Algorithmique-Programmation : AP1 - Contrôle final - novembre 2010 - durée : 1h30

Sans documents - Calculette non autorisée et sans objet - Ordinateur interdit
4 pages de sujet

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 (**Petites questions**)

Question 1

Écrire une fonction qui calcule la somme des éléments pairs d'une liste.

Question 2

Quel est le type (le plus général) de la fonction `ff` définie ci-dessous ? Si l'expression est mal typée, expliquez pourquoi en une phrase.

```
let ff  f g x = (f x, g x)
```

Question 3

Quel est le type de l'expression `ff succ pred` avec `succ` et `pred` respectivement la fonction successeur et la fonction prédécesseur sur les entiers ? Si l'expression est mal typée, expliquez pourquoi en une phrase.

Question 4

Quelle est la valeur de l'expression `ff succ pred 4` ?

Question 5

Quel est le type (le plus général) de la fonction `gg` définie ci-dessous ? Si l'expression est mal typée, expliquez pourquoi en une phrase.

```
let gg (f, g) x = (f x, g x)
```

Question 6

Quel est le type de l'expression `gg succ` ? Si l'expression est mal typée, expliquez pourquoi en une phrase.

Exercice 2 (**Arbres**)

On reprend le type des arbres binaires :

```
type 'a arb = Vide | Noeud of 'a arb * 'a * 'a arb;;
```

Question 7

Définir en Ocaml l'arbre suivant que vous nommerez `a1`. Quel est le type de `a1` ?

```
let a1 = Noeud( Noeud (Vide, 3, Vide) ,
                1,
                Noeud (Noeud (Vide, 5, Vide), 4,  Vide))
;;
```

Question 8

Définir en Ocaml l'arbre suivant que vous nommerez `a2`. Quel est le type de `a2` ?

```
        (1, true)
        /       \
(3 , false)   (4, true)
```

```
let a2 =
Noeud (Noeud (Vide, (3, false), Vide), (1, true),
 Noeud (Vide, (4, true), Vide))
```

`a2` a le type `(int * bool) arb`

Question 9

Écrire la fonction `split` qui travaille avec des arbres dont les valeurs sont des couples (comme `a2`) et retourne un couple de 2 arbres. Les valeurs du premier arbre du résultat sont les premières compsantes des couples de l'arbre initial et les valeurs du deuxième arbre du résutat sont les les deuxièmes composantes des couples de l'arbre initial. Les arbres (initial et résultats) ont tous la même forme. Par exemple l'application de la fonctio `split` sur l'arbre `a2` donne les arbres `c1` et `c2` dessinés ci-dessous :

```
          1                true
        /   \              /  \
       3     4          false  true
```

```
  let rec split a  = match a with
| Vide -> (Vide, Vide)
| Noeud (g, (r1,r2), d)-> let g1, g2 = split g  in
                                       let d1, d2 = split d  in
                                       (Noeud(g1, r1, d1), Noeud(g2, r2,d2))
;;
val split : ('a * 'b) arb -> 'a arb * 'b arb = <fun>

# split a2;;
- : int arb * bool arb =
(Noeud (Noeud (Vide, 3, Vide), 1, Noeud (Vide, 4, Vide)),
 Noeud (Noeud (Vide, false, Vide), true, Noeud (Vide, true, Vide)))
```

Question 10

Quel est le type (le plus général) de la fonction `split` ?

Question 11

Écrire la fonction `combine` qui réalise le travail inverse. Elle prend deux arbres en paramètre `a` et `b` de même forme et retourne l'arbre composé de couples dont les premiers composants sont formés des valeurs qui proviennent de `a` pour les noeuds correspondants et les deuxièmes composants sont composés des valeurs provenant des noeuds de `b` correspondants. Si les deux arbres `a` et `b` n'ont pas la même forme, la fonction échoue.

Par exemple la fonction `combine` appliquée aux arbres `c1` et `c2` retourne l'arbre `a2`. Si on l'applique à `a1` et `c1`, elle échoue car les deux arbres n'ont pas la même forme.

```
# let rec combine a b = match a, b with
  Vide, Vide -> Vide
| Noeud(g1, r1, d1), Noeud(g2, r2, d2) ->
             let g= combine g1 g2 and d = combine d1 d2 in
             Noeud(g, (r1, r2), d)
| _ -> failwith"pas la meme forme";;
        val combine : 'a arb -> 'b arb -> ('a * 'b) arb = <fun>

# combine c1 c2;;
- : (int * bool) arb =
Noeud (Noeud (Vide, (3, false), Vide), (1, true),
 Noeud (Vide, (4, true), Vide))
# combine a1 c2;;
Exception: Failure "pas la meme forme".
```

Question 12

Quel est le type (le plus général) de la fonction `combine` ?

Exercice 3 (**Gestion de plannings**)

L'objectif de ce problème est de modéliser la notion de planning associé à une salle (salle de classe ou salle de réunion par exemple) et de fournir quelques fonctions d'aide à la gestion des plannings.

Un planning concerne une seule salle et pour une semaine (jours ouvrables seulement). On appelle date le nombre d'heures depuis le lundi 0h. Par exemple la date correspondant au mardi 15h est représentée par l'entier 39. Dans la suite le type `date` désigne le type `int`.

On définit le type `jour` de la manière suivante :

```
type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi ;;
```

Question 13

Écrire une fonction `date_of_jour_heure` de type `jour * int -> date` qui prend en paramètre un jour et une heure et retourne la date correspondante. On suppose que l'heure est bien comprise entre 0 et 23. Par exemple `date_of_jour_heure (Lundi, 0)` retourne 0, `date_of_jour_heure (Mardi, 15)` vaut 39.

```
# let int_of_jour j = match j with
  Lundi -> 0
| Mardi -> 1
| Mercredi -> 2
| Jeudi -> 3
| Vendredi -> 4;;
          val int_of_jour : jour -> int = <fun>

# let date_of_jour_heure (j,h) = (int_of_jour j) * 24 + h;;
val date_of_jour_heure : jour * int -> int = <fun>

date_of_jour_heure (Lundi, 0);;
#   - : int = 0

# date_of_jour_heure (Mardi, 15);;
- : int = 39
```

On appelle créneau un intervalle de temps représenté par deux dates, une date de début et une date de fin. Par exemple le créneau Lundi 15h à Lundi 17h représente un intervalle de temps de 2h commençant le lundi à 15h et se terminant à 17h (ou plus exactement à 16h59).

Question 14
Compléter l'interface de la fonction définie ci-dessous.

```
(*interface toto
type :
args : l
precondition :
postcondition :
*)
let toto  l = map date_of_jour_heure l;;
```

Dorénavant on représente un créneau par un couple de deux dates : le premier composant du couple est la date de début et le second composant la date de fin. La date de fin est supposée supérieure ou égale à la date de début. Dans la suite on appelle `creneau` le type d'un créneau soit le type `int*int`.

Un planning est une liste de créneaux, c'est précisément la liste des créneaux pendant lesquels la salle correspondante est occupée. Dans la suite le type `planning` désigne une liste de créneaux, soit le type `int*int list`. Ainsi le planning composé des créneaux lundi 8h à 10h, lundi 9h à 11h, lundi 11h à 13, lundi 14h à 16, mardi 15h à 16h est représenté par la liste `[(8,10);(9,11);(14,16);(39,40)]`.

Question 15
Écrire une fonction récursive `vérif` qui vérifie qu'un planning contient des créneaux bien formés (date de fin supérieure ou égale à la date de fin).

```
let rec verif l = match l with
   [] -> true
| (d,f)::r -> d<=f && verif r;;

let pl_ex = [(8,10);(9,11);(14,16);(39,40)];;

verif pl_ex;;
```

Question 16
Réécrire la fonction `vérif` en utilisant la fonctionnelle `forall` donnée ci-dessous.

```
let rec forall p l = match l with
   [] -> true
| x::r -> (p x) && (forall p r);;

let verif l = forall (function (x,y) ->x <= y) l;;
```

Question 17
Écrire une fonction `trie_planning` qui trie un planning selon la date de début des créneaux, et, si les dates de début sont égales, selon la date de fin.

Vous utiliserez la fonction `List.sort` de la bibliothèque standard dont le type est `('a->'a->int) ->'a list -> 'a list`, et dont voici la documentation en français :
Trie une liste dans l'ordre croissant pour une fonction de comparaison donnée. Cette fonction de comparaison doit retourner 0 si ses arguments sont égaux, un entier strictement positif si le premier argument est plus grand, et un entier strictement négatif si le premier argument est plus petit.

```
let comp (d1, f1) (d2,f2) =
if d1=d2 && f1=f2 then 0
else
if d1 < d2 || (d1=d2 && f1<f2)  then -1
else 1;;

let trie_planning l = List.sort comp l;;

trie_planning pl_ex;;

trie_planning (pl_ex@pl_ex);;
[(8, 10); (8, 10); (9, 11); (9, 11); (14, 16); (14, 16); (39, 40); (39, 40)]
```

Dans la suite on dit qu'un planning est bien formé si les créneaux sont bien formés et si la liste est triée selon les créneaux dans l'ordre croissant.

Question 18

Écrire une fonction `fusion` qui réalise la fusion de deux plannings bien formés. Le résultat obtenu doit être un planning bien formé contenant tous les créneaux des deux listes.

```
let rec fusion pl1 pl2 = match pl1, pl2 with
[], _ -> pl2
| _, [] -> pl1
| c1::r1, c2::r2 -> if comp c1 c2 =0 then c1::c2::(fusion r1 r2)
                            else if comp c1 c2 >0 then c2::(fusion pl1 r2)
                                    else c1::(fusion r1 pl2);;
fusion pl_ex pl_ex;;
```

Question 19

Écrire une fonction récursive `date_libre` de type `date -> planning -> bool` qui prend en paramètre une date et un planning et retourne true si la date n'est dans aucun créneau du planning, false sinon. Par exemple, la date 34 (i.e. Mardi 14h) est libre dans le planning exemple précédent, mais la date 9 (Lundi 9h) ne l'est pas.

```
let rec date_libre date pl = match pl with
  [] -> true
| (debut, fin)::p -> (date < debut || date >= fin) && date_libre date p;;

let date_libre date p = forall (fun (debut, fin) -> date < debut || date >= fin) p;;
```

Question 20

Réécrire la fonction précédente en utilisant la fonctionnelle `fold_right` rappelée ci- dessous.

```
let date_libre pl = fold_right (function (debut, fin) y -> (date < debut || date >= fin) &&y ) l tru
```

On appellera planning canonique un planning qui vérifie les conditions suivantes : il est bien formé ; les créneaux sont disjoints deux à deux (pas de chevauchement) ; la date de fin d'un créneau n'est pas égale à la date de début du créneau suivant.

La projection d'un planning p bien formé est le planning canonique p' qui vérifie la propriété suivante : une date est dans un créneau de p' si et seulement si elle est dans (au moins) un créneau

de p. Par exemple la projection du planning dont les créneaux sont les suivants lundi 8h à 10h, lundi 9h à 11h, lundi 11h à 13h, lundi 14h à 16h, lundi 16h à 17h, mardi 15h à 16h donne la planning canonique p' = lundi 8h à 13h, lundi 14h à 17h, mardi 15h à 16h.

Question 21

Écrire une fonction `projection` qui calcule la projection d'un planning supposé **bien formé**.

```
let rec projection pl = match pl with
| [] -> []
| [c] -> [c]
| (d1,f1)::(d2,f2)::l ->
if f1 < d2 (* strict *) then (d1,f1)::(projection ((d2,f2)::l)) else projection ( (d1, (max f1 f2))
;;

projection [(8, 10); (9, 11); (14, 16); (16,17) ; (39, 40)]
;;
- : (int * int) list = [(8, 11); (14, 17); (39, 40)]
```

Question 22

Écrire une fonction `compl` qui calcule le planning canonique p' complémentaire d'un planning p donné **supposé en forme canonique**. C'est-à-dire : une date est dans un créneau de p' si et seulement si elle n'est dans aucun créneau de p.

Par exemple le complément du planning précédent est le planning [(0, 8); (11, 14); (17, 39); (40, 120)] i.e. lundi de 0h à 8h (exclus), lundi de 11h à 14h, lundi de 17h à mardi 15h, mardi 16h à vendredi minuit.

Il s'agit ici d'écrire une fonction qui calcule les trous de lundi 0h à vendredi minuit, soit la date 120. On va d'abord écrire une fonction qui calcule le complément d'un planning à partir de la date h.

```
let bornesup = (*vendredi 24h*) date_of_jour_heure (Vendredi, 24);;

let rec compl_aux h p = match p with
| [] -> if h >= bornesup then [] else [(h, bornesup)]
| (d,f)::l -> if h < d then (h, d)::(compl_aux f l) else (compl_aux f l);;

let compl p = compl_aux 0 p;;
```

Question 23

Écrire une fonction `possibilites_debut` de type `int -> planning -> planning` telle que `possibilites_debut m p` renvoie le planning canonique dont les créneaux représentent les possibilités pour placer le début d'un cours de m heures sur le planning p `(supposé dans sa forme canonique)` sans que le cours chevauche un créneau existant. On supposera pour simplifier qu'il n'est pas possible de placer un cours à cheval sur deux semaines. La fonction retournera la liste vide s'il n'y a pas de solution. Par exemple avec le planning `[(8, 10); (9, 11); (14, 16); (39, 40)]`, pour un cours de 3h, la fonction doit calculer le planning `[(0, 5); (11, 11); (16, 36); (40, 116)]` : en effet on peut placer le début entre les dates 0 et 5 (le cours finira avant la date 8 où la salle est prise), on peut aussi placer le cours à la date 11 (le cours finira à la date 14), ensuite on peut placer le début du cours entre les dates 16 et 36, et les dates 40 et 117. En revanche avec la planning (du lundi 1h à vendredi 22h) `[(1,118]`, la fonction retourne la liste vide (avec 118=`date_of_jour_heure (Vendredi, 22)`). On ne peut en effet y placer un cours de 3h. Indication : utiliser la fonction `compl` de la question précédente.

```
let possibilites_debut m p =
let rec aux cp = match cp with
| [] -> []
| (d,f)::l -> let l' = aux l in
                  let nouvfin = f-m in if nouvfin >= d (* le egal est important *) then
                  (d,nouvfin)::l' else l'
in aux (compl p);;

# pl_ex;;
- : (int * int) list = [(8, 10); (9, 11); (14, 16); (39, 40)]

# possibilites_debut 3 pl_ex;;
- : (int * int) list = [(0, 5); (11, 11); (16, 36); (40, 117)]
```

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);;
```
