7 novembre 2005
Durée : 2 heures
Documents autorisés

Examen : Logique, Déduction et Programmation
Master 1 Informatique 2005-2006

Examen partiel

Typage

Exercice 1. Les définitions suivantes produisent des erreurs de type :

```
 1 let binome x y  =
 2   let
 3     rec fact = function n ->
 4       if n <= 0. then 1. else n *. fact (n -. 1.)
 5   in
 6     fact x / (fact y *. fact (x -. y)) ;;

 8 let rec somme_bornee (x,y) f acc =
 9   if x > y then acc else
10     somme_bornee x (y-1) f (acc + f y);;

12 let rec map f = function
13     [] -> ""
14   | t::q -> (f t)^(map f);;
```

Pour chacun des noms définis (`binome`, `somme_bornee`, `map`)

1. expliquer pour quelle raison l'erreur de type se produit,
2. corriger la définition, de façon qu'un erreur de type ne se produise pas,
3. après correction, en calculer le type.

Évaluation

Exercice 2.
- Expliquer ce que veut dire que une expression du langage Ocaml est un valeur.
- Expliquer quand une expression fonctionnelle (c.-à-d. de type `'a -> 'b` où `'a` et `'b` peuvent être remplacés par des autres types) est un valeur.
- Que se passe-t'il à la définition des noms `expr1` et `expr2` :

```
let expr1 = print_endline "1"; 2;;
let expr2 () = print_endline "1"; 2;;
```

Exercice 3. Donner les étapes de l'évaluation par valeur ET de l'évaluation par nom de l'expression suivante :

```
(fun x -> fun y -> x)
2
( (fun f -> f 3) (fun x -> x +1) )
```

Fonctions récursives

Exercice 4. Considérez la fonction `append` définie par :

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

Démontrer les egalités suivantes :
– `append l1 [] = l1`,
– `append (append l1 l2) l3 = append l1 (append l2 l3)`.
Conseil : utiliser l'induction sur la longueur de la liste `l1`.

Types récursifs

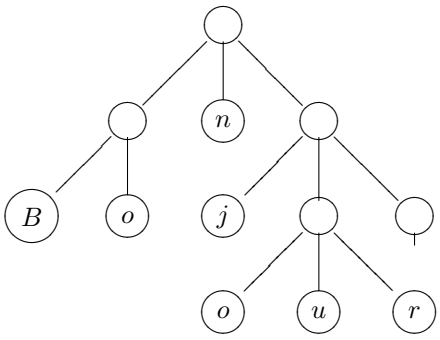
Considérez le type récursif `arbre` avec son itérateur générique `arbre_iter` :

```
 1  type arbre = Feuille of char | Noeud of arbre list ;;

 3  let rec arbre_iter funvar funcoller premier funnoeud = function
 4      Feuille c -> funvar c
 5    | Noeud fils ->
 6        let
 7          recur = arbre_iter funvar funcoller premier funnoeud
 8        in
 9          funnoeud (
10            List.fold_left funcoller premier
11                       (List.map recur fils));;
```

Un tel arbre a un nombre arbitraire fini de fils et peut stocker l'information, des caractères, sur les feuilles.

Exercice 5. Écrire une expression OCaml correspondant à l'arbre suivant :



Exercice 6. En utilisant l'itérateur générique `arbre_iter`
– écrire une fonction `no_feuilles : arbre -> int` qui compte le nombre des feuilles dans un arbre.
– écrire une fonction `frontiere : arbre -> string` qui produit la chaîne de caractères placés sur les feuilles dans l'ordre de gauche à droite. (On pourra supposer qu'une fonction `char_to_string : char -> string`, transformant un caractère en chaîne de caractères, est définie).
Par exemple :

```
# let a = Noeud [Feuille 'a'; Noeud [Feuille 'c'; Feuille 'b']];;
no_feuilles a;;
- : int = 3
# frontiere a;;
- : string = "acb"
```

Programmation

Exercice 7. On peut représenter un nombre positif en notation binaire par une suite d'entiers contenant seulement des 0 ou des 1. Le bit de poids plus faible sera en tête de la liste. Par exemple, les listes

```
[1;0;1], [1;1;0;1;0]
```

représenteront les nombres 5 et 11 respectivement. Le but de l'exercice est d'écrire une fonction `sommebin` qui calcule la (représentation binaire de la) somme de deux listes.
Plus précisément, on implémentera (le style de programmation sera fonctionnel) les fonctions décrites dans l'interface suivante :

```
(* On lèvera cette exception
   si une liste n'est pas la représentation
   binaire d'un nombre, à savoir si elle
   contient des entier différents de 0 et 1 *)
exception Nombre_non_binaire

(* Transformer une liste de 1,0 en entier  *)
val to_int : int list -> int

(* Transformer un entier en sa représentation binaire *)
val from_int : int -> int list

(* Calcule la représentation binaire
   de la somme [i] + l,
   où i in {0,1}
   et l est la représentation binaire d'un nombre
*)
val carry : int * int list -> int list

(* Calcule la représentation binaire
   de la somme [i] + l1 + l2,
   où i in {0,1}
   et l1,l2 sont des représentations binaires de deux nombres.
   On se sert pas des fonctions to_int et from_int
*)
val sommebinc : int * int list * int list -> int list

(* Calcule la représentation binaire
   de la somme  l1 + l2,
   où l1,l2 sont des représentations binaires de deux nombres.
   On se sert pas des fonctions to_int et from_int
*)
val sommebin : int list -> int list -> int list
```