Feuille d'exercices n°3

Exercice I : Filtrages sur les listes

Q1 – Définir la fonction de signature

```
len_comp_3 (xs:'a list) : int
```

qui donne	1	si `xs` contient au moins 3 éléments
	0	si `xs` contient exactement 3 éléments
	-1	sinon.

Q2 – Définir la fonction de signature

```
swap_hd_snd (xs:'a list) : 'a list
```

donne la liste obtenue en inversant l'ordre des deux premiers éléments de `xs`. Par exemple:
`(swap_hd_snd [1;2;3;4])` donne la liste `[2;1;3;4]`.

La fonction `swap_hd_snd` renvoie `xs` inchangée si elle contient moins de 2 éléments.

Q3 – Définir la fonction de signature

```
swap_hd_fst (xs:('a*'a) list) : ('a*'a) list
```

qui donne la liste obtenue en inversant l'ordre des valeurs dans le premier élément de `xs`. On aura que `(swap_hd_fst [])` = `[]`.

Exemple:
`(swap_hd_fst [ (1,2); (3,4); (5,5) ])` donne la liste `[ (2,1); (3,4); (5,5) ]`.

Exercice II : Constructions de liste avec récursion sur entiers

Q1 – Définir la fonction de signature

```
repeat (n:int) (x:'a) : 'a list
```

qui donne la liste contenant `n` fois `x`.

Exemples:
`(repeat 7 true)` donne `[true; true; true; true; true; true; true]`
`(repeat 0 "Hello")` donne `[]`
`(repeat (-42) [1;2;3])` donne `[]`

Q2 – Définir la fonction

```
range_i (i:int) (j:int) : (int list)
```

qui donne la liste `[i; i+1; ..; j]`.
Remarque: si `i` > `j` alors `(range i j)` donne la liste vide.

Exercice III : Manipulation de listes avec récursion sur listes

Q1 – Définir la fonction de signature

```
intercale1 (z:'a) (xs:'a list) : 'a list
```

qui intercale `z` entre les éléements de `xs`. Le premier et le dernier éléments de `(intercale z xs)` sont le premier et le dernier éléments de `xs`.

Exemples:
`(intercale1 1 [])` donne `[]`
`(intercale1 0 [5])` donne `[5]`
`(intrecale1 0 [1;2;3])` donne `[1;0;2;0;3]`

Q2 – Définir la fonction de signature

```
begaie (xs:'a list) : ('a list)
```

qui double la taille d'une liste en dupliquant chacun de ses éléments. Exemples:
`(begaie [])` donne `[]`
`(begaie [1;2;3])` donne `[1;1;2;2;3;3]`

Exercice IV : Application, filtrage, accumulation

Q1 – Définir la fonction de signature

```
inverse_i (xs:float list) : float list
```

Schématiquement, on a que `(inverse_i [x1; ..; xn])` donne la liste `[ 1 / x1; ..; 1 / xn]` et on a que `(inverse_f [])` vaut `[]`. **ATTENTION** au type des éléments des listes.

Utilisez la fonction `float_of_int` de la bibliothèque standard.

Q2 – Définir la fonction de signature

```
list_interval (ns:int list) : int list
```

qui donne la liste des éléments de `ns` qui sont compris entre -10 et 10 (inclus).

Q3 – Définir la fonction de signature

```
parenthese (xs:string list) : string
```

qui donne la chaîne de caractères obtenue en mettant entre parenthèses et en concaténant les éléments de `xs`.

Exemples:
`(parenthese [])` vaut la chaîne vide `""`
`(parenthese ["do";"re";"mi"])` donne la chaîne `"(do)(re)(mi)"`

Utilisez l'opérateur de concaténation des chaînes de caractères `^` (caractère accent circonflexe, notation infixe).

Travaux sur machines

Exercice V : Tri fusion

Le tri par fusion repose sur l'utilisation d'une *fonction d'interclassement* appliquée à deux listes triées. La fonction d'interclassement construit son résultat en parcourant en parallèle les deux listes et en sélectionnant le plus petit des deux éléments en tête de liste. Le parcours se poursuit en enlevant l'élément sélectionné jusqu'à ce qu'une des deux listes soit épuisée.

On utilisera l'opérateur de comparaison polymorphe `<` (notation infixe) prédéfini en OCaml. Il est de type `'a -> 'a -> bool`.

Q1 – Donner la définition récursive de la fonction de signature sigmerge: (xs:'a list) (ys:'a list) 'a list qui donne la liste obtenue par interclassement des éléments de `xs` et `ys` selon l'ordre `<`.

Exemples:
`(merge [8; 5; 7] [])` donne `[8; 5; 7]`
`(merge [] [8; 5; 7])` donne `[8; 5; 7]`
`(merge [1] [3; 6; 2])` donne `[1; 3; 6; 2]` (car ml($1 < 3$))
`(merge [5] [3; 6; 2])` donne `[3; 5; 6; 2]` (car `not (5 < 3)` et `(5 < 6)`)
`(merge [1; 5] [3; 6; 2])` donne `[1; 3; 5; 6; 2]`
On remarque que si les listes sont ordonnées, le résultat de la fusion est également une liste ordonnée:
`merge [0;2;4;8] [1;3;5;7]` donne `[0; 1; 2; 3; 4; 5; 7; 8]`

Q2 – Donner la définition de la fonction de signature

```
split (xs:'a list) : ('a list) * ('a list)
```

Schématiquement `(split [`x_1`; `x_2`; `x_3`; `x_4`; ...])` donne le couple de listes `([`x_1`; `x_3`; ...], [`x_2`; `x_4`; ...])`.

Attention à la parité du nombre d'élements de `xs`. Par exemple:
`(split [1])` donne `([1],[])`
`(split [1;2])` donne `([1],[2])`
`(split [1;2;3])` donne `([1;3],[2])`

On utilise une liaison locale `let-in` pour destructurer le résultat de l'appel récursif.

Q3 – En déduire la définition de la fonction

```
merge_sort (xs:'a list) : 'a list
```

qui donne une liste ordonnée par `<` des éléments de `xs`(pour la relation d'ordre induite par la fonction de comparaison passée en argument à `merge_sort`) de son second argument.

Exercice VI : D'autres ordres

On a utilisé l'ordre prédéfini `<` pour définir la fonctions de fusion. Mais on veut ppouvoir utiliser d'autres ordres, par exemple, pour trier en ordre croissant, sans avoir à redéfinir `(merge)`. Pour cela, on passe la fonction de comparaison en paramètre.

Q1 – Définir

```
merge (cmp: 'a -> 'a -> bool) (xs:'a list) (ys:'a list) : 'a list
```

qui donne la liste obtenue par interclassement des éléments de `xs` et `ys` selon l'ordre induit par la fonction `cmp` (`x` est avant `y` lorsque `(cmp x y)` vaut `true`).

Par exemple

- `(merge (fun x y -> x >= y) [2;5] [3;1])` donne `[3; 2; 5; 1]`

Q2 – Définir ensuite

```
merge_sort (cmp:'a -> 'a -> bool) (xs:'a list) : 'a list
```

qui donne la liste ordonnées des éléments de `xs` selon l'ordre induit par `cmp`.

Q3 – En déduire une fonction de tri pour les listes de couples d'entiers selon l'ordre défini par:

> *le couple* (x_1, x_2) *est plus petit que le couple* (y_1, y_2) *si et seulement si l'entier* $x_1 + x_2$ *est inférieur ou égal à l'entier* $y_1 + y2$.