Les listes

1 Parcours simples

Exercice 1.1 (Somme)

Écrire une fonction qui calcule la somme de tous les éléments d'une liste d'entiers.

Exercice 1.2 (Compte)

Écrire une fonction qui compte le nombre d'occurrences d'une valeur donnée dans une liste quelconque.

Exercice 1.3 (Recherche)

Écrire une fonction qui recherche si un élément est présent dans une liste.

Exercice 1.4 (i$^{\text{ème}}$)

Écrire une fonction qui donne la valeur du $i^{ème}$ élément d'une liste. La fonction devra déclencher une exception `Invalid_argument` si i est négatif ou nul, ou une exception `Failure` si la liste est trop courte.

Exercice 1.5 (Maximum)

Écrire une fonction qui retourne la valeur maximum d'une liste.

---

2 Le résultat est une liste

Exercice 2.1 (Liste arithmétique)

Écrire la fonction `arith_list` n a_1 r qui construit la liste des n premiers termes de la suite arithmétique de premier terme a_1 et de raison r.

Exemples d'application :

```
# arith_list 12 2 1;;
 - : int list = [2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13]
# arith_list 11 0 3;;
 - : int list = [0; 3; 6; 9; 12; 15; 18; 21; 24; 27; 30]
```

Exercice 2.2 (Concaténation)

Écrire une fonction qui concatène deux listes (l'équivalent de l'opérateur `@`).

Exemple d'application :

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

---

3 Listes et ordres

Exercice 3.1 (Croissance ?)

Écrire une fonction qui teste si une liste est triée en ordre croissant.

Exercice 3.2 (Recherche)

Écrire une fonction qui recherche si un élément est présent dans une liste triée (en ordre croissant).

Exercice 3.3 (Suppression)

Écrire une fonction qui supprime d'une liste l triée (en ordre croissant) la première occurrence d'un élément x (s'il est présent).

Exercice 3.4 (Insertion)

Écrire une fonction qui ajoute un élément à sa place dans une liste triée en ordre croissant.

Exercice 3.5 (Inverse)

Écrire une fonction qui inverse une liste :

1. en utilisant l'opérateur `@` ;
2. sans utiliser l'opérateur `@`.

Que pensez-vous de la complexité de ces deux fonctions ?

---

4 Deux listes

Exercice 4.1 (Égalité)

Écrire une fonction qui teste si deux listes sont identiques.

Exercice 4.2 (Fusion)

Écrire une fonction qui fusionne deux listes triées en une seule.
Comment éliminer les éléments communs aux deux listes ?

Exemple d'application :

```
# merge [1; 5; 6; 8; 9; 15] [2; 3; 4; 5; 7; 9; 10; 11];;
 - : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 15]
```

5 Des listes et des couples

Exercice 5.1 (Association – *C1 - nov. 2017*)

Écrire la fonction `assoc k list` où `list` est une liste de couples (*key*, *value*) triés par clés (key) croissantes. Les clés sont des entiers naturels non nuls. Elle retourne la valeur (value) associée à la clé (key) `k`. Si `k` n'est pas valide ou si aucun couple n'a pour clé `k`, elle déclenche une exception.

Exemples d'utilisation :

```
♯ assoc 5 [(1, "one"); (2, "two"); (3, "three"); (5, "five"); (8, "eight")];;
 - : string = "five"

♯ assoc 4 [(1, "one"); (2, "two"); (3, "three"); (5, "five"); (8, "eight")];;
 Exception: Failure "not found".

♯ assoc (-1) [(1, "one"); (2, "two"); (3, "three"); (5, "five"); (8, "eight")];;
 Exception: Invalid_argument "k not a natural".
```

Exercice 5.2 (Dominos – *C1♯ - avril 2017*)

Pour cet exercice, on s'intéresse au jeu de Dominos :
— Un domino sera représenté par un couple d'entiers (a, b)
— Une suite de dominos (chaîne) sera donc une liste de couples d'entiers $[(a_1, b_1); (a_2, b_2); \cdots; (a_n, b_n)]$.

On considère qu'une chaîne de dominos est valide si les parties voisines des dominos ont le même nombre de points (voir les exemples ci-dessous).

Écrire la fonction `is_dominos` qui vérifie si une chaîne de dominos est valide.

Exemples d'application :

```
# is_dominos [] ;;
- : bool = true

# is_dominos [(1,2); (2,3); (3,3); (3,6)] ;;
- : bool = true

# is_dominos [(2,3); (2,4); (1,4)] ;;
- : bool = false
```

Exercice 5.3 (Codage RLE simplifié)

Le but de cet exercice est de compresser une liste d'éléments grâce à une version simplifiée de l'algorithme RLE (Run Length Encoding).

La compression RLE standard permet de compresser des éléments par factorisation, mais uniquement lorsque ceci permet de gagner de la place. Cette compression est essentiellement utilisée dans le format d'images pcx, elle est non destructive, et très performante sur des images comportant de longues suites de pixels de couleur constante. Votre algorithme RLE simplifié effectuera cette factorisation dans tous les cas, y compris quand cela prend plus de place que l'objet non compressé.

Le flux que l'on encode est une liste d'éléments ; le flux encodé est une liste de couples, chaque couple étant composé du nombre d'éléments consécutifs identiques, puis de cet élément.

1. Écrire une fonction `encode_rle` de type `'a list -> (int * 'a) list` qui permet de compresser une liste en utilisant l'algorithme RLE.

   ```
   ♯ encode_rle ['b';'b';'b';'c';'a';'a';'e';'e';'e';'e';'d';'d'] ;;
    - : (int * char) list = [(3, 'b'); (1, 'c'); (2, 'a'); (4, 'e'); (2, 'd')]
   ```

2. Écrire une fonction `decode_rle` qui permet de décompresser une liste compressée en RLE.

   ```
   ♯ decode_rle [(6, "grr")] ;;
    - : string list = ["grr"; "grr"; "grr"; "grr"; "grr"; "grr"]
   ```

Exercice 5.4 (Liste de couples)

1. Écrire la fonction `combine` ayant les spécifications suivantes :
   `val combine : 'a list -> 'b list -> ('a * 'b) list`

   Transform a pair of lists into a list of pairs : `combine [a1;...;an] [b1;...; bn]` is `[(a1,b1);...; (an,bn)]`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

   Exemple :

   ```
   # combine [1; 2; 3] ['a'; 'b'; 'c'];;
    - : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
   ```

2. Écrire la fonction `split` ayant les spécifications suivantes :

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

   Transform a list of pairs into a pair of lists : split [(a1,b1) ; ... ; (an,bn)] is ([a1 ; ... ; an], [b1 ; ... ; bn]). Not tail-recursive.

   Exemple :

   ```
   # split [(1,'a'); (2,'b'); (3,'c')];;
    - : int list * char list = ([1; 2; 3], ['a'; 'b'; 'c'])
   ```

6 Petits problèmes

Exercice 6.1 (Crible d'Ératosthène)

Le *crible d'Ératosthène* est un algorithme simple qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné N. L'algorithme procède par éliminations :
— Au départ, tout nombre à partir de 2 est supposé premier.
— Ensuite, pour chaque entier qui est premier, ses multiples ne sont pas premiers et sont donc éliminés.

Écrire la fonction `eratosthenes` qui appliquée à un entier n strictement supérieur à 1 donne la liste de tous les entiers premiers jusqu'à n.
Exemple d'application :

```
# eratosthenes 30 ;;
 - : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```

Exercice 6.2 (Bonus : Suite)

Soit la suite suivante :

ligne 0 :	1	contient 1 "1"
ligne 1 :	11	contient 2 "1"
ligne 2 :	21	contient 1 "2" suivi de 1 "1"
ligne 3 :	1211	contient 1 "1" followed by 1 "2" followed by 2 "1"
ligne 4 :	111221	...
ligne 5 :	312211	
ligne 6 :	13112221	
...		

Écrire une fonction `sequence` qui retourne la $n^{ème}$ ligne de cette suite, sous forme d'une liste d'entiers.

Exemple d'application :

```
# sequence 5 ;;
 - : int list = [3; 1; 2; 2; 1; 1]
# sequence 0 ;;
 - : int list = [1]
```