TP01 : Montée en compétences Lisp

Léo Przewlocki - Alexandre Touzeau

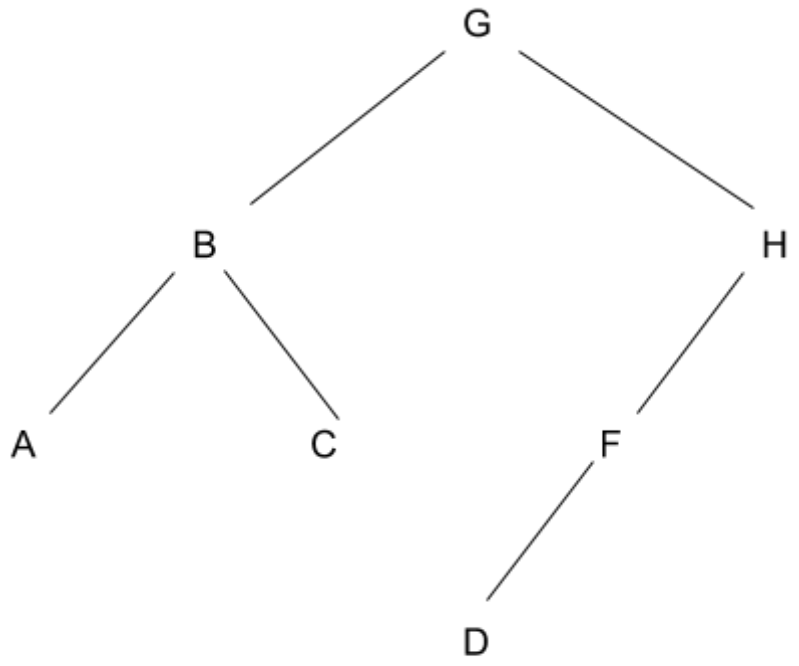
Exercice 1

1)

- 35 est un atome - Confirmation à l'aide de (atom 35) qui nous renvoie la valeur True
- (35) est une liste constitué de 1 atome - Confirmation à l'aide de (list '(35)) renvoie True
- (((3) 5) 6) est une liste
- -34RRRR est un atome
- T est un atome
- NIL est à la fois une liste et un atome
- () est équivalent à NIL c'est donc également à la fois une liste et un atome

2)

Le sujet ne spécifie pas dans quel représentation on se place pour représenter la liste (((A)(B)(C)) G (((((D)) F) H))) sous forme d'arbre. Nous avons donc choisis arbitrairement la forme infixée.



L'objet le plus profond est **D**.

3)

- (CADR (CDR (CDR (CDR '(DO RE MI FA SOL LA SI))))) nous donne SOL
- (CONS (CADR '((A B)(C D))) (CDDR '(A (B (C))))) nous donne ((C D))
- (CONS (CONS 'HELLO NIL) '(HOW ARE YOU)) donne ((HELLO) HOW ARE YOU)
- (CONS 'JE (CONS 'JE (CONS 'JE (CONS 'BALBUTIE NIL)))) donne (JE JE JE BALBUTIE)
- (CADR (CONS 'TIENS (CONS '(C EST SIMPLE) ()))) donne (C EST SIMPLE)

4)

```
(defun nombres3 (L)
  (if (and (and (and (numberp (car L))) (numberp (cadr L))) (numberp (caddr L))) (princ "BRAVO !")
    (princ "PERDU !")
  )
)
```

Pour la fonction nombres3 on utilise simplement l'opération logique "and" et l'on test successivement si les trois premiers elements sont un nombre (numberp).

```
(defun grouper (L1 L2)
  (if (and (not (equal L1 ())) (not (equal L2 ()))) (cons (list (car L1) (car L2)) (grouper (cdr L1) (cdr L2)))
  NIL
  )
)
```

Ici on commence ici par vérifier que les deux listes font la même taille.
Si c'est le cas on unifie (à l'aide de cons - unifie le car et le cdr passé en argument) le car de L1 et celui de L2 et on rappelle de manière récursive sur le cdr de chacune des listes.

```
(defun monReverse (L)
  (if (null L) NIL
    (append (monReverse (cdr L)) (list (car L)))
  )
)
```

La fonction est ici basique. On joue ici sur le récursivité pour append dans une nouvelle liste dans l'ordre inverse de la liste de base.

```
(defun palindrome(L)
  (if (equal (monReverse L) L) T
    NIL
  )
)
```

Une des propriété d'un palindrome est d'être égal à son inverse. Il suffit alors d'utiliser la fonction "monReverse" définit précédemment.

```
(defun monEqual (X1 X2)
(if (and (eq X1 NIL) (eq X2 NIL)) NIL
  (if (and (listp X1) (listp X2))
    (if (eq (car X1) (car X2)) (monEqual (cdr X1) (cdr X2))
      NIl
    )
    (if (eq X1 X2) T
      NIL)
  )
))
```

Il s'agit ici encore une fois d'une fonction récursive.
eq est l'égalité des pointeurs.
equal est l'égalité des symboles aux feuilles de l'objets.
On ne peut pas utiliser eq pour les nombres et caractères.

Nous avons ici testé toutes nos fonctions à l'aide de simples tests comme celui-ci:
"*(write (palindrome '(x a m a x)))*"

Exercice 2

code lisp
`(defun list-triple-couple (x)
  (mapcar (lambda (n) (list n (* 3 n))) x)
)`

Il suffit de comprendre comment fonctionne '*mapcar*'. Le premier argument est la fonction qui sera appliquée à chaque élément de la liste, qui est le deuxième argument.

La fonction passé à *mapcar* est comme demandé une fonction anonyme., qui commence par un lambda, suivi la variable en argument et la fonction elle-même.

Exercice 3

my-assoc: Cette fonction est un simple parcours de liste à l'aide de "dolist", on test pour chaque élément de la liste si le car est égal à la clé passée en paramètre. Si c'est le cas alors on affiche l'élément de la liste : la clé + la valeur.

cles: Nous avons ici opté pour une fonction récursive (après avoir tenté dans un premier temps de refaire un dolist). La condition d'arrêt est que la liste passée en argument soit nulle (NIL). On utilise "append" pour chaque occurrence de la fonction sur le car du car (soit la clé du premier élément puis la clé du second etc…). On rappelle ensuite récursivement la fonction sur le cdr de la liste passée en argument. On obtient ainsi au final la liste des clés.

creation: De la même manière que pour la fonction précédente nous avons ici fait le choix de la récurrence. La condition d'arrêt étant sur la liste des clés passées en argument nous avons ici supposé que l'utilisateur entre bien deux listes de la même taille. La fonction est ici simple, il suffit d'append les car des deux listes puis de rappeler récursivement sur les cdr. La difficulté est de ne pas oublier de mettre "list" juste après le append pour construire une liste finale constituée de plusieurs sous-liste clé-valeur.

Remarque: Les fonctions sont ici toute suivie d'un commentaire pour le test.

```
(defun my-assoc (cle l)
  (dolist (x l NIL)
    (if (eq (car x) cle) (print x) NIL)
  )
)
;(my-assoc 'Pierre '((Yolande 25) (Pierre 22) (Julie 45)))

(defun cles (l)
  (if (eq l NIL) NIL
    (append (list (car (car  l))) (cles (cdr l))) ;Attention a ne pas oublier le "list" sinon exception pour le dernier elmt de l
  )
)
;(write (cles '((Yolande 25) (Pierre 22) (Julie 45))))

(defun creation (c v)
  (if (eq c NIL) NIL
    (append (list(append (list (car c)) (list (car v)))) (creation (cdr c) (cdr v)))
  )
)
;(write (creation '(Yolande Pierre Julie) '(25 22 45)))
```

Exercice 4

A.

Après avoir complété BaseTest, nous nous retrouvons avec une liste de 22 éléments.

```
(setq BaseTest
   '(
      (
         "Campagnes de Clovis Ier"
         486
         508
         (
            (
               "Royaume Franc"
            )
            (
               "Soissons"
               "Royaume alaman"
               "Royaume des Burgondes"
               "Royaume wisigoth"
               "Royaume ostrogoth"
               "Royaume wisigoth"
            )
         )
         (
            "Soissons"
            "Zulpich"
            "Dijon"
            "Vouille"
            "Arles"
         )
      )
      (
         "Guerre de Burgondie"
         523
         533
         (
            (
               "Royaume Franc"
            )
            (
               "Royaume des Burgondes"
            )
         )
```

Code extrait de de l'implémentation de la base de données BaseTest. Nous avons implémenté la base de donnée en entier.

B.

Les fonctions de cette question sont assez triviales. Nous connaissons la position des éléments demandés dans les listes de conflit donc il suffit de manier les *'car'* et '*cdr'*. Pour la dernière fonction, nous utilisons '*nthcdr 4'* afin d'appliquer quatre fois '*cdr'*.

```lisp
(defun dateDebut (conflit)
    (cadr conflit)
)

(defun nomConflit (conflit)
    (car conflit)
)

(defun allies (conflit)
    (car (cadddr conflit))
)

(defun ennemis (conflit)
    (cadr (cadddr conflit))
)

(defun lieu (conflit)
    (nthcdr 4 conflit)
)
```

C.

Nous réutilisons ici les fonctions créées à la question précédente. Nous utilisons *'dolist'* pour parcourir tous les conflits, et des conditions *'if'* qund nous recherchons un élément en particulier.

```lisp
(defun FB1 (conflits)
    (dolist (x conflits NIL)
        (print (nomConflit x))
    )
)

;(FB1 BaseTest)

(defun FB2 (conflits)
  (dolist (x conflits NIL)
    (if (not (equal (member "Royaume franc" (allies x) :test #'string=) NIL))
      (print (nomConflit x))
      NIL
    )
  )
)

;(FB2 BaseTest)
```

```
(defun FB3 (conflits a)
  (dolist (x conflits NIL)
    (if (not (equal (member a (allies x) :test #'string=) NIL))
      (print (nomConflit x))
      NIL
    )
  )
)

;(FB3 BaseTest "Royaume franc")

(defun FB4 (conflits)
  (dolist (x conflits NIL)
    (if (equal 523 (dateDebut x))
      (print (nomConflit x))
      NIL
    )
  )
)

;(FB4 BaseTest)

(defun FB5 (conflits)
  (dolist (x conflits NIL)
    (if (and (>= (dateDebut x) 523) (<= 715 (dateDebut x)))
      (print (nomConflit x))
      NIL
    )
  )
)

;(FB5 BaseTest)
```

Dans la fonction FB6, nous créons un compteur avec *'setq'* que nous incrémentons ensuite avec *'incf'*. Nous affichons celui-ci en fin de fonction.

```
(defun FB6 (conflits)
  (setf result 0)
  (dolist (x conflits NIL)
    (if (not (equal (member "Lombards" (ennemis x) :test #'string=) NIL))
      (incf result)
      NIL
    )
  )
  (print result)
)

;(FB6 BaseTest)
```