de garder une structure d'arbre binaire de recherche. Puis on supprime N qui est alors une feuille ou un noeud avec un seul fils ; cela nous ramène au cas précédent.



Exercice 8. (7 points) Écrivez une implémentation de l'interface pour le arbres de recherche binaires suivante :

```
(* arbreb.mli : interface pour les arbres binaires de recherche*)
type 'a t = Vide | Noeud of int*'a*'a t *'a t

exception Pas_Trouve
(* Recherche : on retourne l'info associée à une clé.
   On levera l'exception Pas_Trouve
   si la clés n'appartient pas à l'abre *)
val rechercher : int -> 'a t -> 'a

(* Insertion d'un couple (clé,info) dans un arbre *)
val inserer : (int*'a) -> 'a t -> 'a t

(* Suppression d'un noeud portant une clé donnée  *)
val supprimer : int -> 'a t -> 'a t
```

Conseil : pour l'implémentation de `supprimer` écrivez une fonction `plus_proche` pour chercher le plus proche prédécesseur.

Solution.

```
1 type 'a t = Vide | Noeud of int*'a*'a t *'a t;;

3 exception Pas_Trouve;;
```

```
let rec rechercher cle arbre = match arbre with
    Vide -> raise Pas_Trouve
  | Noeud (m,x,fg,fd) -> if cle = m then x else
      if cle < m then rechercher cle fg else rechercher cle fd
;;

let rec inserer (cle,info) arbre = match arbre with
    Vide -> Noeud (cle,info,Vide,Vide)
  | Noeud (m,x,fg,fd) ->
      if cle = m then
        if info = x then Noeud (m,x,fg,fd) else
          failwith "Impossible inserer"
      else
        if cle < m then Noeud(m,x,inserer (cle,info) fg,fd)
        else  Noeud(m,x,fg,inserer (cle,info) fd)
;;

let rec plus_proche_gauche arbre = match arbre with
    Vide -> failwith "plus_proche_gauche"
  | Noeud (m,x,fg,Vide) -> (m,x)
  | Noeud (m,x,fg,fd) -> plus_proche_gauche fd
;;

let rec supprimer cle arbre = match arbre with
    Vide -> Vide
  | Noeud (m,x,fg,fd) ->
      if cle < m then Noeud (m,x,supprimer cle fg,fd) else
        if m < cle then  Noeud (m,x,fg,supprimer cle fd) else
          (
            match (fg,fd) with
                (Vide,Vide) -> Vide
              | (x,Vide) -> x
              | (Vide,y) -> y
              | (_,_) ->
                  let (m',x') = plus_proche_gauche fg in
                    Noeud (m',x',supprimer m' fg,fd)
          )
;;
```

□