Rapport TP1

Montée en compétences Lisp

ALLIANCE SORBONNE UNIVERSITÉ utc

Table des matières

1 Introduction

Nous avons été amenés à travailler sur le premier TP de IA01 durant les mois de septembre et octobre 2022. Ce premier TP est une mise en pratique des premières notions vues en cours. Il porte principalement sur les types, les fonctions et la manipulation de liste à travers celles-ci.

Ce rapport contient nos fonctions (sans les commentaires que nous avons uniquement mis dans les fichiers *Lisp*), l'explication des fonctions, nos choix vis-à-vis du sujet, notre compréhension de l'énoncé et des fonctions alternatives.

Les différentes fonctions, jeux de données de test et commentaires explicatifs sont présents dans les fichiers suivants :

- *TP1-EX1.cl* : pour l'exercice 1
- *TP1-EX2.cl* : pour l'exercice 2 et les fonctions de l'exercice 1 utilisées
- *TP1-EX3.cl* : pour l'exercice 3 et les fonctions de l'exercice 1 et 2 utilisées
- *TP1-EX4.cl* : pour l'exercice 4 et les fonctions de l'exercice 1, 2 et 3 utilisées

2 Exercice 1 : Mise en condition

2.1 Déterminez le type des objets *Lisp* suivant :

L'objectif de cet exercice est de bien comprendre les différents types d'objets en *Lisp*. Nous avons cherché à être les plus précis possible sur chacun des types.

```
35
(35)
(((3) 5) 6)
-34RRRR
T
NIL
()
```

FIGURE 1 – Différents types d'objets en *Lisp*

Nous avons déterminé les types suivants pour chaque objet :

- 35 : atome de type nombre
- (35) : liste contenant un atome de type nombre et de profondeur 0
- (((3) 5) 6) : liste contenant 3 éléments de type nombre et de profondeur 2
- -34RRRR : symbole
- T et NIL : symboles : ce sont les deux constantes de *Lisp*
- () : liste vide évaluée à NIL

2.2 Traduisez sous forme d'arbre la liste suivante :

```
(((A)(B)(C)) G (((((D)) F) H)))
```

FIGURE 2 – Arbre sous forme de liste en *Lisp*

L'objectif de cette question est de nous faire comprendre la notion de liste et d'arbre défini par une liste. Ainsi, nous avons pour cette question une liste que nous allons transformer en arbre. Pour cela, nous utilisons la méthode vue en cours permettant de décrire une liste sous forme d'arbre :

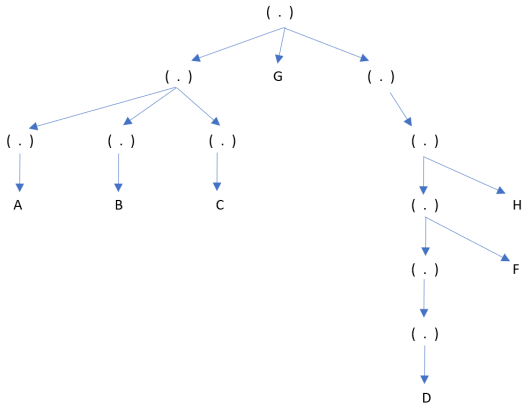

FIGURE 3 – Schéma de l'arbre de la liste

L'objet le plus profond est donc D et a une profondeur de 5.

2.3 Que font les appels de fonctions suivants :

L'objectif de cette question est de nous faire comprendre le fonctionnement des fonctions CAR, CDR et CONS. Nous avons expliqué les différentes étapes et la décomposition étape par étape de ces fonctions dans le fichier de l'exercice 1.
Pour résumer, nous avons compris que :

- (CAR liste) : retourne le premier élément
- (CDR liste) : retourne la liste sans son premier élément
- (CONS liste1 liste2) : retourne une liste dont le CAR est liste1 et le CDR les éléments de liste2 donc :
  - si liste1 est un atome, il est juxtaposé au reste de la liste
  - si liste1 est une liste, elle est laissée entre () et juxtaposé au reste de la liste
  - les éléments de liste2 sont ajoutés à la liste

2.4 Écrire les fonctions suivantes :

2.4.1 nombre3

Cette fonction retourne BRAVO si les 3 premiers éléments de la liste L sont des nombres et PERDU sinon. Nous comprenons ainsi que tous les cas ne vérifiant pas la condition : *avoir 3 nombres en début de liste*, vont retourner PERDU.

Ainsi les cas suivants vont retourner PERDU :

- une liste de taille inférieure à 3
  - liste de taille 2 : (1 2)
  - liste de taille 1 : (1)
  - liste vide : ()
- un élément autre qu'une liste est passée en paramètre

Notre fonction vérifie que l'objet passé en paramètre est bien une liste à l'aide de (listp obj), que sa taille est supérieure ou égale à trois à l'aide de (>= (length obj) 3). Enfin, nous vérifions que les trois premiers éléments sont des nombres à l'aide de (numberp (CAR obj)).

```
(defun nombre3 (l)
  (if (AND (listp l)
           (>= (length l) 3)
           (numberp (car l))
           (numberp (cadr l))
           (numberp (caddr l)))

      'BRAVO
      'PERDU
  )
)
```

FIGURE 4 – Fonction nombre3

Le test de la taille de la liste supérieure à 3 est facultatif puisque les prédicats (numberp NIL) aurait retourné NIL et la fonction aurait retourné PERDU.

```
(defun nombre3 (l)
  (if (AND (listp l)
           (numberp (car l))
           (numberp (cadr l))
           (numberp (caddr l)))

      'BRAVO
      'PERDU
  )
)
```

FIGURE 5 – Fonction nombre3 - sans length

2.4.2 grouper

Cette fonction prend en paramètre deux listes et retourne une liste composée de listes. Dans cette liste de retour, le i^e élément correspondra à une liste composée du i^e élément de la première liste et du i^e élément de la deuxième liste. Si une des listes n'a pas de i^e élément, alors on ne prend que le i^e élément de la liste qui en a un.

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

FIGURE 6 – Fonction grouper

2.4.3 monReverse

Cette fonction inverse une liste de façon récursive en ajoutant à la fin d'une liste son premier élément, puis en avant-dernière position son deuxième élément, etc. Les appels récursifs se font sur le cdr de la liste, car pour un appel donné, on traite le premier élément de la liste donnée en paramètre.

Nous avons codé cette fonction en deux versions. La première vérifie la taille de la liste pour économiser un appel récursif dans le cas où la taille de la liste serait de 2. Nous nous sommes par la suite rendus compte que cette vérification n'était pas vraiment une optimisation, car bien qu'on économise un appel récursif dans le cas ou la taille de la liste est 2, on perd le temps de faire la vérification dans tous les cas ou la liste n'est pas de taille 2. Par exemple, pour une liste de taille 50, on fait 48 fois la vérification sur la taille de la liste, et on gagne 1 appel récursif lorsqu'on arrive à une liste de taille 2.

Nous avons donc codé une version alternative qui n'effectue pas cette vérification de taille. Cette nouvelle fonction optimise donc le nombre d'instructions exécutées par appel et au total.

```
(defun monReverse(L)
    (if (>= (length L) 3)
        (append (monReverse (cdr L)) (list (car L)))
        (if (= (length L) 2)
            (append (cdr L) (list (car L)))
            L
        )
    )
)
```

FIGURE 7 – Fonction monReverse - version récursive avec vérification

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

FIGURE 8 – Fonction monReverse - version récursive sans vérification

2.4.4 palindrome

Ici, l'objectif est de comparer les éléments, un par un, de chaque extrémité d'une liste. Nous avons développé deux fonctions. Nous avons également codé deux fonctions.

La première fonction utilise le résultat de la monReverse et le compare à la liste initiale. La seconde fonction compare la première lettre à la dernière, puis fais appel à elle-même avec le mot dénué de sa première lettre et de sa dernière lettre. La condition d'arrêt de cette fonction récursive est : *le mot ne fasse qu'une ou 0 lettre.*

Même pour cette deuxième version, nous devons utiliser la fonction primitive (reverse list). La raison est que bien qu'en *Lisp*, on puisse sélectionner le premier et dernier élément d'une liste avec (cdr list) et (last list), on ne peut pas sélectionner une liste sans son premier élément et dernier élément, sans l'inverser (l'équivalent python serait liste[1 :-1]). Nous prenons donc le cdr de la liste inversée (liste sans son dernier élément), nous remettons la liste à l'endroit, et nous prenons son cdr (liste sans son premier élément).

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

FIGURE 9 – Fonction palindrome - version 1 avec comparaison avec le mot inverse

```
(defun palindrome (L)
  (if (or (equal (length L) 1) (equal (length L) 0))
    T
    (if (equal (car L) (car (last L)))
      (palindrome (cdr (reverse (cdr (reverse l)))))
      NIL
    )
  )
)
```

FIGURE 10 – Fonction palindrome - version 2 avec comparaison lettre par lettre

3 Exercice 2 : Objets fonctionnels

L'objectif de cet exercice est de voir comment utiliser mapcar avec des fonctions anonymes, des expressions lambda :

— mapcar est une instruction utilisée pour appliquer une opération sur chaque élément d'une liste (son car, puis son cadr, puis son caddr, etc).
— lambda est utilisé pour définir une fonction anonyme, (il n'est pas toujours nécessaire de donner un nom à une fonction). Les fonctions anonymes servent surtout pour définir des fonctions de service utilisées temporairement à l'intérieur d'autres fonctions.

Le travail demandé est d'écrire une fonction, *list-triple-couple*, qui prend en paramètre une liste et retourne une liste de couples comprenant les éléments de la liste ainsi que leur triple.

La fonction est composée d'une fonction anonyme qui calcule le triple d'un élément (lambda (x) (* x 3)). Ensuite, on utilise mapcar sur cette fonction et sur la liste pour calculer le triple de chaque élément et les récupérer dans une liste. Enfin, on se sert de grouper de l'exercice 1 pour faire des couples entre la liste d'éléments et la liste de leurs triples.

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

FIGURE 11 – Fonction list-triple-couple

4 Exercice 3 : a-list

Écrivez les trois fonctions décrites ci-dessous :

4.1 my-assoc (cle a-list)

Cette fonction retourne NIL si cle ne correspond à aucune clé de la liste d'association, la paire correspondante dans le cas contraire ; cette fonction existe sous le nom de assoc.

Dans cette fonction, on va vérifier si la liste est non vide (ceci sera la condition d'arrêt de la récursivité), on vérifie ensuite si le car du premier élément correspond à notre clé, si oui, on renvoie cet élément, si non, on effectue une recherche dans le reste de la liste.

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

FIGURE 12 – Fonction my-assoc

4.2 cles (a-list)

Cette fonction retourne la liste des clés d'une A-liste, nous l'avons codée en deux versions.

La première version utilise la récursivité, elle prend le car du car de la liste (soit la première clé), puis fait un appel récursif avec le cdr de la liste pour récupérer les autres clés. Pour la condition d'arrêt, on vérifie simplement si la liste n'est pas vide.

```
(defun cles (a-list)
  (if a-list
    (append (list (car (car a-list))) (cles (cdr a-list)))
    NIL
  )
)
```

FIGURE 13 – Fonction cles - version récursive

La deuxième version est itérative et plus simple. Nous nous sommes rendus compte qu'il n'était pas nécessaire de faire tout ce que nous avons fait dans la

première version et qu'utiliser un mapcar pour récupérer le car de chaque élément était suffisant.

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

FIGURE 14 – Fonction cles - version itérative

4.3 creation (listeCles listeValeurs)

Cette fonction retourne une A-liste à partir d'une liste de clés et d'une liste de valeurs. Le principe est simple : on compare la taille des listes, si les deux tailles sont égales, alors on fait appel à la fonction grouper de l'exercice 1 pour faire les couples, et si ce n'est pas le cas, on ne peut pas faire de a-liste et on renvoie NIL.

```
(defun creation (listeCles listeValeurs)
  (if (equal (length listeCles) (length listeValeurs))
    (grouper listeCles listeValeurs)
    NIL
  )
)
```

FIGURE 15 – Fonction creation

5 Exercice 4 : gestion d'une base de connaissances en Lisp

On considère une base de connaissances sur les guerres de France définies par les propriétés suivantes : nom du conflit, date de début de conflit, date de fin de conflit, belligérants (alliés y compris la France, ennemis), lieu).

Soit la base BaseTest suivante comprenant un seul conflit :

```
(setq BaseTest
'( ("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume des
Burgondes")) ("Vezeronce"
"Arles"))))
```

5.1 Compléter BaseTest avec les conflits commençant avant l'an 1100.

Nous pouvons déduire de l'énoncé que le format attendu et standardisé pour chaque conflit est le suivant :

```
("nom de la guerre" DateDebut DateFin ((liste allies) (liste
ennemis)) (liste lieux))
```

Cette partie est faite dans le fichier Lisp. Nous avons pris la décision de saisir tous les conflits par lieux et ennemis afin de ne pas perdre l'information de quel ennemi s'est battu dans quel lieu. Le sujet ne répondait pas à cette question, une perte d'information ne nous semblait pas judicieuse. De plus, nous avons pris la liberté d'enlever les différents signes de ponctuation (tel que les - entre les mots) et les différents accents afin de ne pas bloquer une quelconque recherche telle que celle possible dans *FB3*.

5.2 Définir les fonctions de service :

5.2.1 dateDebut (conflit)

Cette fonction retourne la date de début du conflit passé en argument. Cela correspond au deuxième élément, donc au cadr.

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

FIGURE 16 – Fonction dateDebut

5.2.2 nomConflit (conflit)

Cette fonction retourne le nom du conflit passé en argument. Cela correspond au premier élément, donc au car.

```
(defun nomConflit (conflit)
  (car conflit)
)
```

FIGURE 17 – Fonction nomConflit

5.2.3 allies (conflit)

Cette fonction retourne les alliés du conflit passé en argument. Cela correspond au premier élément du quatrième élément, donc au car du cadddr.

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

FIGURE 18 – Fonction allies

5.2.4 ennemis (conflit)

Cette fonction retourne les ennemis du conflit passé en argument. Cela correspond au deuxième élément du quatrième élément, donc au cadr du cadddr.

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

FIGURE 19 – Fonction ennemis

5.2.5 lieu (conflit)

Cette fonction retourne le lieu du conflit passé en argument. Cela correspond au cinquième élément, donc au car du cddddr.

```
(defun lieu (conflit)
  (car (cddddr conflit))
)
```

FIGURE 20 – Fonction lieu

5.2.6 dateFin (conflit)

Cette fonction retourne la date de fin du conflit passé en argument. Cela correspond au troisième élément, donc au car du caddr. Cette fonction ne faisait pas partie des fonctions demandées, mais était la seule manquante pour obtenir les différentes informations d'un conflit.

```
(defun dateFin (conflit)
    (caddr conflit)
)
```

FIGURE 21 – Fonction dateFin

5.3 Définir les fonctions suivantes :

Pour les fonctions suivantes, nous avons souhaité ajouter une gestion d'erreur en vérifiant les différents paramètres. Ainsi, pour chacune des fonctions, nous vérifions son type à l'aide des prédicats suivants :

- (listp list) : vérifiant que l'objet est une liste
- (string str) : vérifiant que l'objet est une chaîne de caractères
- obj : vérifiant que l'objet n'est pas NIL

5.3.1 FB1

Cette fonction affiche tous les conflits. On utilise mapcar sur print pour afficher chaque élément de la base, et donc chaque conflit.

```
(defun FB1 (base_test)
  (if (and base_test (listp base_test))
    (mapcar 'print base_test)
    NIL
  )
)
```

FIGURE 22 – Fonction FB1 - version 1

Nous avons également codé une fonction utilisant mapcar et les différentes fonctions codées précédemment. Cette fonction va donc en pratique créer une copie de la BaseTest et est donc moins approprié, car va occuper plus d'espace mémoire qu'un simple parcours avec affichage.

```
(defun FB1 (base_test)
  (if (and base_test (listp base_test))
    (mapcar #'(lambda (conflit)
                (list
                 (nomConflit conflit)
                 (dateDebut conflit)
                 (dateFin conflit)
                 (allies conflit)
                 (ennemis conflit)
                 (lieu conflit)
                 )
                )
    base_test)
    NIL
  )
)
```

FIGURE 23 – Fonction FB1 - version 2

5.3.2 FB2

Cette fonction affiche les conflits du "Royaume Franc". On effectue pour cela une double boucle, une qui va parcourir la base conflit par conflit, et une autre qui va parcourir les alliés d'un conflit donné. Si le conflit contient "Royaume Franc", on l'affiche.

```
(defun FB2 (base_test)
  (if (and base_test (listp base_test))
    (mapcar #'(lambda (conflit)
                (dolist (allie (allies conflit))
                  (if (equal allie "Royaume franc")
                    (print conflit)
                  )
                )
              )
    base_test)
    NIL
  )
)
```

FIGURE 24 – Fonction FB2

5.3.3 FB3

Cette fonction retourne la liste des conflits dont un allié est précisé en argument. Cette fonction fonctionne sous le même principe que la précédente avec le principe de la double boucle. Les différences sont :

- l'allié recherché n'est plus forcément "Royaume franc" mais peut être n'importe quel allié passé en argument
- on nous demande de retourner une liste des conflits correspondants, et non pas d'uniquement les afficher.

On initialise dont une liste vide pour y ajouter les conflits correspondants au fur et à mesure. On inverse cette liste à la fin, car cette dernière est dans l'ordre anti-chronologique, étant donné que l'instruction cons ajoute un élément en tête de liste.

```
(defun FB3 (base_test allie_rec)
  (if (and base_test (listp base_test) (stringp allie_rec))
    (let ((liste_conflits NIL))
      (mapcar #'(lambda (conflit)
        (dolist (allie (allies conflit))
          (if (equal allie allie_rec)
            (setq liste_conflits (cons conflit liste_conflits))
          )
        )
      )
      base_test)

      (reverse liste_conflits)
    )
    NIL
  )
)
```

FIGURE 25 – Fonction FB3

5.3.4 FB4

Cette fonction retourne le conflit dont la date de début est 523. Pour cela, on crée une copie de la base. On va ensuite vérifier l'égalité entre la date de début de son premier élément (à l'aide de la fonction dateDebut définie plus haut), si la date ne correspond pas, on supprime le premier élément de la copie de la base et on continue la recherche en recommençant avec le nouveau premier élément. Si la date correspond, on retourne le premier élément de la copie de la base (celui qui vient d'être traité et qui correspond).

```
(defun FB4 (base_test)
  (if (and base_test (listp base_test))
    (let ((copie_base base_test))
      (loop
        (if (not (equal (dateDebut (car copie_base)) 523))
              (pop copie_base)
              (return (car copie_base))
        )
      )
    )
    NIL
  )
)
```

FIGURE 26 – Fonction FB4

5.3.5 FB5

Cette fonction retourne la liste des conflits dont la date de début est comprise entre 523 et 715. Cette fonction fonctionne sous le même principe que la précédente avec le principe de vérification de date. La différence est qu'ici, on recherche un intervalle et non une date précise. De plus, on a ici plusieurs conflits qui vont correspondre à la recherche, on retournera donc une liste de conflits. Pour les mêmes raisons que pour la fonction FB3, on inverse la liste à la fin.

```
(defun FB5 (base_test)
  (if (and base_test (listp base_test))
    (let ((liste_conflits NIL))
      (mapcar #'(lambda (conflit)
                  (if (and (>= (dateDebut conflit) 523) (<= (
dateDebut conflit) 715))
                    (setq liste_conflits (cons conflit
liste_conflits))
                  )
                )
      base_test)

      (reverse liste_conflits)
    )
    NIL
  )
)
```

FIGURE 27 – Fonction FB5

5.3.6 FB6

Cette fonction calcule et retourne le nombre de conflits ayant pour ennemis les "Lombards". Cette fonction fonctionne comme FB2, mis à part la recherche qui n'est plus "Royaume Franc" dans les alliés, mais "Lombards" dans les ennemis. La deuxième différence est qu'au lieu d'afficher les conflits correspondants, il faut seulement retourner le nombre de conflits correspondants. Pour cela, nous avons codé deux fonctions. Dans la première, on incrémente un compteur à chaque fois que l'on trouve "Lombards" dans les ennemis d'un conflit.

```
(defun FB6 (base_test)
  (if (and base_test (listp base_test))
    (let ((compteur 0))
      (mapcar #'(lambda (conflit)
        (dolist (ennemi (ennemis conflit))
          (if (equal ennemi "Lombards")
            (setq compteur (+ compteur 1))
          )
        )
      )
      base_test
      )

      compteur
    )
    NIL
  )
)
```

FIGURE 28 – Fonction FB6 - avec compteur

Dans la seconde fonction, nous utilisons la fonction primitive (length list) sur une liste que nous créons dans un let et que nous augmentons de chaque conflit ayant *Lombards* dans les ennemis à l'aide de la fonction (append list1 list2).

```
(defun FB6 (base_test)
  (if (and base_test (listp base_test))
    (let ((liste_lombard NIL))
       (length
          (dolist (conflit base_test liste_lombard)
            (dolist (ennemi (ennemis conflit))
              (if (equal ennemi "Lombards")
                  (setq liste_lombard (append liste_lombard (list
conflit)))
              )
            )
          )
      )
    )
    NIL
  )
)
```

FIGURE 29 – Fonction FB6 - avec taille de liste

6 Conclusion

Le premier TP nous a permis de mettre en application les notions vues en cours et en TD sur une nouvelle problématique plus large et donc où les choix d'implémentations dépendent de l'interprétation du sujet.

Nous avons grâce à ce TP appris à faire fonctionner les différents blocs suivant :

- (lambda () ()) : la fonction anonyme *lambda*
- (mapcar #'() list) : la fonction *mapcar*, qui est un des différents blocs itératif, itérant sur le *car* de la liste
- (loop ...) : la fonction *loop*, qui est un des différents blocs itératif
- (dolist (. . .) ...) : la fonction *dolist*, qui est un des différents blocs itératif
- (let ((. .)) ...) : la fonction *let* permettant de créer une variable locale

Table des figures