Nom :

Prénom :

RICM3 - 2015/2016

Algorithmique et Programmation Fonctionnelle

EXAMEN

Durée : 2h, le seul document autorisé est une feuille A4 recto-verso manuscrite de notes personnelles

---

Cet énoncé comporte des parties indépendantes. Le barème est indicatif, le nombre de points correspond au nombre de minutes estimé nécessaire pour faire les exercices. Le total des points est de 120 points (+10 points de bonus).

Exercices

Préliminaires

Quelques définitions pour rappel.
Un **prédicat sur** T, où T est un type, est une fonction de T vers *bool* ;
on dit qu'une valeur x de type T **satisfait** un prédicat p si $p\,x$ renvoie `true` ;
un **prédicat** est un prédicat sur un type T non précisé, c'est-à-dire une une fonction à un argument vers *bool*.

On rappelle également la spécification de quelques combinateurs de listes prédéfinis en OCaml :

- `for_all : ('a -> bool) -> 'a list -> bool`
  où `for_all p l` vérifie si tous les éléments de `l` satisfont le prédicat `p`.
- `map : ('a -> 'b) -> 'a list -> 'b list`
  est telle que `map f [a1; ...; an]` construit et renvoie la liste `[f a1; ...; f an]`.
- `filter : ('a -> bool) -> 'a list -> 'a list`
  où `filter p l` renvoie tous les éléments de `l` qui satisfont le prédicat `p` dans le même ordre.
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
  est telle que `fold_left f a [b1; ...; bn]` renvoie `f (... (f (f a b1) b2) ...) bn`.

Exercice 1 : Fonctions d'ordre supérieur (20 points)

1.1. Écrire une fonction `loop : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a` telle que `loop p f x` applique la fonction `f` de façon répétée à l'argument `x` tant que la valeur obtenue ne vérifie pas le prédicat `p`.

Par exemple,
`loop (fun x -> x > 5) (fun x -> x + 3) 1` renvoie 7,
`loop (fun x -> x > 5) (fun x -> x + 3) 6` renvoie 6.

1.2. Utiliser `loop` pour définir une fonction `sqrt` qui calcule la racine carrée (arrondie à l'entier inférieur) d'un entier positif donné en argument.

1.3. On rappelle la définition du type option :

```
type 'a option = None | Some of 'a
```

Écrire une fonction `find : ('a -> bool) -> 'a list -> 'a option` telle que `find p l` renvoie `Some(x)` où `x` est le premier élément de `l` qui satisfait le prédicat `p`, ou bien `None` si aucun élément de `l` ne satisfait `p`.

1.4. À l'aide d'un combinateur de liste d'ordre supérieur, définir une fonction `emballer : 'a list -> 'a list list` qui transforme une liste d'éléments en une liste de listes à un seul élément chacune de sorte que, par exemple, `emballer [1;2;3] = [[1];[2];[3]]`.

Exercice 2 : Split et combine (30 points)

Soient les fonctions fst, snd, $split$, $combine$ et len définies par :

```
let fst (x, y) = x
let snd (x, y) = y
let rec split l = match l with
  | [] → ([], [])
  | (x, y) :: l → ((x :: fst(split l)), (y :: snd(split l)))
let rec combine (l1, l2) = match (l1, l2) with
  | ([], []) → []
  | (t1 :: q1, t2 :: q2) → (t1, t2) :: (combine (q1, q2))
let rec len l = match l with
  | [] → 0
  | t :: q → 1 + len q
```

2.1. Démontrez que pour tout couple c, on a $(fst\ c, snd\ c) = c$.

2.2. Démontrez que pour toute liste l de couples, $combine(split\ l) = l$.

2.3. Donner les types des fonctions `split` et `combine`.

2.4. Soit la fonction `map2` telle que
`map2 f [a1; ...; an] [b1; ...; bn]` renvoie `[f a1 b1; ...; f an bn]`.
Donner le type de `map2` et l'implémenter en utilisant `failwith` dans le cas où les deux listes données n'ont pas la même longueur.

2.5. Réécrire la fonction `combine` à l'aide de `map2`.

Exercice 3 : Module/foncteur (50 points)

Le module `MULTIENS`, dont la signature est donnée ci-dessous, spécifie une structure de données, que l'on appelera **multi-ensemble**. Il permet de stocker des **éléments** de type quelconque (`'a` en OCaml) en tenant compte du nombre d'occurrences de chaque élément dans la structure mais pas de leur ordre.
Pour chaque élément x, un multi-ensemble peut donc : ne pas le contenir, en contenir un seul ou plusieurs exemplaires. La signature de ce foncteur contient :

- le type `'a mens`, représentant un multi-ensemble ;
- `mensVide` est un multi-ensemble vide (ne contenant aucun élément) ;
- `estVide m` vaut vrai si et seulement si `m` est un multi-ensemble vide ;
- `inserer e m` renvoie un multi-ensemble similaire à `m` mais auquel on a ajouté un exemplaire de l'élément `e` ;
- `nbOcc e m` renvoie le nombre (éventuellement nul) d'occurrences de `e` dans `m`
- `choisir m` renvoie, pour un multi-ensemble non vide, un couple formé d'un élément `e` pris dans `m` (n'importe lequel) et du multi-ensemble des éléments restants une fois `e` enlevé.

```
module type MULTIENS =
sig
  type α mens
  val mensVide : α mens
  val estVide : α mens → bool
  val inserer : α → α mens → α mens
  val nbOcc : α → α mens → int
  val choisir : α mens → (α × α mens)
end
```

3.1. Donner une première implémentation de la signature `MULTIENS` dans laquelle le type `'a mens` est implémenté par une liste d'éléments.

Exemple : Si `m` est un multi-ensemble contenant 2 exemplaires des éléments `"a"` et `"b"`, et 3 exemplaires de l'élément `"d"`, alors `m` peut être représenté par la liste `["a"; "a"; "b"; "b"; "d"; "d"; "d"]`, mais aussi par `["d"; "b"; "b"; "a"; "d"; "a"; "d"]` ou toute autre permutation de cette liste.

3.2. Donner une deuxième implémentation de la signature `MULTIENS` dans laquelle le type `'a mens` est implémentée par une liste de couples (élément e, nombre d'occurrences de e).
De plus, on impose que, à tout moment, la liste ne contienne :

— aucun couple de la forme $(e, 0)$;

— jamais deux couples (e, i) et (e, j) pour le même élément e (qui pourraient être remplacés par un unique couple $(e, i + j)$).

Exemple : Dans cette implémentation, le multi-ensemble `m` est représenté par la liste `[("a", 2); ("d", 3); ("b", 2)]` ou toute autre permutation de cette liste. La liste ne doit pas contenir le couple `("c", 0)`.

À partir de cette question, on se place du côté « utilisateur » du module. Toutes les fonctions devront donc être définies en utilisant uniquement les fonctions fournies dans la signature MULTIENS, indépendemment de la façon dont celles-ci sont implémentées.

3.3. Écrire une fonction `union` qui prend en arguments deux multi-ensembles et renvoie leur union.
Ainsi, si un élément e est présent en i exemplaires dans `m1` et en j exemplaires dans `m2`, il sera présent en $i + j$ exemplaires dans `union m1 m2`.

3.4. Écrire une fonction `supprimer` qui prend en arguments un élément et un multi-ensemble et renvoie le multi-ensemble privé de **toutes** ses occurrences de cet élément.

3.5. Écrire une fonction `inclus` qui prend en arguments deux multi-ensembles et vérifie si le premier est inclus dans le second, autrement dit si chaque élément présent dans `m1` est présent en au moins autant d'exemplaires dans `m2`.

3.6. En déduire une fonction `comparer` qui prend en arguments deux multi-ensembles et renvoie une des valeurs du type suivant :

type *ordre* = *Egal* | *InfEgal* | *SupEgal* | *Incomparable*

Exercice 4 : Analyse syntaxique/flot (20 points)

4.1. On considère un type de chemins défini comme suit :

type *direction* = *Gauche* | *Droite*
type *chemin* = *direction list*
Écrire deux analyseurs qui prennent en entrée un flot de caractères `g`, `d` ou `f` et qui produisent respectivement :

- un `chemin` tel que défini plus haut, obtenu en consommant une suite de `g` et de `d` terminée par un `f` ;
- la liste des chemins représentés par le flot de caractères

Par exemple, un flot contenant les caractères `ggfgdfdf` donnera la liste de chemins `[[Gauche; Gauche]; [Gauche; Droite]; [Droite]]`.

4.2. On considère également un type d'arbre de mots défini comme suit :

type *arbmots* = *Mot* of *string* | *Choix* of *arbmots* × *string* × *arbmots*
On va maintenant lire un arbre de mots dans un flot de caractères.

La syntaxe est la suivante :
- un mot est une suite de lettres en minuscules
- les nœuds `Choix` sont représentés sous la forme `(a1{mot}a2)` où *a1* et *a2* sont les fils gauche et droit de ce nœud, et *mot* est le mot (éventuellement vide) stocké à ce nœud.

Ainsi, l'arbre

```
let exarbmots = Choix (Choix (Mot "an", "a", Mot "au"),
                       "",
                       Choix (Mot "fa", "", Mot "fi"))
```

est représenté par le flot suivant :

```
let flot_arbmots = Stream.of_string "((an{a}au){}(fa{}fi))"
```

On vous donne un analyseur permettant de lire une suite de lettres :

```
let rec p_mot m = parser
                | [< 'a'..'z' as c ; m2 = p_mot (m^(String.make 1 c)) >] -> m2
                | [< >] -> m
```

Vous noterez que cet analyseur prend un argument `m`, qui contient la suite des lettres déjà lues.

Écrire un analyseur qui prend en entrée un flot de caractères respectant la syntaxe donnée et produit un *arbmots* correspondant.