NOM :

PRÉNOM :

RICM3 - 2014/2015

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* ;
un **prédicat** est un prédicat sur un type T non précisé, c'est-à-dire une une fonction à un argument vers *bool*.

On se donne la fonction suivante permettant de filtrer les éléments d'une liste l satisfaisant un critère de sélection p, où p est un prédicat.

```
let rec filter p l = match l with
  | [] → []
  | x :: l → if p x then x :: filter p l else filter p l
```

Cette fonction sera utilisée dans plusieurs des exercices à suivre.

Exercice 1 : Récursivité et listes (10 points)

Écrire les fonctions *decroiss*, *prod* et *fact* et donner leur type :
– *decroiss* n rend la liste $[n; ...; 1]$, *decroiss* 0 rend une liste vide,
– *prod* l rend le produit des éléments de la liste l,
– *fact* n rend la factorielle de n.
Il est demandé d'utiliser *decroiss* et *prod* pour programmer *fact*.

Corrigé

```
let rec decroiss n =
 if n = 0 then [] else n :: decroiss (n - 1)

let rec prod l = match l with
   | [] -> 1
   | n :: l -> n * prod l

let fact n = prod (decroiss n)
```

decroiss : int → int list
prod : int list → int
fact : int → int

Exercice 2 : Module/foncteur (30 points)

Le foncteur de type `MEMOIRE`, dont la signature est donnée ci-dessous, spécifie une structure de données, que l'on appelera une **mémoire**, permettant de mémoriser des **éléments** de type quelconque (`'a` en OCaml) à chacun desquels est associée une **clé** (de type `cle`). On suppose que la mémoire peut contenir plusieurs éléments identiques ayant même clé. La signature de ce foncteur contient :
- le type `'a mem`, représentant une mémoire ;
- la fonction `memVide` qui renvoie une mémoire vide (ne contenant aucun élément) ;
- la fonction `(estVide m)`, qui vaut vrai si et seulement si `m` est une memoire vide ;
- la fonction `(inserer m e c)` qui renvoie une mémoire `m` à laquelle l'élément `e` de clef `c` a été ajouté ;
- la fonction `(extraire m c)` qui renvoie la liste de tous les éléments de `m` dont la clé est « égale » à celle de `c` (selon la fonction `Cle.egal`).

Ce foncteur `MEMOIRE` est paramétré par un module de type `CLE` dont la signature contient :
- le type `cle` ;
- une fonction d'égalité entre clés, la fonction `egal`.

```
module type CLE =
sig
  type cle
  val egal : cle -> cle -> bool
end

module type MEMOIRE = functor (Cle : CLE) ->
sig
  type 'a mem
  val memVide : 'a mem
  val estVide : 'a mem -> bool
  val inserer : 'a mem -> 'a -> Cle.cle -> 'a mem
  val extraire : 'a mem -> Cle.cle -> 'a list
end
```

2.1. En complétant le code OCaml ci-dessous, donner une première implémentation du module `MEMOIRE` dans laquelle le type `'a mem` est implémenté par une liste de couples (élément, clé).

module $MEMOIRE\_1$: $MEMOIRE$ = functor (Cle : CLE) $\rightarrow$
struct
  type α mem = (α $\times$ $Cle.cle$) $list$
  let $memVide$ = ...
  let $estVide$ m = ...
  ...
end

Exemple : Si `m1` est une mémoire contenant les élément `"a"`, `"b"` et `"c"` de clés respectives `2`, `3` et `2` alors `m` sera représenté par la liste `[("a", 2); ("b", 3); ("c", 2)]`.
On pourra *si on le souhaite* utiliser (sans les écrire) les fonctions prédéfinies suivantes sur les listes OCaml – la première est rappelée dans les préliminaires en début de sujet :

```
val filter : ('a -> bool) -> 'a list -> 'a list
val split : ('a * 'b) list -> 'a list * 'b list
```

`(filter p l)` renvoie tous les éléments de la liste `l` satisfaisant le predicat `p` (cf. préliminaires). `split` transforme une liste de couples en un couple de listes : `split [(a1,b1); ...; (an,bn)]` renvoie `([a1; ...; an], [b1; ...; bn])`.

Corrigé

module $MEMOIRE\_1$: $MEMOIRE$ = functor (Cle : CLE) $\rightarrow$
struct
  type α mem = (α $\times$ $Cle.cle$) $list$
  let $memVide$ = [ ]
  let $estVide$ mem = (mem = [ ])
  let $inserer$ mem $elem$ cle = ($elem, cle$) :: mem
  let $extraire$ mem cle =
    fst ($List.split$ ($List.filter$ (fun x $\rightarrow$ let (e, c) = x in ($Cle.egal$ c cle)) mem))
end

2.2. En complétant le code OCaml ci-dessous, donner une deuxième implémentation du module `MEMOIRE` dans laquelle le type `'a mem` est implémentée par une liste de couples (clé c, liste d'éléments e ayant même clé c).

```
module MEMOIRE_2 : MEMOIRE = functor (Cle : CLE) →
struct
  type α mem = (Cle.cle × (α list)) list
  let memVide = ...
  let estVide m = ...
  ...
end
```

Exemple : Dans cette implémentation, si on suppose que les clés `2` et `3` sont différentes (selon la fonction `Cle.egal`), la mémoire `m1` contenant les éléments `"a"`, `"b"` et `"c"` de clés respectives `2`, `3` et `2` est représentée par la liste `[(2, ["a"; "c"]); (3, ["b"])]`.

Corrigé

```
module MEMOIRE_2 : MEMOIRE = functor (Cle : CLE) →
struct
  type α mem = (Cle.cle × (α list)) list
  let memVide = []
  let estVide mem = (mem = [])
  let rec inserer mem elem cle = match mem with
    | [] → [(cle, [elem])]
    | (c, l) :: f → if (Cle.egal c cle) then
          (c, elem :: l) :: f
        else
          (c, l) :: (inserer f elem cle)
  let extraire mem cle =
    List.concat
      (List.map snd
          (List.filter
              (fun x → let (c, l) = x in (Cle.egal c cle)) mem))
end
```

2.3. Donner une implémentation du module `CLE` dans laquelle :
– le type `cle` est le type `int`
– la fonction `(egal c1 c2)` vaut vrai ssi `c1` et `c2` ont même parité.

Corrigé

```
module Cle =
struct
  type cle = int
  let egal c1 c2 = (c1 mod 2 = c2 mod 2)
end
```

Exercice 3 : Preuve (45 points)

Cet exercice a pour objet de démontrer et utiliser des propriétés de la fonction *filter* rappelée dans les préliminaires en début du sujet.

3.1. Au moyen de cette fonction *filter* , donner une fonction *filtre_pairs* qui filtre les valeurs paires d'une liste d'entiers et une fonction *filtre_impairs* qui filtre les valeurs impaires d'une liste d'entiers.

Corrigé

Plusieurs solutions possibles.

let $pair\ n\ =\ n$ mod $2 = 0$
let $filtre\_pairs\ l\ =\ filter\ pair\ l$
let $filtre\_pairs\ =\ filter$ (fun $n\ \rightarrow\ n$ mod $2 = 0$)
let $filtre\_impairs\ l\ =\ filter$ (fun $n\ \rightarrow\ \neg\ (pair\ n)$) l

3.2. Soient deux critères de sélection respectivement représentés par les prédicats p et q sur le même type. Comment serait codé le prédicat correspondant à la conjonction de ces deux critères ?

Corrigé

fun $x \rightarrow p\ x$ && $q\ x$

3.3. Soient p et q deux prédicats équivalents (pour tout x, on a $p\ x = q\ x$). Démontrer que

$$\forall l,\ filter\ p\ l = filter\ q\ l$$

Corrigé

On raisonne par récurrence structurelle sur l.

– Pour $l = []$, $filter\ p\ [] = [] = filter\ q\ []$
– Pour $l = x :: u$, avec pour hypothèse de récurrence $filter\ p\ u = filter\ q\ u$, on a

$$\begin{aligned} filter\ p\ (x :: u) &= \textsf{if}\ p\ x\ \textsf{then}\ x :: filter\ p\ u\ \textsf{else}\ filter\ p\ u \\ &= \textsf{if}\ q\ x\ \textsf{then}\ x :: filter\ p\ u\ \textsf{else}\ filter\ p\ u \\ &= \textsf{if}\ q\ x\ \textsf{then}\ x :: filter\ q\ u\ \textsf{else}\ filter\ q\ u \\ &= filter\ q\ (x :: u) \end{aligned}$$

3.4. Voici deux versions du double filtrage, *filter2_a* et *filter2_b* :

let $\mathit{filter2\_a}\ p\ q\ l\ =\ \mathit{filter}\ p\ (\mathit{filter}\ q\ l)$
let $\mathit{filter2\_b}\ p\ q\ l\ =\ \mathit{filter}\ (\textsf{fun}\ x\ \rightarrow\ p\ x\ \&\&\ q\ x)\ l$
Quel est le type de ces deux fonctions ?
Quel est l'avantage de *filter2_b* par rapport à *filter2_a* ?

Corrigé

```
('a -> bool) -> ('a -> bool) -> 'a list -> 'a list
```

filter2_b est en une seule passe, donc plus efficace.

Démontrer que *filter2_a* et *filter2_b* sont équivalents, au sens où ils calculent toujours le même résultat :

$$\forall\, p\, q,\ \forall\, l,\ \mathit{filter2\_a}\ p\ q\ l = \mathit{filter2\_b}\ p\ q\ l$$

On pourra se contenter d'exposer l'idée de la démonstration sans entrer dans tous les détails.

Corrigé

Soient p et q 2 prédicats arbitraires, on montre
$\forall\, l, \mathit{filter}\ p\ (\mathit{filter}\ q\ l) = \mathit{filter}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ l$ par récurrence structurelle sur l.

- Pour $l = []$, $\mathit{filter}\ p\ (\mathit{filter}\ q\ []) = \mathit{filter}\ p\ [] = [] = \mathit{filter}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ []$
- Pour $l = x :: u$, avec pour hypothèse de récurrence $\mathit{filter}\ p\ (\mathit{filter}\ q\ u) = \mathit{filter}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ u$, on considère les 4 cas où $p\,x$ et $q\,x$ valent true ou false. Par exemple, si tous les deux valent true, on a

$$\begin{aligned}
\mathit{filter}\quad p\quad & (\mathit{filter}\ q\ (x :: u)) \\
&= \mathit{filter}\ p\ (\textsf{if}\ q\,x\ \textsf{then}\ x :: \mathit{filter}\ q\ u\ \textsf{else}\ \mathit{filter}\ q\ u) \\
&= \mathit{filter}\ p\ (x :: \mathit{filter}\ q\ u) \\
&= \textsf{if}\ p\,x\ \textsf{then}\ x :: \mathit{filter}\ p\ (\mathit{filter}\ q\ u)\ \textsf{else}\ \mathit{filter}\ p\ (\mathit{filter}\ q\ u) \\
&= x :: \mathit{filter}\ p\ (\mathit{filter}\ q\ u) \\
&= x :: \mathit{filter}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ u \\
&= \textsf{if}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ x\ \textsf{then}\ x :: \mathit{filter}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ u \\
&\quad\ \textsf{else}\ \mathit{filter}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ u \\
&= \mathit{filter}\ (\textsf{fun}\ x \rightarrow p\,x\ \&\&\ q\,x)\ (x :: u)
\end{aligned}$$

  Les 3 autres cas sont similaires.

3.5. En utilisant les deux questions précédentes, démontrer

$$\forall\, l,\ \mathit{filtre\_pairs}\ (\mathit{filtre\_impairs}\ l) = []$$

et

$$\forall\, l,\ \mathit{filtre\_pairs}\ (\mathit{filtre\_pairs}\ l) = \mathit{filtre\_pairs}\ l$$

Corrigé
En utilisant la question précédente, il suffit de démontrer respectivement

$$\forall\, l,\ \mathit{filter}(\mathsf{fun}\ n \rightarrow \mathit{pair}\, n\ \&\&\ \mathit{impair}\, n)\ l = []$$

et

$$\forall\, l,\ \mathit{filter}(\mathsf{fun}\ n \rightarrow \mathit{pair}\, n\ \&\&\ \mathit{pair}\, n)\ l = \mathit{filter}\ \mathit{pair}\ l$$

On utilise la question 3.3 en remarquant respectivement que le prédicat
$\mathsf{fun}\ n \rightarrow \mathit{pair}\, n\ \&\&\ \mathit{impair}\, n$ est équivalent à $\mathsf{fun}\ n \rightarrow false$, d'autre part que le prédicat
$\mathsf{fun}\ n \rightarrow p\, n\ \&\&\ p\, n$ est équivalent à p, en prenant ici $p = \mathit{pair}$.

Exercice 4 : Analyse syntaxique/flot (25 points)

C'est la distribution des cadeaux par le père ONoël. Cette année il a appris la récursivité et a décidé de festoyer avec ses nouvelles connaissances. Sa hotte est une boîte, sachant qu'une boîte contient pêle-mêle des cadeaux et (récursivement) des boîtes. Il y a des boîtes roses (à ouvrir par les filles) et des boîtes bleues (à ouvrir par les garçons). Il peut y avoir des boîtes bleues dans les roses et réciproquement, ce qui rend la distribution plus amusante.
Les types de données utilisés pour représenter tout cela sont les suivants.

```
type cadeau = Chocolats | Joujou | Livre
type boite =
  | Cadeau of cadeau
  | Rose of boite list
  | Bleu of boite list
```

4.1. On rappelle une version du programme *fold* sur les listes.

```
let rec fold f l a = match l with
  | [] -> a
  | b :: l -> f b (fold f l a)
```

Donner le type de cette fonction.

Corrigé
('a → 'b → 'b) → 'a list → 'b → 'b

4.2. Écrire une fonction qui calcule le nombre total de livres dans une boîte. Indication : deux solutions (au moins) sont possibles, l'une avec *fold*, l'autre au moyen de deux fonctions mutuellement récursives (rappel de leur syntaxe : let rec $f_1\, x \ldots = \ldots$ and $f_2\, y \ldots = \ldots$).

Corrigé

let rec *nblivres* b = match b with
  | *Cadeau* (*Livre*) $\rightarrow$ 1
  | *Cadeau* (_) $\rightarrow$ 0
  | *Rose* (l) $\rightarrow$ *fold* (fun b y $\rightarrow$ *nblivres* b + y) l 0
  | *Bleu* (l) $\rightarrow$ *fold* (fun b y $\rightarrow$ *nblivres* b + y) l 0

let rec *nblivres* b = match b with
  | *Cadeau* (*Livre*) $\rightarrow$ 1
  | *Cadeau* (_) $\rightarrow$ 0
  | *Rose* (l) $\rightarrow$ *nblis* l
  | *Bleu* (l) $\rightarrow$ *nblis* l
and *nblis* l = match l with
  | [] $\rightarrow$ 0
  | b :: l $\rightarrow$ *nblivres* b + *nblis* l

4.3. Pour décrire une boîte on utilise une syntaxe utilisant des lettres (c pour les chocolats, j pour un joujou, l pour un livre), des parenthèses pour les boîtes roses et des crochets pour les boîtes bleues.
La grammaire correspondante est :

$$
\begin{array}{rcl}
C & ::= & c \mid j \mid l \\
B & ::= & C \quad \mid \quad (\; L \;) \quad \mid \quad [\; L \;] \\
L & ::= & \varepsilon \quad \mid \quad B \; ; L
\end{array}
$$

Écrire un analyseur qui prend en entrée un flot de caractères respectant cette syntaxe et produit la donnée de type *boite* correspondante.

Corrigé

let p_cadeau = parser
  | [⟨ ''c' ⟩] → *Chocolats*
  | [⟨ ''j' ⟩] → *Joujou*
  | [⟨ ''l' ⟩] → *Livre*

let rec p_boite = parser
  | [⟨ c = p_cadeau ⟩] → *Cadeau c*
  | [⟨ ''(' ; l = p_liste; '')' ⟩] → *Rose l*
  | [⟨ ''[' ; l = p_liste; '']' ⟩] → *Bleu l*
and p_liste = parser
  | [⟨ b = p_boite; l = p_liste ⟩] → b :: l
  | [⟨ ⟩] → [ ]

Exercice 5 : Lambda-calcul (20 points)

On nomme I et S deux λ-termes particuliers :

$$\begin{array}{rcl} I & = & \lambda x.x \\ S & = & \lambda xyz.x\ z\ (y\ z) \end{array}$$

5.1. Effectuer la β-réduction des termes suivants :

– $I\ y$

Corrigé

$$\begin{array}{rcl} I\ y & = & (\lambda x.x)\ y \\ & \rightarrow & y \end{array}$$

– $I\ I$

Corrigé

$$\begin{array}{rcl} I\ I & = & (\lambda x.x)\ (\lambda x.x) \\ & =_\alpha & (\lambda y.y)\ (\lambda x.x) \\ & \rightarrow & (\lambda x.x) \end{array}$$

– $S\ I\ I$

Corrigé

$$\begin{array}{rcl} S\ I\ I & = & (\lambda xyz.x\ z\ (y\ z))\ (\lambda x.x)\ (\lambda x.x) \\ & =_\alpha & (\lambda xyz.x\ z\ (y\ z))\ (\lambda t.t)\ (\lambda u.u) \\ & \to & (\lambda yz.(\lambda t.t)\ z\ (y\ z))\ (\lambda u.u) \\ & \to & (\lambda yz.z\ (y\ z))\ (\lambda u.u) \\ & \to & \lambda z.z\ ((\lambda u.u)\ z) \\ & \to & \lambda z.z\ z \end{array}$$

5.2. Que peut-on dire de la β-réduction du terme $(S\ I\ I)\ (S\ I\ I)$?

Corrigé

D'après la question précédente $S\ I\ I \to^* \lambda z.z\ z$.

Par conséquent :

$$\begin{array}{rcl} (S\ I\ I)\ (S\ I\ I) & \to^* & (\lambda z.z\ z)\ (\lambda z.z\ z) \\ & =_\alpha & (\lambda x.x\ x)\ (\lambda z.z\ z) \\ & \to & (\lambda z.z\ z)\ (\lambda z.z\ z) \\ & = & (S\ I\ I)\ (S\ I\ I) \end{array}$$

Ce terme se β-réduit en lui-même indéfiniment.