TD de LISP numéro 1 et 2

---

1. **lambda-expressions**
   évaluer les expressions suivantes en précisant les environnements au cours de l'évaluation
   - `((lambda (x) (+ (* 2 x) 3)) 4)`
   - `((lambda (x y) (* (+ x 2) (+ y 6))) 7 8)`
   - `((lambda (v) ((lambda (x) (* 3 x)) (+ v 2))) 8)`
   - `((lambda (v) ((lambda (x) (* v x)) (+ v 2))) 8)`
   - `((lambda (v) ((lambda (v) (* 3 v)) (+ v 2))) 8)`
   - `((lambda (x y z) (+ (* x y) (* y z))) 2 3 4)`
   - `((lambda (x y) (* x x y y)) 4)`
   - `((lambda (x) (* x x 2)) 4 5)`
   - `(lambda (x) (* x x 2))`

---

2. **fonctions globales**
   - `(defun f (x) (+ 3 x))`
     `(defun g (v) (* 5 (f (+ v 2))))`
     évaluer `(g 8)`

     qu'est ce que cela donne si f est définie par
     `(defun f (x) (+ v x))`
     Comparer avec la lambda-expression correspondante qui précède.

   - définir quelques fonctions numériques courantes
     - `(fact n)` : factorielle d'un entier n
     - `(fibo n)` : fibonacci (suite définie par `u(0)=u(1)=1` et `u(n)=u(n-1)+u(n-2)`)
       donner une approximation de la complexité de votre définition
       sur la base du temps mis pour calculer `(fibo 15)`, déduire une estimation du temps pour `(fibo 50)`.

---

3. **les listes et les cellules**
   - comparer () et (()) et ((())) :
     - comparer les `CAR` et `CDR` de ces listes
     - comparer leurs représentations en doublets
     - et leurs représentations complètement pointées
     - que donne leur comparaison par les prédicats `eq` et `equal` ?
       et avec `=` ?
   - de combien de cellules sont faites les listes :
     - (1 2 3 4)
     - (1 (2 3) 4)
     - (1 (2) (3) 4)
       donner leur représentation par doublets et écrire leur représentation en paire pointée
   - qu'obtenez-vous avec les exemples précédents si vous inversez partout les `CAR` et les `CDR` ?
   - représentation par doublets pour les définitions des fonctions `f` et `g` qui précèdent.

---

4. **fonctions sur les listes plates**
   On ne considère que les éléments de la liste, c'est-à-dire les `CAR` des cellules.
   Ecrire les fonctions suivantes :
   - `(member x l)` qui retourne la sous-liste de la liste `l` commençant par l'élément `x`.
   - `(length l)` qui retourne la longueur d'une liste plate, c'est-à-dire le nombre de cellules au premier niveau ;
   - `(last l)` qui retourne la dernière cellule d'une liste plate (au premier niveau) ;
     faites une version qui assure qu'un seul test est effectué à chaque pas de la récursion ;
   - `(makelist n)` qui crée une liste de longueur n, contenant les entiers de 1 à n, en ordre décroissant ;
     et en ordre croissant ?
   - `(copylist l)` qui retourne une copie au premier niveau de la liste `l` ;
   - `(remove x l)` qui retourne une copie de la liste `l` privée des occurrences de `x` ;
     faire la même chose en n'enlevant que la première occurrence de x ;
   - `(append l1 l2)` qui concatène 2 listes ;
   - `(adjoin x l)` qui "ajoute" `x` à la liste `l` si `x` n'y est pas déjà ;
     dans la fonction appelante, que devient `l` ?

---

5. **fonctions sur les arbres binaires**
   On considère que chaque cellule est un noeud dont les `CAR` et `CDR` sont les successeurs
   - `(size tree)` qui retourne la taille d'un arbre, c'est-à-dire son nombre de cellules total, à tous les niveaux ;
     alternativement, définir la fonction de façon à calculer le nombre de feuilles, ou le nombre de feuilles non `NIL`.
   - `(copytree tree)` qui copie un arbre ;
   - `(subst x y tree)` qui substitue dans l'arbre `tree` toutes les occurrences de x par y ;
   - `(tree-leaves tree)` qui retourne la liste des feuilles de l'arbre

---

6. **tests d'égalité**
   - comparer les différentes fonctions de tests d'égalité (`=`, `eq`, `eql`, `equal`) sur les nombres, en faisant varier l'ordre de grandeur de `n` dans `(= (fact n) (fact n))`.
   - proposer une définition de `equal` correspondant aux spécifications.

---

7. **récursion terminale et enveloppée**
   - Les diverses fonctions qui précèdent sont-elles en récursion terminale ou enveloppée ?
   - Dans ce dernier cas, en donner une version en récursion terminale, par exemple en appliquant la transformation du cours ?
   - dans le cas de `copylist` et `makelist`, que calculent leurs versions récursives terminales ?
   - appliquée aux récursions doubles, la transformation du cours ne donne pas une récursion terminale sur les deux appels récursifs : trouver une autre solution pour `fibonacci` en récursion terminale ; quelle est la complexité de cette nouvelle définition ?
   - pour les autres récursions doubles (fonctions sur les arbres par ex.) la transformation du cours donne une récursion partiellement terminale : appliquer à `size`.
     qu'est-ce que cela donne pour `copytree` et `tree-leaves` ?
