Algorithmique-Programmation : IAP1 - 2 novembre 2009

Sans documents - Sans calculette

Durée : 1h30 -

Le sujet comporte **3 pages**

Corrigé

Les exercices sont indépendants. Ne vous bloquez donc pas sur une question.

Pour répondre à certaines questions, il se peut que vous ayez besoin de définir des fonctions auxiliaires. Dans ce cas indiquez clairement ce que font ces fonctions auxiliaires en donnant par exemple leur interface.
Pour les fonctions explicitement demandées dans l'énoncé, n'écrivez leur interface que si celle-ci est explicitement demandée dans l'énoncé.

Exercice 1 (**Devinez ...**)

Question 1

Complétez l'interface de la fonction `toto`. Si la fonction comporte une erreur de typage, vous l'indiquerez et expliquerez brièvement pourquoi.

```
(* interface toto
type : ..............................
args : ..............................
précondition : aucune
postcondition : ........................
raises : ............................
tests : ne pas compléter
* )
let toto = map (function x -> 2 * x);;
```

```
(* interface toto
type : int list -> int list
args : l
précondition : aucune
postcondition : retopurne la liste contenant le double de chaque élément de la liste l
raises : .rien
tests : ne pas compléter
* )
let toto = map (function x -> 2 * x);;
```

Exercice 2 (**Supprimez ...**)

On s'intéresse à la fonction `supprimer` qui prend en paramètre un élement `e` et une liste `l` et produit la liste `l` privée de toutes les occurrences de `e`.

Question 2

Quel est le type de la fonction `supprimer` ?

`'a -> 'a list -> 'a list`

Question 3

Écrire la fonction `supprimer` sous la forme d'une fonction récursive.

```
let rec supprimer e l = match l with
  [] -> []
| x::r -> if e=x then supprimer e r
          else x::(supprimer e r)
```

Question 4

Écrire la fonction `supprimer` à l'aide d'une des fonctionnelles en fin de sujet.

```
let supprimer e l = filter (function x -> not(x=e)) l;;
ou
let supprimer e l = fold_right (function x -> function y -> if x=e then y else x::y) l [];;
ou
let supprimer e l = fold_left (function x -> function y -> if y=e then x else y::x) [] l;;
```

La dernière solution ne garde pas l'ordre des éléments.

Exercice 3 (**Ensembles d'entiers**)

On définit le type `ens` pour décrire des ensembles finis d'entiers de la manière suivante :

```
type ens = Empty | Interval of int * int | Union of ens * ens ;;
```

où `Empty` désigne l'ensemble vide, `Interval (i,j)` représente l'intervalle `i..j` c'est-à-dire l'ensemble des entiers `i` à `j` (bornes comprises) (`i` et `j` sont supposés **croissants ou égaux**) et `Union (e1,e2)` représente l'union des 2 ensembles `e1` et `e2` supposés **disjoints**.

Le singleton $\{i\}$ sera représenté par la valeur `Interval (i,i)`.
Chaque fois que ce sera possible, on privilégiera la représentation sous forme d'un intervalle unique.

Ces hypothèses de représentation sont supposées vérifiées par toute valeur de type `ens`, paramètre d'une fonction. Chaque fois qu'il est demandé de construire un ensemble, il faudra faire en sorte de respecter ces contraintes.

Question 5

Définir `s1`, `s2` et `s3` correspondant respectivement aux ensembles $\{1,2,3,4,5,6,7,8\}$, $\{1,2,3,7,8\}$ et $\{-3,-2;-1;0;1;7\}$.

```
let s1 = Interval (1,8);;
let s2 = Union(Interval(1,3), Interval(7,8));;
let s3 = Union (Interval (-3,1), Interval(7,7));;
```

Question 6

Écrire une fonction `appartient` qui vérifie si un entier appartient à un ensemble représenté par une valeur de type `ens`. Elle retournera `true` si l'élement est présent et `false` sinon.

```
let appartient e f = match f with
    Empty -> false
  | Interval (a,b) ->  e >= a && e <= b
  | Union (e1 , e2) -> appartient e e1 || appartient e e2;;
```

Question 7

Écrire la fonction `card` qui calcule le cardinal d'un ensemble.

```
let rec card e = match e with
  Empty -> 0
| Interval (i,j) -> abs (j - i + 1)
| Union (e1, e2) -> (card e1) + (card e2);;

# card s2;;
- : int = 5
# card s3;;
- : int = 6
#
```

Question 8

Écrire une fonction `forall_ens` qui prend en paramètre un ensemble de type `ens` et un prédicat (fonction de type `int -> bool`) et qui retourne `true` si tous les élements de l'ensemble vérifient le prédicat et `false` sinon.

```
(*interface forall_int
tye : (int -> bool) -> int -> int -> bool
arg : p i j
precondition : aucune
postcondition : retourne true si
tous les entiers compris entre i et j vérifient le prédicat p
*)
let rec forall_int p i j =
if i > j then true
else p i && forall_int p (i+1) j;;

let rec forall_ens p e = match e with
  Empty -> true
| Interval (i,j) -> forall_int p i j
| Union (e1, e2) -> forall_ens p e1 && forall_ens p e2;;

# forall_ens (function x -> x>0) s1;;
- : bool = true
# forall_ens (function x -> x>0) s3;;
- : bool = false
```

Question 9

Écrire la fonction de conversion `to_list` d'un ensemble de type `ens` en une liste d'entiers. Si une fonction auxiliaire est nécessaire vous la définirez et donnerez son interface (sans les cas de test).

```
let rec interval a b =
  if a > b then [] else a:: (interval (a+1) b);;

let rec to_list f = match f with
    Empty -> []
  | Interval (a,b) -> interval a b
  | Union (e1, e2)  -> (to_list e1) @ (to_list e2);;

# to_list s1;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
# to_list s2;;
- : int list = [1; 2; 3; 7; 8]
# to_list s3;;
- : int list = [-3; -2; -1; 0; 1; 7]
```

ou avec une fonction intermédiaire utilisant un accumulateur pour ne pas utiliser la concaténation de listes

```
(* interface ...
args e acc
precondition a<=b
postcondition : retourne la liste des entiers compris entre a et b  concaténée à acc
*)
let rec interval_acc a b l =
```

```
  if a > b then l else a::(interval_acc (a+1) b l);;

(* interface
arg e acc
pre e est correctement représenté
post : retourne la liste des entiers de e concaténée à acc
*)
let rec to_list_acc e acc = match e with
  Empty -> acc
| Interval(i,j) -> interval_acc i j  acc
| Union(e1, e2) -> to_list_acc e1 (to_list_acc e2 acc);;

let to_list e = to_list_acc e [];;
```

Question 10

Écrire une fonction `enlever` qui prend en paramètre un ensemble `e` de type `ens` et un entier `n` et retourne le nouvel ensemble contenant les élements de `e` mais ne contenant plus l'entier `n`. Il est demandé d'utiliser la structure d'intervalle quand cela est possible.

```
let rec enlever x e = match e with
| Empty -> Empty
| Interval(i,j) -> if i=x then
                    if i=j then Empty else Interval(i-1, j)
                   else
                   if x=j then
                    if i=j then Empty else Interval(i, j-1)
                   else
                   if i<x && x<j then Union(Interval(i,x-1), Interval(x+1,j))
                   else e
| Union(e1, e2) -> Union(enlever x e1, enlever x e2)

 enlever 4 s1;;
- : ens = Union (Interval (1, 3), Interval (5, 8))
# enlever 4 s2;;
- : ens = Union (Interval (1, 3), Interval (7, 8))
# enlever 3 s2;;
- : ens = Union (Interval (1, 2), Interval (7, 8))
# enlever 7 s3;;
- : ens = Union (Interval (-3, 1), Empty)
# enlever (-1) s3;;
- : ens = Union (Union (Interval (-3, -2), Interval (0, 1)), Interval (7, 7))
```

On remarque que dans le dernier cas un des deux appels récursifs retournera l'ensemble de départ (e s'il existe ne peut se trouver que dans un des sous-ensembles.

Question 11

Écrire la fonction de conversion `to_ens` d'une liste d'entiers dans l'ordre croissant en un ensemble de type `ens`. Si une fonction auxiliaire est nécessaire vous la définirez et donnerez son interface (sans les cas de test).

```
exception Echec;;

(*interface aux_interval
```

```
type int list -> int  -> int
arguments l (la liste) r2 (l'entier)
precondition les  elements de l sont les entiers consécutifs r2 + 1, r2+2 etc.
postcondition retourne le dernier de l
raises Echec si l est vide ou si les elements de l ne sont pas les entiers consécutifs r2+1, r2+2 etc
*)
let rec aux_interval l r2  = match l with
    [] ->  raise Echec
  | [a] -> if r2 +1 = a then a else raise Echec
  | a::q -> if r2+1 = a then aux_interval a q else raise Echec
;;

let to_ens l = match l with
    [] -> Empty
  | [a] -> Singleton a
  | a::q -> try let v = aux_interval a q in Interval (a,v)
            with Echec -> List l
;;
```

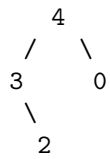
Exercice 4 (**Arbres**)

On définit le type des arbres binaires de la façon suivante :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

Question 12

Définir l'arbre ci-dessous en Ocaml. On l'appellera `ex1`. Quel est son type ?




```
let ex1 = Noeud (4, Noeud(3, Vide, Noeud (2, Vide, Vide)), Noeud (0, Vide, Vide));;
```

Il s'agit d'une valeur de type `int arbre`

Le niveau d'un nœud dans un arbre est 1 si le nœud est la racine de l'arbre, sinon son niveau est celui de son parent augmenté de 1. Par exemple,

Question 13

Écrire la fonction `valeurs_niveau` qui prend en paramètre un arbre a et un niveau (entier naturel non nul) n et retourne la liste des valeurs des nœuds de niveau n contenues dans a. Dans l'exemple précédent la liste des valeurs de niveau 2 est [3;0], la liste des valeurs de niveau 3 est [2], la liste des noeuds de niveau 4 est vide.

```
(* on suppose que le niveau est bien un entier
naturel non nul *)
let rec valeurs_niveau a n = match a with
```

```
  Vide -> []
| Noeud(r,g,d) -> if n=1 then [r]
                  else
                  (valeurs_niveau g (n-1))@(valeurs_niveau d (n-1));;
```

ou

```
(* on suppose ici que le niveau est bien un entier
naturel non nul *)
let valeurs_niveau a n =
   let rec valeurs_niveau_rec a n l = match a with
     Vide -> l
   | Noeud(r,g,d) -> if n=1 then r::l
                     else
                     valeurs_niveau_rec g (n-1)(valeurs_niveau_rec d (n-1) l)
   in valeurs_niveau_rec a n [] ;;
```

Question 14

Quel est le type de la fonction `valeurs_niveau` ?

Elle est de type `'a arbre -> int -> 'a`

Question 15

Écrire une fonction `élaguer` qui permet d'élaguer un arbre à un niveau donné. On supprime alors tous les nœuds de niveau strictement supérieur. Par exemple si on élague l'arbre `ex1` au niveau 2, on obtiendra l'arbre ci-dessous.

```
    4
   /  \
  3    0
```

Aux niveaux 3 et plus, l'arbre élagué est identique à l'arbre initial.

```
(* on suppose que le niveau est bien un entier naturel
 non nul, sinon introduire une fonction qui fait le test*)
let rec élaguer a n = match a with
  Vide -> Vide
| Noeud(r,g,d) -> if n=1 then Noeud(r,Vide, Vide)
                  else
                  Noeud(r, (élaguer g (n-1)), (élaguer d (n-1))) ;;
```

Rappel : fonctionnelles classiques sur les listes :

```
let rec map f li = match li with
  [] -> []
| x::l -> (f x)::(map f l);;

let rec filter p li = match li with
  [] -> []
| x::l -> if (p x) then x::(filter p l) else (filter p l);;

let rec fold_right f l e = match l with
  [] -> e
| a::r -> f a (fold_right f r e);;
```

```
let rec fold_left f e l = match l with
  [] -> e
| a::r -> fold_left f (f e a) r;;
```
