TDTP 3 : mémorisation (let)

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.*

À préparer avant la séance

• Donner la valeur retournée par les expressions ci-dessous :

- `(list 'a (cons '(b c) '(d)))`
- `(append '(a) '(b c) '(d))`
- `(append (list 'a '(b)) '(c))`

• On définit les listes suivantes :

```
(define L1 '(a b c))
(define L2 '(d e))
```

Donner les expressions utilisant L1 et L2 et permettant d'obtenir les résultats suivants :

… → `((b c) (d e))`
… → `(c e)`
… → `(b c d)`

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

À faire pendant la séance

(TP) Définir une fonction qui, étant donnés les coefficients a, b et c d'un trinôme ax^2+bx+c, retourne les racines sous la forme d'une liste. La liste sera vide si $\Delta<0$, n'aura qu'un élément si $\Delta=0$, et deux éléments sinon.

`(racines 1 2 -3)` → `(-3 1)`

(TD) Définir une fonction qui retourne 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)`

(TD) Utiliser la fonction `fibo-liste` pour écrire une nouvelle version de la fonction écrite au TDTP1 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=50. Êtes-vous maintenant convaincu-e de l'intérêt de calculer la complexité d'un algorithme ?

(TD) Nous souhaitons définir la fonction `som-prod` qui retourne la somme et le produit d'une liste non vide de nombres.

`(som-prod '(2 4 1 3))` → `(10 24)`

1. Écrire une première version récursive de la fonction sans utiliser le let. Dérouler ensuite le fonctionnement de cette fonction sur l'exemple ci-dessus.

2. Écrire une seconde version de cette fonction en y intégrant le let, c'est-à-dire en utilisant le résultat de l'appel récursif pour effectuer les calculs. Les calculs seront effectués de manière habituelle, c'est-à-dire en remontant.

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

Pour s'entraîner (exercices supplémentaires facultatifs)

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

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