TD6 – Récursivité terminale

Exercice 6.1 (Somme des premiers entiers naturels impairs)**.**
Définir une fonction récursive terminale de signature `sum_impairs (n : int) : int` qui calcule la somme des n premiers entiers naturels impairs $(1 + 3 + \cdots + (2n - 1))$.

```
# (sum_impairs  4);;
- : int = 16
```

Exercice 6.2 (Itération d'une fonction)**.**
Définir une fonction récursive terminale de signature :

```
iter_f (n : int) (f : 'a -> 'a) (x : 'a) : 'a
```

qui calcule $f^n(x)$.

```
# iter_f 5 (fun x -> x+1) 0;;
- : int = 5
```

Exercice 6.3 (Concaténation de listes)**.**

1. Définir une fonction récursive terminale de signature :

   ```
   rev_append (l1 : 'a list) (l2 : 'a list) : 'a list
   ```

   telle que `rev_append l1 l2 = (List.rev l1) @ l2`

   ```
   # rev_append [1;2] [3;4;5;6];;
   - : int list = [2; 1; 3; 4; 5; 6]
   ```

2. En déduire une fonction récursive terminale de signature `rev (l : 'a list) : 'a list` qui inverse l'ordre d'apparition des éléments de la liste `l`.

   ```
   # rev [1;2;3;4;5];;
   - : int list = [5; 4; 3; 2; 1]
   ```

3. Utiliser la fonction `rev` pour définir une fonction récursive terminale de signature :

   ```
   append (l1 : 'a list) (l2 : 'a list) : 'a list
   ```

   qui concatène deux listes.

   ```
   # append [1;2] [3;4;5;6];;
   - : int list = [1; 2; 3; 4; 5; 6]
   ```

Exercice 6.4 (Itérateurs sur les listes)**.**

1. Définir une fonction récursive terminale de signature :

   ```
   map (f : 'a -> 'b) (l : 'a list) : 'b list
   ```

   telle que `map f l = List.map f l`.
2. Définir une fonction récursive terminale de signature :

   ```
   filter (f : 'a -> bool) (l : 'a list) : 'a list
   ```

   telle que `filter f l = List.filter f l`.
3. Définir une fonction récursive terminale de signature :

   ```
   partition (f : 'a -> bool) (l : 'a list) : 'a list * 'a list
   ```

   telle que `partition f l` est la paire de listes `(l1,l2)` où `l1` contient les éléments `e` de `l` tels que `f e = true` et `l2` contient les éléments `e` de `l` tels que `f e = false`. L'ordre d'apparition des éléments dans la liste `l` est conservé dans les listes `l1` et `l2`.

```
# partition (fun x -> x mod 2 = 0) [0; 1; 2; 3; 4; 5; 6; 7; 8];;
- : int list * int list = ([0; 2; 4; 6; 8], [1; 3; 5; 7])
```

4. Définir une fonction récursive terminale de signature :

   `fold_right (f : 'a -> 'b -> 'b) (l : 'a list) (b : 'b) : 'b`

   telle que `fold_right f l b = List.fold_right f l b`.

Exercice 6.5 (Parcours d'arbres binaires)**.**

On considère le type des arbres binaires défini par :

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

1. Définir une fonction récursive terminale de signature :

   `post_travers (bt : 'a btree) : 'a list`

   qui construit la liste du parcours postfixe de l'arbre binaire `bt`. Par exemple, avec l'arbre `t_ex` introduit page 9, on obtient :

```
# post_travers t_ex;;
- : int list = [4; 5; 3; 7; 3; 2; 5]
```

2. Définir une fonction récursive terminale de signature :

   `pref_travers (bt : 'a btree) : 'a list`

   qui construit la liste du parcours préfixe de l'arbre binaire `bt`. Par exemple, avec l'arbre `t_ex` introduit page 9, on obtient :

```
# pref_travers t_ex;;
- : int list = [5; 3; 4; 5; 2; 3; 7]
```

3. Définir une fonction récursive terminale de signature :

   `inf_travers (bt : 'a btree) : 'a list`

   qui construit la liste du parcours infixe de l'arbre binaire `bt`. Par exemple, avec l'arbre `t_ex` introduit page 9, on obtient :

```
# inf_travers t_ex;;
- : int list = [4; 3; 5; 5; 3; 7; 2]
```