Devoir sur table

Novembre 2019

Documents autorisés: poly et notes de cours, notes de TD

L'épreuve durera 1h30. Le sujet vaut 30 points plus 4 points de bonus. La note sur sera alculée par la formule $\frac{2n}{3}$ où n est votre nombre total de points.

Dans ce devoir, vous pourrez utiliser les fonctions suivantes de la bibliothèque standard:

`min : 'a -> 'a -> 'a` (minimum de deux valeurs)

`List.mem : 'a -> 'a list -> bool` (appartenance à une liste)

`List.assoc : 'a -> ('a * 'b) list -> 'b` (valeur associée à une clé)

`List.map : ('a -> 'b) -> 'a list -> 'b list`

`List.filter : ('a -> bool) -> 'a list -> 'a list`

`List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

`List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

Exercice I : Récurrence terminale

Soit les fonctions

```
let f1 x =
  let rec g x y =
    if (x < 0) then y
    else x + (g (x-1) y)
  in
    (g x 0)
```

```
let f2 x =
  let rec g x y =
    if (x < 0) then y
    else (g (x-1) (x+y))
  in
    (g x 0)
```

Q1 – [1pt] `f1` utilise-t-elle une récurrence terminale ?

Q2 – [1pt] `f2` utilise-t-elle une récurrence terminale ?

Exercice II : Suites récursives

Soit la suite sur les entiers naturels définie par

$$u_n = \begin{cases} 5 & \text{si } n = 0 \\ n^2 + u_{n-1} & \text{sinon} \end{cases}$$

Q1 – [2pt] Donnez une définition de la fonction `u` telle que `(u n)` donne la valeur de $u_{\mathtt{n}}$. On fait ici l'hypothèse que l'argument `n` est positif.

Q2 – [2pt] Donnez une autre définition de la fonction `u` qui, cette fois, déclenche l'exception `(Invalid_argument "u")` lorsque `n` est négatif.

Q3 – [3pt] Donnez une version **récursive terminale** de la fonction `u`.

EXERCICE III : Listes sans doublon

> **Dans cet exercice** sur les listes, l'ordre dans lequel les valeurs seront rangées dans les listes données en résultat sera indifférent.

Q1 – [2pt] Définir la fonction de signature

```
add_item (x:'a) (xs:'a list) : 'a list
```

qui donne la liste obtenue en ajoutant `x` à `xs` sauf si `x` est déjà présent dans `xs`.

Q2 – [2pt] Définir la fonction de signature

```
add_list (xs:'a list) (ys:'a list) : 'a list
```

qui donne la liste obtenue en ajoutant tous les éléments de `xs` à `ys`, sauf ceux qui sont déjà présents dans `ys`.
Votre définition est-elle récursive terminale ?

Q3 – [2pt] Définir la fonction de signature

```
add_list_list (xss:('a list) list) : 'a list
```

telle que, si `xss` est la liste `[`$\mathtt{xs}_1$`; ...; `$\mathtt{xs}_{n-1}$`; `$\mathtt{xs}_n$`]` alors `(add_list_list xss)` est la liste

`(add_list `$\mathtt{xs}_1$` ... (add_list `$\mathtt{xs}_{n-1}$` `$\mathtt{xs}_n$`)...)`

EXERCICE IV : Listes d'associations

Dans cet exercice, on utilisera un *catalogue de produits* représenté par une liste de couples de type `(string * float) list`. Chaque élément de telles listes représente l'association d'un nom de produit (la *clé*, de type `string`) à son prix (la *valeur*, de type `float`). On suppose que dans un catalogue les clés sont unique (une seule entrée par nom de produit). Pour alléger l'écriture, on déclare

```
type cat = (string * float) list
```

Q1 – [2pt] Définir la fonction de signature

```
prix_list (xs:string list) (tarifs: cat) : float
```

qui donne la somme des prix des produits de la liste `xs` sachant que la liste `tarifs` est un *catalogue de produits*.

Exemple: si `xs` est la liste `["limonade"; "cornichons"]` et `tarifs` est la liste `[("lait", 2.35); ("cornichons", 3.80); ("concombre", 7.21); ("limonade", 12.5); ("raviolis", 5.75)]` alors `(prix_list xs tarifs)` vaut `16.3`

On fera l'hypothèse que tous les noms de produits de `xs` sont présents dans `tarifs`.

Q2 – [3pt] On suppose que l'on dispose de deux *catalogues de produits*. On cherche ici à calculer le prix d'une commande en prenant pour chaque produit le prix minimal entre ceux donnés par l'un ou l'autre catalogue. On fait l'hypothèse que les deux catalogues contiennent tous deux un prix pour chaque produit.

En utilisant la fonction `min` de la bibliothèque standard, définir la fonction de signature

```
min_prix_list (xs:string list) (tarifs1: cat) (tarifs2: cat) : float
```

qui donne la somme des prix des produits de `xs` en choisissant le prix minimal entre celui offert par `tarifs1` et celui offert par `tarifs2`

Q3 – [4+1pt] On veut maintenant savoir quels produits il faut commander chez le fournisseur 1 (qui propose le *catalogue de produits* `tarifs1`) et ceux qu'il faut commander chez le fournisseur 2 (qui propose le *catalogue de produits* `tarifs2`). Pour cela, on construit à partir de `xs` (en tenant compte de `tarifs1` et `tarifs2`) un couple de listes `(xs1,xs2)` où `xs1` est la liste des produits de `xs` qu'il faut commander chez le fournisseur 1 et `xs2` celle de ceux qu'il faut commander chez le fournisseur 2.

Définir la fonction de signature

```
split_list (xs:string list) (tarifs1:cat) (tarifs2:cat) : (string list * string list)
```

qui effectue ce calcul.

Il y a un point de bonus pour une définition récursive terminale.

Exercice V : Schémas d'itération

Dans cet exercice, il y a un point de bonus pour chaque réponse qui utilise un des itérateurs de la bibliothèque standard.

Q1 – [2+1pt] Définir la fonction de signature

```
liste_diff (xs:float list) (m:float) : float list
```

telle que `(liste_diff [`x_1`; ...; `x_n`] m)` donne la liste `[`x_1`-.m; ...; `x_n`-.m]`.

Q2 – [2+1pt] Définir la fonction de signature

```
liste_inter (xs:float list) (m1:float) (m2:float) : float list
```

qui donne la liste des éléments de `xs` qui sont compris (au sens large) entre `m1` et `m2`.

Q3 – [2+1pt] Définir la fonction de signature

```
sum_square (xs:float list) : float
```

qui donne la somme des carrés des éléments de `xs`.