NOM : **RICM3 - 2016/2017**

PRÉNOM : **Algorithmique Programmation Fonctionnelle**

Examen

Durée : 2h

Le seul document autorisé est une feuille A4 recto-verso manuscrite de notes personnelles.

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 + 20 points de bonus, vous êtes donc encouragés à bien traiter seulement une partie des questions, plutôt qu'à vouloir tout traiter trop vite.**

La difficulté des questions est plutôt croissante tout au long du sujet, les parties les plus difficiles sont indiquées par des étoiles.

La plupart des questions sont indépendantes, et on pourra toujours supposer qu'une fonction demandée en exercice est disponible pour les questions suivantes.

---

1 Un module pour des multi-ensembles (55 points)

On souhaite dans cette partie être capable de représenter en Ocaml des *multi-ensembles*, qui sont des ensembles avec répétitions. par exemple, `me = {3,4,6,3,6,6}` est un multi-ensemble d'entiers contenant deux exemplaires de `3`, un exemplaire de `4` et trois exemplaires de `6`.

1.1 Signature (10 points)

On définit pour cela la signature minimale suivante :

```
module type Multens =
sig
  type α t
  val empty : unit → α t
  val add : α → α t → α t
end
```

où :
- le type α `t` est celui des multi-ensembles dont les éléments sont de type α ;
- la fonction `empty` permet de créer un nouveau multi-ensemble vide ;
- et la fonction `add` permet d'ajouter un élément à un multi-ensemble existant.

Compléter la signature précédente pour que les modules de signature `Multens` proposent également les opérations suivantes :

— `isempty` qui permet de savoir si un multi-ensemble donné est vide ;
— `nbocc` qui donne le nombre d'occurences (éventuellement nul) d'un élément dans un muti-ensemble ;
— `remove` qui enlève un exemplaire d'un élément dans un multi-ensemble.
— Vous définirez enfin une exception appropriée pour le cas où on cherche à enlever un élément absent du multi-ensemble, et vous utiliserez cette exception dans les implémentations des sections suivantes.

Corrigé

module type *Multens* =
sig
  type α *t*
  val *empty* : *unit* $\rightarrow$ α *t*
  val *add* : $\alpha \rightarrow \alpha\ t \rightarrow \alpha\ t$

  val *is_empty* : $\alpha\ t \rightarrow$ *bool*
  val *nbocc* : $\alpha \rightarrow \alpha\ t \rightarrow$ *int*
  val *rem* : $\alpha \rightarrow \alpha\ t \rightarrow \alpha\ t$
  exception *Element_absent*
end

1.2 Une première représentation simple (10 points)

On choisit dans un premier temps de représenter un multi-ensemble comme une simple liste Ocaml, dans laquelle on ne tiendra pas compte de l'ordre des éléments.

Ainsi, le multi-ensemble `me` donné en exemple au début de cet exercice pourra être représenté indifférement par les listes :
— `let me = [ 3 ; 3 ; 4 ; 6 ; 6 ; 6 ]`,
— `let me = [ 6 ; 6 ; 3 ; 4 ; 3 ; 6 ]`,
— et bien d'autres encore.

Implémenter complètement un module respectant la signature que vous avez proposée, en se basant sur cette représentation :

Corrigé

module *DupMultens* : *Multens* =
struct
  type α *t* = α *list*
  let *empty* () = [ ]
  let *add* x e = $x :: e$

  let *is_empty* e = $(e = [\,])$
  let rec *nbocc* x e = match e with
      [ ] $\rightarrow$ 0
    | $y :: e'$ $\rightarrow$ (if $x = y$ then 1 else 0) + *nbocc* x e'
  exception *Element_absent*
  let rec *rem* x e = match e with
      [ ] $\rightarrow$ *raise Element_absent*
    | $y :: e'$ $\rightarrow$ if $x = y$ then e' else $y ::$ (*rem* x e')
end

1.3 Une représentation plus compacte (15 points)

Afin de gagner en espace mémoire et en efficacité de traitement, on propose maintenant de représenter un multi-ensemble d'éléments (de type α) comme une liste **de taille minimale** de couples `(x, n)`, où `x` est un élément du multi-ensemble et `n` son nombre **total** d'occurrences dans ce multi-ensemble.

Le critère de minimalité implique notamment qu'aucun couple de la liste ne peut être de la forme `(x, 0)`.

Voici deux (parmi d'autres...) implémentations possibles du multi-ensemble `me` donné en exemple plus haut :

— `let me = [ (3,2) ; (4,1) ; (6,3) ]`,
— `let me = [ (6,3) ; (3,2) ; (4,1) ]`.

Notons que la liste `[(6,1) ; (3, 2) ; (6,2) ; (4,1)]` n'est pas une implémentation valide : cette liste n'est pas minimale car l'élément `6` y apparait deux fois.

Implémenter complètement un module respectant la signature que vous avez proposée, en se basant sur cette nouvelle représentation :

Corrigé

module $CoupMultens$: $Multens$ =
struct
  type $\alpha\ t = (\alpha \times int)\ list$
  let $empty\ () = [\,]$
  let rec $add\ x\ e$ = match e with
      $[\,] \rightarrow [(x,1)]$
    | $(y,n) :: e' \rightarrow$ if $x = y$ then $(x, n+1) :: e'$ else $(y,n) :: (add\ x\ e')$

  let $is\_empty\ e = (e = [\,])$
  let rec $nbocc\ x\ e$ = match e with
      $[\,] \rightarrow 0$
    | $(y,n) :: e' \rightarrow$ if $x = y$ then n else $nbocc\ x\ e'$
  exception $Element\_absent$
  let rec $rem\ x\ e$ = match e with
      $[\,] \rightarrow$ $raise\ Element\_absent$
    | $(y,n) :: e'$ when $x = y \rightarrow$ if $n = 1$ then e' else $(x, n-1) :: e'$
    | $(y,n) :: e' \rightarrow (y,n) :: (rem\ x\ e')$
end

1.4 Utilisation externe du module (20 points)

Soit un module `M` de signature `Multens` mais dont l'implémentation concrète vous est inconnue.

Implémenter les fonctions suivantes :

1. `present : 'a -> 'a M.t -> bool` qui détermine si un élément est présent (au moins une fois) dans un multi-ensemble.

   Corrigé

   let $present\ x\ e = M.nbocc\ x\ e > 0$

2. `add_n : int -> 'a -> 'a M.t -> bool` qui ajoute plusieurs copies d'un même élément à un multi-ensemble.

   Corrigé

   let rec $add\_n\ n\ x\ e$ = if $n = 0$ then e else $M.add\ x\ (add\_n\ (n-1)\ x\ e)$

3. `remove_all : 'a -> 'a M.t -> 'a M.t` qui supprime **toutes** les occurrences d'un élément dans un multi-ensemble.
   Indication : Une méthode élégante pour implémenter cette fonction consiste à récupérer l'exception levée par la fonction `remove`.

Corrigé

let rec $remove\_all\ x\ e$ = try let e' = $M.rem\ x\ e$ in $rem\_all\ x\ e'$ with
  | $M.Element\_absent \rightarrow e$

2 Arbres et expressions (85 points)

Dans cette partie, on va considérer des expressions arithmétiques simples et leur représentation par différents types d'arbres.

2.1 Des opérations comme constructeurs (10 points)

Le premier type d'arbres considéré est le suivant.

`Add` correspond à l'addition de deux sous-expressions et `Prod` correspond au produit de deux sous-expressions. Une expression comme `((2 * 3) + 1)` est représentée par l'arbre `Add (Prod (Const (2), Const (3)), Const (1))`.

type $expr$ =
  | $Const$ of int
  | Add of $expr \times expr$
  | $Prod$ of $expr \times expr$

Écrire une fonction `eval_expr` de type `expr` → `int`, telle que `eval_expr a` calcule la valeur entière de l'expression représentée par l'arbre `a`.

Corrigé

let rec $eval\_expr\ e$ = match e with
  | $Const\ (n) \rightarrow n$
  | $Add\ (e1,\ e2) \rightarrow eval\_expr\ e1 + eval\_expr\ e2$
  | $Prod\ (e1,\ e2) \rightarrow eval\_expr\ e1 \times eval\_expr\ e2$

Proposer un type `exprsous` pouvant comporter non seulement l'addition et et la multiplication, mais aussi la soustraction.

Corrigé

```
type exprsous =
  | Const of int
  | Add of exprsous × exprsous
  | Soustr of exprsous × exprsous
  | Prod of exprsous × exprsous
```

2.2 Arbres paramétrés par des opérations (15 points)

Dans la suite de cette partie on va étendre les expressions autorisées de façon différente. On se donne tout d'abord un type général d'arbres binaires, dont les feuilles sont étiquettées par des entiers et dont les nœuds sont étiquettés par un type donné en paramètre.

```
type α arbre_gen = F of int | Op of α arbre_gen × α × α arbre_gen
```

Considérons par exemple un type énuméré permettant de choisir entre deux opérations binaires, l'addition et le produit.

```
type op_ap = Plus | Mult
```

On peut donner une représentation équivalente au type `expr` au moyen du type suivant.

```
type arbre_ap = op_ap arbre_gen
```

L'expression `((2 * 3) + 1)` serait ici représentée par l'arbre
`Op (Op (Const (2), Mult, Const (3)), Plus, Const (1))`.

Écrire une fonction `expr_arbre` traduisant fidèlement un arbre de type `expr` vers un arbre de type `arbre_ap`

Corrigé

```
let rec expr_arbre e = match e with
  | Const (n) → F n
  | Add (e1, e2) → Op (expr_arbre e1, Plus, expr_arbre e2)
  | Prod (e1, e2) → Op (expr_arbre e1, Mult, expr_arbre e2)
```

Écrire une fonction `arbre_expr` traduisant fidèlement un arbre de type `arbre_ap` vers un arbre de type `expr`. Les fonctions `expr_arbre` et `arbre_expr` sont inverses l'une de l'autre.

Corrigé

```
let rec arbre_expr a = match a with
  | F (n) -> Const n
  | Op (a1, Plus, a2) -> Add (arbre_expr a1, arbre_expr a2)
  | Op (a1, Mult, a2) -> Prod (arbre_expr a1, arbre_expr a2)
```

Donner un type énuméré `op_asp` permettant de choisir entre l'addition la soustraction et le produit. Donner un type d'arbres binaires pour ces trois opérations, utilisant le type α `arbre_gen`.

Corrigé

```
type op_asp = Plus | Moins | Mult
type arbre_asp = op_asp arbre_gen
```

2.3 Arbres paramétrés à l'ordre supérieur * (20 points)

Dans le type α `arbre_gen`, on peut instancier le paramètre α par autre chose qu'un type énuméré, de façon à étiquetter les nœuds binaires non pas par des noms d'opérations (des constructeurs comme `Plus`), mais par les fonctions elles-mêmes, comme `fun x y -> x+y`.

On définit la fonction `trad` comme suit.

```
let trad = function
  | Plus -> fun x y -> x + y
  | Mult -> fun x y -> x × y
```

Donner le type de cette fonction.

Corrigé

```
type type_de_trad = op_ap -> (int -> int -> int)
```

En utilisant `trad`, écrire une fonction `fonctise` qui rend un arbre de fonctions de type `(int -> int -> int) arbre_gen` à partir d'un arbre de type `op_ap arbre_gen`.

** *Un bonus sera accordé aux réponses qui définissent préalablement une fonction d'ordre supérieur* `map` *de type* $(\alpha \to \beta) \to (\alpha$ `arbre_gen` $\to \beta$ `arbre_gen`$)$.

Corrigé

```
let rec fonctise a = match a with
  | F (x) -> F (x)
  | Op (g, f, d) -> Op (fonctise g, trad f, fonctise d)
```

Écrire une fonction `eval` de type `(int -> int -> int) arbre_gen -> int`, telle que `eval a` calcule la valeur entière de l'expression représentée par l'arbre `a`. Il est demandé que `eval (fonctise (expr_arbre e))` rende le même résultat que `eval_expr e`.

Corrigé

```
let rec eval a = match a with
  | F (x) -> x
  | Op (g, f, d) -> f (eval g) (eval d)
```

2.4 Environnements (10 pts)

Cette section (facile) est indépendante des autres sections. Les résultats pourront être admis dans la suite. Un environnement permet d'associer des noms et des valeurs. Dans notre cas les noms seront des caractères et les valeurs des entiers.

```
type env = char -> int
```

Définir trois fonctions `env_init`, `affecter` et `rechercher` fournissant respectivement

- un environnement initial dans lequel, à votre guise, tous les noms sont associés à la valeur 0, ou bien (bonus) aucun nom n'est associé à une valeur ;
- un nouvel environnement est construit à partir d'un ancien environnement, un nom et sa valeur associée ;
- la valeur associée à un nom.

Corrigé

```
let env_init = fun _ -> 0
(* Ou bien *)
let env_init = fun _ -> failwith "yapa"

let rechercher k e = e k

let affecter k v e = fun h -> if h = k then v else (e h)
```

2.5 Arbres avec variables ** (10 pts)

On complète nos arbres tout d'abord en introduisant la possibilité qu'une feuille soit étiquetée par un caractère, représentant un nom de variable. L'évaluation d'un tel arbre doit alors se calculer en prenant en compte un environnement, comme défini dans la section précédente. Enfin, cet environnement pourra être modifié complétant une expression par un mécanisme analogue au `let` de Ocaml, liant un nom à la valeur d'une expression.

Le type des arbres est le suivant.

```
type α arbre_let =
   | Num of int
   | Var of char
   | Op of α arbre_let × α × α arbre_let
   | Let of char × α arbre_let × α arbre_let
```

L'évaluation de `Let('x'`$, a_1, a_2$`)` est similaire à l'évaluation de l'expression Ocaml
`let x =` a_1 `in` a_2.

Écrire une fonction qui évalue un tel arbre dans un environnement donné. Utiliser les fonctions définies dans la partie 1 de cette énoncé pour les calculs sur des tables d'association. On pourra prendre pour α soit `op_ap`, soit `int -> int -> int` (bonus das ce cas).

Corrigé

```
let rec eval_arbre_fct e a = match a with
   | Num (x) → x
   | Var (x) → rechercher x e
   | Op (g, f, d) → f (eval_arbre_fct e g) (eval_arbre_fct e d)
   | Let (v, a1, a2) →
       let e' = affecter v (eval_arbre_fct e a1) e in
       eval_arbre_fct e' a2
```

2.6 Analyse syntaxique * (20 points)

Écrire un analyseur de flots de caractères qui lit un chiffre (un caractère compris entre `'0'` et `'9'`) et rend l'entier correspondant. On pourra utiliser la fonction `Char.code` qui rend la valeur ascii d'un caractère.

Corrigé

```
let valchiffre c = Char.code c − Char.code '0'
let p_chiffre = parser
   | [⟨ ''0'..'9' as c ⟩] → valchiffre c
```

On se donne une grammaire pour représenter les expressions définies précédemment. Les opérations binaires sont complètement parenthésées, et une expression qui serait écrite en Ocaml
`let x =` a_1 `in` a_2
sera écrite ici avec des accolades, un signe `=` et un point :
`{x=`a_1`.`a_2`}`

Voici la grammaire.

```
C ::= 0..9
V ::= a..z
O ::= + | *
```

```
E ::= C | V | (E O E) | {x = E . E}
```

Écrire un analyseur de flots de caractères correspondant à cette grammaire et rendant à votre guise un arbre de type `op_ap arbre_gen` ou un arbre de type `(int -> int -> int) arbre_gen`.

Il est possible de répondre partiellement à cette question en ne considérant que les expressions correspondant aux arbres plus simples introduits à partir de 2.2.

Corrigé

let *p_op_asp* = parser
  | [< ''+' >] → *Plus*
  | [< ''-' >] → *Moins*
  | [< ''*' >] → *Mult*

Ou bien

let *p_op_fct* = parser
  | [< ''+' >] → (+)
  | [< ''-' >] → (−)
  | [< ''*' >] → fun *x* *y* → *x* × *y*

let rec *p_arbre_let* *p_op* = parser
  | [< *n* = *p_chiffre* >] → *Num* (*n*)
  | [< ''a'..'z' as *c* >] → *Var* (*c*)
  | [< ''('; *a1* = *p_arbre_let* *p_op*; *op* = *p_op*; *a2* = *p_arbre_let* *p_op*; '')' >] →
    Op (*a1*, *op*, *a2*)
  | [< ''{'; ''a'..'z' as *c*; ''=';
      a1 = *p_arbre_let* *p_op*; ''.'; *a2* = *p_arbre_let* *p_op*;''}' >] →
    Let (*c*, *a1*, *a2*)