Forest Thomas

Quintaneiro Raphaël

14/10/2022

TP01

IA01

utc
Université de Technologie
Compiègne

Sommaire

Introduction

Ce TP se compose de quatre exercices.

Tout d'abord une mise en condition qui se compose d'exercices d'application basique en langage LISP.

Le second s'articule autour de l'utilisation de la fonction mapcar et de l'utilisation des expressions lambda.

Le troisième permet de manipuler des listes d'association, nous créons ici plusieurs fonctions de base pour manipuler ces listes particulières.

Enfin, le dernier exercice traite de la gestion d'une base de connaissance. Ce dernier peut s'assimiler à un problème dans lequel nous devons créer et interroger un ensemble de données concernant les guerres de France.

Afin de nous permettre de réaliser correctement ce TP, nous utiliserons les connaissances vues en cours et en TD mais nous pourrons aussi nous appuyer sur de la documentation externe, notamment la documentation fournie par common-lisp.net.

Ce compte rendu permet d'expliquer le raisonnement et les points de détail de nos fonctions LISP fournies en annexe.

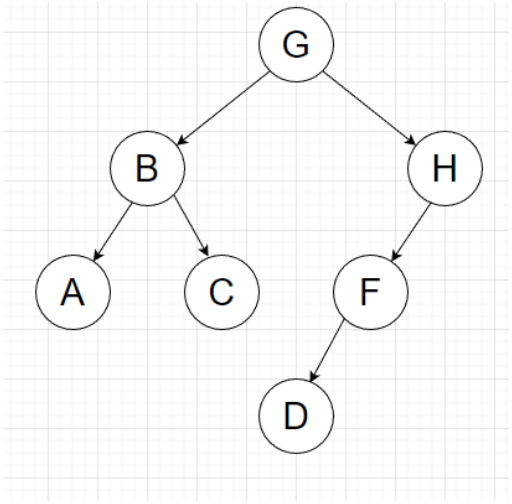
Exercice 1 : Mise en condition

1.

35 -> **entier**
(35) -> **liste**
(((3) 5) 6) -> **liste imbriquée**
-34RRRR -> **atome**
T -> **booléen (vrai)**
NIL -> **booléen (faux)**
() -> **liste vide**

2.

En prenant en compte qu'une liste contient 3 éléments : celui du milieu étant un nœud possédant comme fils gauche le premier élément et comme fils droit le dernier élément de la liste, alors nous obtenons la représentation sous forme d'arbre ci-dessous. L'objet le plus profond de cet arbre est donc « D ».



3.

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

4.

- La fonction ***nombres3*** retourne *BRAVO* si les 3 premiers éléments de la liste *L* sont des nombres, sinon perdu.

```
(defun nombres3 (L)
              (if (and (numberp (car L)) (numberp (cadr L)) (numberp (caddr L)))
                  'BRAVO
                'PERDU))
```

⇨ Pour chacun des 3 premiers éléments de la liste L, on vérifie que c'est bien un nombre grâce au prédicat ***numberp***.
⇨ Input : (nombres3 '(1 2 3 R S 4))
⇨ Output : BRAVO

- La fonction ***grouper*** prend 2 listes *L1* et *L2* en paramètre et retourne la liste composée des éléments successifs de ces deux listes.

```
(defun grouper (L1 L2)
              (if (and (car L1) (car L2))
                  (cons (cons (car L1) (list (car L2))) (grouper (cdr L1) (cdr L2)))
                NIL))
```

⇨ On commence par vérifier que le premier élément de chaque liste existe. Une fois cette condition validée, on peut construire la liste récursivement : on créer d'abord une liste avec

le premier élément de *L2* avec laquelle on construit la paire voulue puis on construit le reste de la liste en rappelant la fonction avec le reste des deux listes *L1* et *L2*.
Une manière beaucoup plus simple de résoudre ce problème est d'utiliser la fonction ***mapcar*** qui va appliquer la fonction ***list*** successivement sur le *i*-ème élément de chaque liste :

```
(defun grouper (L1 L2)
  (mapcar #'list L1 L2))
```

⇨ Input : (grouper '(1 2 3) '(4 5 6))
⇨ Output : ((1 4)(2 5)(3 6))

- La fonction ***monReverse*** retourne la liste inversée de la liste *L* passée en paramètre.

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

⇨ On vérifie que le premier élément de la liste n'est pas ***NIL***. Si c'est le cas, on ajoute celui-ci à la fin de l'inverse du reste de la liste (procédure récursive).
⇨ Input : (monReverse '(1 2 3 4 5))
⇨ Output : (5 4 3 2 1)

- La fonction ***palindrome*** retourne vrai si la liste *L* est un palindrome (y compris si *L* est vide).

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

⇨ On vérifie simplement l'égalité entre la liste L et son inverse calculé par la fonction précédente.
⇨ Input : (palindrome '(x a m a x))
⇨ Output : T

Exercice 2 : Objets fonctionnels

- La fonction ***list-triple-couple*** retourne la liste des couples composées des éléments de la liste *L* fournie en paramètre et de leur triple.

```
(defun list-triple-couple (L)
               (mapcar #'(lambda (x) (list x (* x 3))) L))
                       )
```

⇨ Grâce à la fonction ***mapcar***, on va appliquer la fonction ***lambda*** sur chaque élément de la liste *L* pour créer des couples composés de l'élément et de son triple.

Exercice 3 : a-list

- La fonction ***my-assoc*** retourne *nil* si *cle* ne correspond à aucune clé de la liste d'association, la paire correspondante dans le cas contraire.

```
(defun my-assoc (cle a-list)
               (if (car a-list)
                   (if (eq cle (caar a-list))
                       (car a-list)
                     (my-assoc cle (cdr a-list)))
                 NIL))
```

⇨ Le principe est de parcourir la liste récursivement jusqu'à ce que l'on trouve la clé ou que l'on arrive à la fin de la liste. On commence alors par vérifier que la liste n'est pas vide puis on vérifie l'égalité entre la clé du premier élément de la liste et la clé recherchée. Si le résultat est positif, on retourne la paire correspondante à l'utilisateur, sinon on continue la recherche en rappelant la fonction avec le reste de la liste.
⇨ Input : (my-assoc 'Pierre '((Yolande 25) (Pierre 22) (Julie 45)))
⇨ Output : (Pierre 22)

- La fonction ***cles*** retourne la liste des clés de la *a-list* passée en paramètre.

```
(defun cles (a-list)
    (mapcar #'car a-list))
```

⇨ Ici, nous avons simplement à appliquer la fonction ***car*** sur chaque élément de la liste *a-list*.
⇨ Input : (cles '((Yolande 25) (Pierre 22) (Julie 45)))
⇨ Output : (Yolande Pierre Julie)

- La fonction ***creation*** retourne une A-liste à partir d'une liste de clés et d'une liste de valeurs.

```
(defun creation (listeCles listeValeurs)
  (mapcar #'list listeCles listeValeurs))
```

⇨ La fonction ***creation*** à exactement le même but que la fonction ***grouper*** de l'exercice 1.
⇨ Input : (creation 'Yolande Pierre Julie) '(25 22 45))
⇨ Output : ((Yolande 25)(Pierre 22)(Julie 45))

Exercice 4 : Gestion d'une base de connaissances en 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)
  (car (cddddr conflit)))
```

⇨ Les fonctions ci-dessus utilises simplement les fonctions ***car***/***cdr*** pour naviguer convenablement dans la *BaseTest.*

Pour les fonctions ***FB1*** à ***FB6***, nous allons d'abord présenter une solution récursive car c'est celle qui nous paraissait la plus instinctive. Néanmoins, nous verrons par la suite que certaines fonctions peuvent être plus efficace lorsqu'elles sont réalisées de manière itérative.

- ***FB1***

```
(defun FB1 (Liste)
            (if (car Liste)
                (progn
                  (print (car Liste))
                  (FB1 (cdr Liste))
                  )))
```

⇨ On affiche simplement la Liste des conflits passée en argument.

Toutes les fonctions proposées ci-dessous auront le même principe pour parcourir la liste de manière récursive. On commencera par vérifier que le ***car*** de la *Liste* existe (condition de fin) puis après avoir effectué les instructions nécessaires, on rappellera la fonction avec le ***cdr*** de la *Liste*.

- ***FB2***

```
(defun FB2 (Liste)
                (if (car Liste)
                  (progn
                    (let ((rf "Royaume Franc"))
                      (if (or (member rf (allies (car Liste)) :test #'equal) (member rf (ennemis (car Liste)) :test #'equal)) ;
                          (print (car Liste)))
                      (FB2 (cdr Liste))
                      ))))
```

⇨ On définit une variable locale *rf* puis on vérifie pour chaque conflit si *rf* fait partie de la liste des alliés OU des ennemis grâce à la fonction ***member*** ainsi qu'à un test ***equal***. Si c'est le cas, on l'affiche, sinon on passe au suivant.

- ***FB3***

```
(defun fb3 (Liste ally)
               (if (car Liste)
                     (if (member ally (allies (car Liste)) :test #'equal)
                         (cons (car Liste) (FB3 (cdr Liste) ally))
                       (FB3 (cdr Liste) ally))
                     ))
```

⇨ Même principe que la fonction ***FB2***, la variable *ally* est ici simplement passée en paramètre et au lieu d'afficher le conflit lorsque *ally* fait partie des alliés, on construit la liste composée du conflit courant ainsi que du reste de la liste. Si ce n'est pas le cas, on passe simplement à l'élément suivant.

⇨ Input : (FB3 BaseTest "Neustrie")

⇨ Output : (("Guerre civile des Francs" 715 719 (("Neustrie") ("Austrasie")) ("Royaume Franc")))

- ***FB4***

```
(defun FB4 (Liste)
               (if (car Liste)
                   (let ((date 523))
                     (if (= date (dateDebut (car Liste)))
                         (car Liste)
                       (FB4 (cdr Liste)))
                     )))
```

⇨ Pour chaque conflit de la *BaseTest*, on vérifie si la date de début est 523. Si c'est le cas on le retourne, sinon on passe au conflit suivant.

⇨ Input : (FB4 BaseTest)

⇨ Output : ("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume des Burgondes")) ("Vezeronce" "Arles"))
NIL

- ***FB5***

```
(defun FB5 (Liste)
                (if (car Liste)
                      (let ((actual_date (dateDebut (car Liste))))
                        (if (and (>= actual_date 523) (<= actual_date 715))
                            (cons (car Liste) (FB5 (cdr Liste)))
                          (FB5 (cdr Liste)))
                        )))
```

⇨ Pour chaque conflit, on récupère la date de début et on vérifie si celle-ci est compris entre 523 et 715. Si c'est le cas on construit la liste composée de l'élément courant et des éléments du reste de la liste, sinon on passe simplement au prochain conflit.

⇨ Input : (FB5 BaseTest)

⇨ Output : (("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume des Burgondes")) ("Vezeronce" "Arles"))
("Conquête de la Thuringe" 531 531 (("Royaume Franc") ("Thuringes")) ("Thuringe"))
("Guere des Goths" 535 553 (("Royaume ostrogoth") ("Empire byzantin")) ("Péninsule italienne"))
("Conquête de l'Alémanie" 536 536 (("Royaume Franc") ("Alemans")) ("Alémanie"))
("Conquête de la Bavière" 555 555 (("Royaume Franc") ("Bavarii")) ("Bavière"))
("Campagnes de Bretagne" 560 578 (("Royaume Franc") ("Royaume du Vannetais")) ("Vannetais"))
("Guère franco-frisonne" 600 793 (("Royaume Franc") ("Royaume de Frise")) ("Pays-bas" "Allemagne"))
("Guerre civile des Francs" 715 719 (("Neustrie") ("Austrasie")) ("Royaume Franc")))

- ***FB6***

```
(defun FB6 (Liste)
                (if Liste
                    (progn
                      (let ((ennemy "Lombards"))
                        (if (member ennemy (ennemis (car Liste)) :test #'equal)
                            (+ 1 (FB6 (cdr Liste)))
                          (FB6 (cdr Liste)))
                        ))
                  0))
```

⇨ ***FB6*** est similaire à ***FB2***. Ici, si *ennemy* fait partie de la liste des ennemis d'un conflit, on ajoute 1 au calcul global, sinon 0.

⇨ Input : (FB6 BaseTest)

⇨ Output : 2

Maintenant, voici une solution itérative pour chacune des fonctions ***FB1*** à ***FB6*** :

```
(defun FB1 (Liste)
  (dolist (l Liste)
    (print l)))

(defun FB2 (Liste)
               (let ((rf "Royaume Franc"))
                 (dolist (l Liste)
                   (if (or (member rf (allies l) :test #'equal) (member rf (ennemis l) :test #'equal))
                       (print l))
                    )))

(defun FB3 (Liste ally)
               (loop for l in Liste
                     unless (NULL (member ally (allies l) :test #'equal))
                   collect l
                     ))

(defun FB4 (Liste)
                (let ((date 523))
                  (loop for l in Liste
                     unless (/= date (dateDebut l))
                   collect l
                     )))

(defun FB5 (Liste)
                  (loop for l in Liste
                     unless (or (< (dateDebut l) 523) (> (dateDebut l) 715))
                   collect l
                     ))

(defun FB6 (Liste)
                (let ((ennemy "Lombards") (x 0))
                  (dolist (l Liste)
                    (if (member ennemy (ennemis l) :test #'equal)
                        (incf x))
                    )
                  x))
```

⇨ On peut remarquer que ces fonction semblent plus concises, notamment lorsqu'il faut afficher des éléments comme dans la fonction ***FB1***. Dans les fonctions ***FB3***, ***FB4*** et ***FB5***, nous avons utilisé la boucle ***for*** ainsi que ***unless*** et ***collect*** qui permettent de récupérer les éléments de la liste sur laquelle la boucle est effectué à moins qu'une certaine condition soit vérifiée.

Conclusion

Ce TP nous a permis de retravailler les concepts vus en cours et en TD. Nous avons dû comme précisé avant ajouter à ces connaissances des documents externes ainsi que des forums notamment celui de www.developpez.net.

L'ensemble des exercices a été traités dans leur intégralité et le code LISP associé à ce documents se trouve dans l'annexe fournis avec ce document.