Nom :

Prénom :

RICM3 - 2015/2016

Algorithmique Programmation Fonctionnelle

Devoir Surveillé

Durée : 1h30

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

Le barème est indicatif. Sachant que le sujet comporte des parties indépendantes, pensez à traiter des questions variées. On pourra toujours supposer qu'une fonction demandée en exercice est disponible pour les questions suivantes.

Exercice 1 : Évaluation (5 points)

1. Dans chacun des trois cas suivants, dire ce que vaut y après avoir évalué les définitions données :

(a)
```
let x = 5
let y = x + 3
let x = 7
```

```
8
```

(b)
```
let a = 5
let f = fun x -> x + a
let a = 10
let y = f a
```

```
15
```

(c)
```
let a = 5
let f = fun x -> x + a
let a = a + 1
let y = f a
```

```
11
```

2. Simplifier l'expression suivante :

```
let y = let x = x + 1 in x
```

```
let y = x + 1
```

3. Indiquer si l'expression suivante est correcte et si oui, la simplifier :

```
let y = let x = x > 1 in x
```

```
let y = x > 1
```

4. Indiquer si l'expression suivante est correcte et si oui, la simplifier :

```
let y = let x = x = 1 in x
```

```
let y = x = 1
```

5. Pour chacune des trois expressions précédentes qui est correcte, donner la valeur obtenue pour y dans un environnement où x est lié à 100.

```
Respectivement : 101, true, false
```

Exercice 2 : GPS (9 points)

On représente en OCaml un système de GPS simplifié pour lequel on va coder quelques fonctions utiles. Pour cela, on se base sur le type suivant :

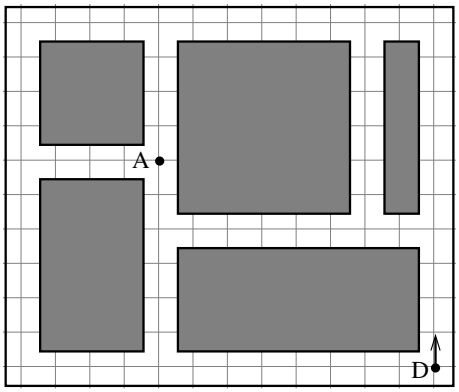
```
type mouvement = Avancer of float | Gauche | Droite | Demitour
```

dans lequel le paramètre flottant du constructeur *Avancer* désigne la distance à parcourir (en kilomètres). On imagine que toutes les rues se croisent à angle droit, et donc que les mouvements *Gauche* et *Droite* désignent des virages à 90°(des quarts de tours).

1. À l'aide de ce type, définissez un type *trajet* permettant de représenter un trajet complet dans notre GPS.

```
type trajet = mouvement list
```

2. Donner un exemple d'un tel trajet sous forme d'un terme OCaml, permettant d'aller du point D au point A sur le plan ci-contre.
La direction initiale de la voiture est indiquée par la flèche, et chaque case de la grille mesure 1 km de côté.



[Avancer(4.) ; Gauche ; Avancer(8.) ; Droite ; Avancer(2.)]

3. Écrire une fonction *longueur* qui calcule la longueur totale d'un trajet (les virages et demi-tours n'ont pas de longueur bien sûr).

Corrigé

```
let rec longueur t = match t with
  | [] -> 0.
  | Avancer(d) :: q -> d +. longueur q
  | _ :: q -> longueur q
```

4. Écrire une fonction *retour* qui calcule le trajet retour d'un trajet donné. On suppose qu'il n'y a pas de sens unique, donc qu'on peut suivre le même chemin en sens inverse.

Corrigé

```
let rec retour t = match t with
  | [] -> []
  | Gauche :: q -> (retour q)@[Droite]
  | Droite :: q -> (retour q)@[Gauche]
  | Demitour :: q -> (retour q)@[Demitour]
  | Avancer(d) :: q -> (retour q)@[Avancer(d)]
```

5. Pour l'une des deux questions précédentes au choix, écrire une version plus efficace de votre fonction à l'aide d'une fonction auxiliaire qui prendra un paramètre supplémentaire.
Si vous aviez déjà fait ce travail spontanément pour les deux questions qui précèdent... bravo ! Vous pouvez passer à la question suivante, ou écrire ici une version plus "naïve" de la fonction *retour*.

Corrigé

```
let retour_tl t =
  let rec raux t r = match t with
    | [] → r
    | Gauche :: q → raux q (Droite :: r)
    | Droite :: q → raux q (Gauche :: r)
    | Demitour :: q → raux q (Demitour :: r)
    | Avancer(d) :: q → raux q (Avancer(d) :: r)
  in raux t []
```

6. Notre GPS n'est pas optimisé et il ne se rend pas compte que parfois deux mouvements successifs peuvent être réécrits pour prendre moins de place. Par exemple, la suite de mouvements [*Gauche* ; *Droite* ; *Avancer*(5.)] est équivalente à [*Avancer*(5.)].
   Écrire une fonction *optimiser* qui prend un trajet en argument et renvoie un trajet équivalent écrit avec le moins de mouvements possible.

Corrigé

```
let rec optimiser t = match t with
  | [] → []
  | [_] → t
  | e :: f :: q → match e :: (optimiser (f :: q)) with
    | [_] → [e]
    | Gauche :: Gauche :: q → optimiser (Demitour :: q)
    | Gauche :: Droite :: q → optimiser q
    | Gauche :: Demitour :: q → optimiser (Droite :: q)
    | Droite :: Droite :: q → optimiser (Demitour :: q)
    | Droite :: Gauche :: q → optimiser q
    | Droite :: Demitour :: q → optimiser (Gauche :: q)
    | Demitour :: Droite :: q → optimiser (Gauche :: q)
    | Demitour :: Demitour :: q → optimiser q
    | Demitour :: Gauche :: q → optimiser (Droite :: q)
    | Avancer(d) :: Avancer(e) :: q → optimiser (Avancer(d +. e) :: q)
    | tt → tt
```

7. On souhaite maintenant améliorer le système :
   – D'une part, chaque ligne droite contiendra non seulement sa longueur (toujours en km) mais aussi la vitesse (en km/h) estimée sur cette ligne droite.
   – D'autre part, on rajoute la possibilité d'indiquer des ronds-points, pour lesquels il faudra noter le numéro de la sortie que doit prendre le conducteur (par exemple « au rond-point prenez la 3è sortie »).
   Redéfinir le type *mouvement* pour intégrer ces deux améliorations.

```
type mouvement_etendu = Avancer of float × float | Gauche | Droite
                      | Demitour | Rondpoint of int
```

8. Écrire une fonction *duree_mouvement* qui calcule la durée d'un mouvement, en supposant que :
   – chaque virage dure 1/1000è d'heure ;
   – un demi-tour dure 5/1000è d'heure ;
   – un rond-point dure 1/1000è d'heure par sortie franchie (donc par exemple 3/1000è pour prendre la 3è sortie) ;
   – dans les lignes droites on utilise l'information contenue dans le type.

   Écrire ensuite une fonction *duree* qui calcule la durée totale d'un trajet.

Corrigé

```
let duree_mouvement v = match v with
  | Avancer(d, v) -> d/.v
  | Gauche | Droite -> 0.001
  | Demitour -> 0.005
  | Rondpoint(i) -> 0.001 *. (float_of_int i)

let rec duree t = match t with
  | [] -> 0.
  | e :: q -> duree_mouvement e +. duree q
```

Exercice 3 : Arbres (3 points)

On considère des arbres un peu différents de ceux étudiés en TD, où seules les feuilles sont étiquetées (ici par des caractères) :

```
type arbre = Feuille of char | Noeud of arbre × arbre
```

Écrire une fonction *les_feuilles* qui prend en argument un arbre a et renvoie la liste $[(c_1,n_1)\,;\ (c_2,n_2)\,;\ \ldots\,;\ (c_p,\ n_p)]$ où :
– chaque c_i est un caractère étiquetant une feuille de a ;
– n_i est la profondeur de cette feuille (la racine étant de profondeur 0).
Exemple :

```
# les_feuilles Noeud( Noeud( Feuille('a'), Feuille('b') ), Feuille('c') )
- : (char × int) list = [('a', 2); ('b', 2); ('c', 1)]
```

Corrigé

```
let les_feuilles a =
  let rec faux a n = match a with
    | Feuille(c) -> [(c, n)]
    | Noeud(g, d) -> (faux g (n + 1)) @ (faux d (n + 1))
  in faux a 0
```

Exercice 4 : Preuves de programmes (3 points)

1. Écrire une fonction *enleve* qui prend en argument un élément e et une liste l, et renvoie la liste dans laquelle toutes les occurrences de e ont été supprimées.

Corrigé

```
let rec enleve e l = match l with
  | [] -> []
  | t :: q when t = e -> enleve e q
  | t :: q -> t :: (enleve e q)
```

2. Quel est le type de *enleve* ?

```
val enleve : 'a -> 'a list -> 'a list = <fun>
```

3. Démontrer que, pour toute liste, $taille\ (enleve\ e\ l) \leq taille\ (l)$