TP numéro 2

N.B. : pour pouvoir utiliser la forme spéciale *let*, vous devez passer au niveau de langage *Etudiant niveau intermédiaire, plus lambda* avec toujours *write* pour *syntaxe de sortie*.

1 Fonctions sur les listes

• Écrire une fonction qui renvoie la sous-liste formée de tous les symboles d'une liste.

```
(symboles '(a 2 b (6 z) "toto" 7 f)) -> (a b f)
```

• Écrire une fonction qui, étant donnée une liste de longueur paire, regroupe deux éléments consécutifs dans une liste.

```
(regroupe '(a b c d e f)) -> ((a b) (c d) (e f))
```

• Écrire une fonction qui renverse une liste.

```
(renverse '(a b c d)) → (d c b a)
```

• Écrire une fonction qui remplace toutes les occurrences d'un élément e_1 dans une liste L par un autre élément e_2.

```
(remplace 'o 'i '(b o n j o u r)) -> (b i n j i u r)
```

2 Mémorisation

• Écrire une fonction qui rend la liste des n+1 premiers nombres de la suite de Fibonacci (de u_0 à u_n) sans faire plusieurs fois les mêmes calculs.

```
(fibo-liste 5) → (8 5 3 2 1 1)
```

• Utiliser la fonction `fibo-liste` pour écrire une nouvelle version de la fonction écrite en TD qui calcule le $n^{ième}$ terme de la suite de Fibonacci. Comparez le nombre de calculs effectués par les deux versions de la fonction pour n=4. Testez les deux versions de la fonction pour n=30. Êtes-vous maintenant convaincu-e de l'intérêt de calculer la complexité d'un algorithme ?

• Écrire une fonction qui étant donnée une liste, construit une liste de deux sous-listes : celle contenant les symboles et celle contenant les nombres. *Attention à ne faire qu'un seul parcours de la liste*.

```
(trie '(tor 1 tue la 2 3 pin 4)) → ((tor tue la pin) (1 2 3 4))
```

3 Calculs en remontant ou en descendant

• On veut écrire une fonction qui calcule la somme des chiffres d'un entier positif.

```
(somme-des-chiffres 341) → 8
```

1. Définir une version récursive de cette fonction de la manière habituelle, en utilisant le résultat de l'appel récursif pour effectuer les calculs en remontant.

2. Définir ensuite une version qui, bien qu'étant récursive, s'inspire de la programmation itérative, en utilisant un paramètre supplémentaire pour effectuer les calculs en descendant.

Indications :

- La fonction modulo permet de trouver le reste de la division (par 10) : `(modulo 341 10) -> 1`
- La fonction quotient permet de trouver le résultat de la division entière (par 10) : `(quotient 341 10) -> 34`

Pour s'entrainer (exercices supplémentaires facultatifs)

• Écrire une fonction qui vérifie que tous les éléments d'une liste sont égaux.

• Écrire une fonction qui supprime tous les éléments d'une liste L qui sont égaux à un élément `e` passé en paramètre.

```
(supprime '(a b a d e f a) 'a) -> (b d e f)
```

• Écrire une fonction qui renvoie la sous-liste formée des n premiers éléments d'une liste.

```
(premiers 3 '(a b c d e)) -> (a b c)
```

• Écrire une fonction qui calcule $\frac{n!+100}{n!+4}$ avec un seul appel à (factorielle n).

La fonction prédéfinie `random` permet d'engendrer des nombres entiers au hasard : `(random x)` retourne un entier dans l'intervalle [0, x[.

• Écrire une fonction qui retourne une liste de nombres entiers positifs pris au hasard. Les deux paramètres de cette fonction sont la longueur de la liste à construire et la valeur maximale des nombres à générer.

```
(liste_random 5 10) → (7 9 6 7 4)
```

• Écrire une fonction qui retourne une liste composée d'un nombre entier pris au hasard entre 0 et N, et d'un booléen indiquant si ce nombre est un multiple de trois ou de sept.

```
(nb_test 10) → (5 #f)
(nb_test 10) → (6 #t)
```

• Écrire une fonction identique à la fonction liste_random, mais qui ne retourne que des nombres pairs.

```
(liste_random_pairs 5 10) → (4 10 6 4 6)
```

• Écrire une fonction qui, étant donné un nombre x et une liste de nombres L, construit deux listes : celle des nombres de L inférieurs ou égaux à x, et celle des nombres de L supérieurs à x.

```
(separe 3 '(5 1 2 3 6 4)) → ((1 2 3) (5 6 4))
```