IA01 : Intelligence Artificielle - Représentation des connaissances

TP1 : Montée en compétences Lisp

ASSAF Nora & GAJAN Antoine

ALLIANCE SORBONNE UNIVERSITÉ utc

Table des matières

Introduction

Ce TP a pour but de nous familiariser avec le langage Lisp. Au cours de celui, les connaissances acquises lors des premiers cours magistraux et travaux dirigés seront mobilisées pour monter en compétences.

Composé de 4 exercices, ce TP abordera des notions transversales : boucle d'interaction, fonctions, fonctions anonymes, mapping, variables et récursivité.

Ce rapport présentera la démarche entreprise pour effectuer ce mini-projet ainsi que les limites rencontrées.

1 Exercice 1 : Mise en condition

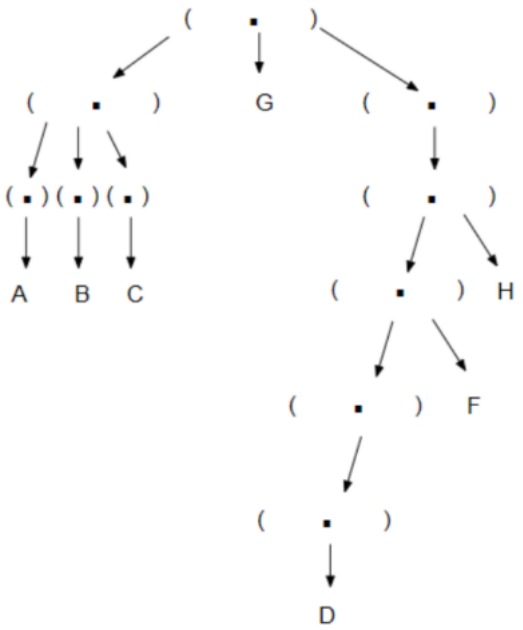
1.1 Déterminer le type des objets Lisp suivants

Il s'agit dans cette question de déterminer le type de différents objets Lisp. Grâce aux connaissances du cours, nous obtenons aisément que :

```
35 ; est un nombre (de type number)
(35) ; est une liste (de type list), de profondeur 0
(((3)5)6) ; est une liste (de type number), de profondeur 2
-34RRR ; est un symbole (de type symbol)
T ; est un symbole (de type number), et plus specifiquement une des 2
    constantes Lisp qui signifie "valeur non nulle" autrement dit "true
    "
NIL ; est un symbole, et plus precisement une des 2 constantes Lisp
() ; est une liste vide evaluee a NIL
```

1.2 Traduisez sous forme d'arbre la liste suivante :

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



La première parenthèse correspondant à une profondeur de 0, on en déduit que l'objet le plus profond est D. Sa profondeur est de 5.

1.3 Que font les appels de fonctions suivants ?

Cette question a pour objectif de nous faire comprendre le fonctionnement de certaines primitives de base, telles que car, cdr ou cons. Grâce aux connaissances théoriques du cours et à la pratique en TD, nous pouvons déduire les résultats suivants pour chacune des listes proposées :

```
(CADR (CDR (CDR (CDR '(DO RE MI FA SOL LA SI)))))

; Etape 1 : (CDR'(DO RE MI FA SOL LA SI)) renvoie (RE MI FA SOL LA SI)
; Etape 2 : (CDR (RE MI FA SOL LA SI)) renvoie (MI FA SOL LA SI)
; Etape 3 : CDR (MI FA SOL LA SI)) renvoie (FA SOL LA SI)
; Etape 4 : (CADR (FA SOL LA SI)) renvoie (CAR(CDR '(FA SOL LA SI))), c
    'est a dire (CAR'(SOL LA SI)), donc SOL.

; Ainsi, (CADR (CDR (CDR (CDR '(DO RE MI FA SOL LA SI))))) renvoie SOL.
```

```
(CONS (CADR '((A B)(C D))) (CDDR '(A (B (C)))))

; On sait que (CONS 'a 'b) renvoie une liste avec a comme car et b
    comme cdr.
; De plus, par un raisonnement analogue a celui effectue ci-dessus, on
    obtient :

; Etape 1 : (CADR '((A B)(C D))) renvoie (CAR '((CD))), c'est-a-dire (
    CD)
; Etape 2 : (CDDR '(A (B (C)))) renvoie (CDR '((B(C))), c'est-a-dire,
    NIL (car la liste n'etait composee que d'un seul element).
; Etape 3 : (CONS (CADR '((A B)(C D))) (CDDR '(A (B (C))))) renvoie (
    CONS '(C D)  NIL), c'est-a-dire, ((C D)).

; Ainsi, cela renvoie ((C D)).
```

```
(CONS (CONS 'HELLO NIL) '(HOW ARE YOU))

; Etape 1 : (CONS 'HELLO NIL) renvoie (HELLO)
; Etape 2 :  (CONS '(HELLO) '(HOW ARE YOU)) renvoie ((HELLO) HOW ARE
    YOU)

; Ainsi, cela renvoie ((HELLO) HOW ARE YOU)
```

```
(CONS 'JE (CONS 'JE (CONS 'JE (CONS 'BALBUTIE NIL))))

; Etape 1 : (CONS 'BALBUTIE NIL) renvoie (BALBUTIE)
; Etape 2 : (CONS 'JE '(BALBUTIE)) renvoie (JE BALBUTIE)
; Etape 3 : (CONS 'JE '(JE BALBUTIE)) renvoie (JE JE BALBUTIE).
; Etape 4 : (CONS 'JE  '(JE JE BALBUTIE)) renvoie (JE JE JE BALBUTIE)

; Ainsi, cela renvoie (JE JE JE BALBUTIE)
```

```
(CADR (CONS 'TIENS (CONS '(C EST SIMPLE) ())))

; Etape 1 : (CONS '(C EST SIMPLE) ()) renvoie ((C EST SIMPLE))
; Etape 2 : (CONS 'TIENS '((C EST SIMPLE))) renvoie (TIENS (C EST
    SIMPLE))
; Etape 3 : (CADR '(TIENS (C EST SIMPLE))) renvoie (C EST SIMPLE)

; Ainsi, cela renvoie (C EST SIMPLE)
```

1.4 Ecrire les fonctions suivantes

- nombres3 (L) : retourne BRAVO si les 3 premiers éléments de la liste L sont des nombres, sinon PERDU

Pour réaliser cette fonction, on teste dans une condition "si"/"if" si les 3 premiers éléments de la liste sont des nombres avec numberp. Si la condition est vérifiée, on renvoie BRAVO, sinon on renvoie PERDU.

```
    ;on teste si les 3 premiers elements de la liste sont des nombres
(defun nombre3 (L)
    (if (AND (numberp (car L)) (numberp (cadr L)) (numberp (caddr L)))
        'BRAVO
        'PERDU
    )
)
```

Afin de vérifier que la fonction retourne le résultat escompté, on peut proposer le jeu de données suivant pour tester :

```
(nombre3 '(45 23 1 a b)) ; retourne BRAVO
(nombre3 '(a 12 4 5 6)) ; retourne PERDU
```

- grouper (L1 L2) : retourne la liste composée des éléments successifs des deux listes passées en arguments L1 et L2.

Nous avons considéré qu'il y a 3 cas à distinguer : les 2 listes sont vides, 1 seule liste est vide et les 2 sont non vides. En fonction des cas, on procédera de manière récursive comme expliqué ci-dessous dans les commentaires de la fonction.

```
    ;fonction recursive
(defun grouper (L1 L2)

    (cond
        ;si les 2 listes sont vides
        ((AND ( = (length L1) 0) (= (length L2) 0))
            ()
        )

        ;si la liste 1 est vide
        (( = (length L1) 0)
            ; on ajoute le premier element de L2 puis on appelle la
   fonction
            ; grouper pour le reste de L2
            (append (list(list (car L2))) (grouper NIL (cdr L2)))
        )

        ;si la liste 2 est vide
        (( = (length L2) 0)
            ; on ajoute le premier element de L1 puis on appelle la
   fonction
            ; grouper pour le reste de L1
            (append (list(list (car L1))) (grouper (cdr L1) NIL))
        )
```

```
        ;si les 2 listes sont non vides
        ((AND (car L1) (car L2))
            ; on ajoute a l'interieur de la liste de profondeur 0 une
    liste qui
            ; contient le premier element de L1 et le premier element
    de L2
            ; puis on rappelle la fonction grouper pour grouper le
    reste des listes
            ; L1 et L2
            (append (list(list (car L1) (car L2))) (grouper (cdr L1) (
    cdr L2)))
     )
     )
)
```

On peut effectuer les tests suivants, qui retournent les résultats escomptés :

```
(grouper '(1 2 3) '(4 5 6)) ; retourne ((1 4) (2 5) (3 6))
(grouper '(2 4 6 8) '(1)) ; retourne ((2 1) (4) (6) (8))
```

- monReverse(L) : retourne la liste inversée de L

Pour créer cette fonction, nous avons considéré que pour inverser une liste, il faut procéder de manière récursive. La condition d'arrêt est la taille de la liste est nulle : dans ce cas, on retourne une liste vide. Sinon, on renvoie une liste où l'on inverse le cdr de la liste et on ajoute le premier élément (car) à la fin de la liste.

```
(defun monReverse (L)
    ; si la liste est non vide
    (if ( > (length L) 0)
        ; on appelle de maniere recursive monReverse sur le cdr de L et
        ; on ajoute a la fin de la liste le premier element de L
        (append (monReverse (cdr L)) (list (car L)))
    )
)
```

Afin de vérifier que la fonction retourne le résultat escompté, on peut proposer le jeu de données suivant pour tester :

```
(monReverse '(1 2 3 4)) ; retourne (4 3 2 1)
(monReverse '(a b c d)) ; retourne (d c b a)
```

- palindrome(L) : retourne vrai si L est un palindrome

Pour ce faire, il faut regarder si le mot L et le mot retourné issu de L sont identiques. Si c'est le cas, on renvoie true. Cela peut être résumé comme suit :

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

Afin de vérifier que la fonction retourne le résultat escompté, on peut proposer le jeu de données suivant pour tester :

```
(palindrome '(k a y a k)) ; retourne true
(palindrome '(1 2 1)) ; retourne true
(palindrome '(b b c d r z)) ; retourne NIL
```

2 Exercice 2 : Objets fonctionnels

2.1 Ecrivez une fonction list-triple-couple qui retourne la liste des couples composés des éléments de la liste fournie en paramètre et de leur triple

Afin de créer la fonction, nous avons considéré que nous devions itérer sur chacun des éléments de la liste, et que pour un élément x, on devait retourner la liste (x 3*x). Ainsi, cela donne le code suivant :

```
(defun list-triple-couple (liste)
    ; pour chaque element x de la liste, on cree une liste qui contient
    ; x et le triple de x, qui sera donc de la forme (x 3x)
    (mapcar #'(lambda (elt) (list elt (* 3 elt))) liste)
)
```

Afin de vérifier que la fonction retourne le résultat escompté, on peut proposer le jeu de données suivant pour tester :

```
(list-triple-couple '(0 1 2 5 9)) ; renvoie ((0 0) (1 3) (2 6) (5 15)
   (9 27))
```

3 Exercice 3 : a-list

3.1 my-assoc (cle a-list) : retourne nil si cle ne correspond à aucune clé de la liste d'association, la paire correspondante dans le cas contraire

Pour concevoir cette fonction qui existe sous le nom de assoc, nous avons décidé de créer une fonction récursive. D'abord on regarde si la clé de la première paire est celle recherchée. Si c'est le cas on arrête le programme et on renvoie la paire correspondante. Le cas échant, on cherche de manière récursive dans le reste de la liste des paires. Lorsque la taille de la liste sera de 0, le programme s'arrêtera et l'on n'aura pas trouvé la clé dans la liste.

```
(defun my-assoc (cle a-list)
    ; on verifie que la longueur de la liste soit > 0
    (if (> (length a-list) 0)
        ; si la cle de la premiere paire de la liste est celle
    recherchee, on renvoie la paire
        (if (equal (caar a-list) cle)
            (car a-list)
            ; sinon, on recherche dans le cdr de la liste (les autres
    paires) placee en parametres
            (my-assoc cle (cdr a-list))
        )
        ; si la longueur de la liste est 0, on n'a pas trouve la cle
    voulue dans les paires
        NIL
    )
)
```

Afin de vérifier que la fonction retourne le résultat escompté, on peut proposer le jeu de données suivant pour tester :

```
(my-assoc 'x '((a 1) (c 21) (x 12) (z 2))) ; renvoie (x 12)
(my-assoc 'd '((a 1) (c 21) (x 12) (z 2))) ; renvoie NIL
```

3.2 cles (a-list) : retourne la liste des clés d'une A-liste

Pour créer cette fonction, on itère sur chacune des paires de la liste avec un mapcar et on ajoute la clé (le premier élément de la paire) à la liste que l'on retournera ensuite.

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

Afin de vérifier que la fonction retourne le résultat escompté, on peut proposer le jeu de données suivant pour tester :

```
(cles '((a 1) (c 21) (x 12) (z 2))) ; renvoie (a c x z)
```

3.3 creation (listeCles listeValeurs) : retourne une A-liste à partir d'une liste de clés et d'une liste de valeurs

Pour la fonction création, la solution la plus facile selon nous consiste à réutiliser la fonction grouper codée dans l'exercice 1. La seule condition à vérifier en plus est d'avoir des listes de même taille pour la création. Si les tailles des 2 listes sont égales, on applique grouper. Sinon, on affiche à l'écran que les tailles sont différentes.

```
(defun creation (listeCles listeValeurs)
    (if (AND (equal (length listeCles) (length listeValeurs)))
        (grouper listeCles listeValeurs)
        (PRINT "Les listes sont de tailles differentes")
    )
)
```

Afin de vérifier que la fonction retourne le résultat escompté, on peut proposer le jeu de données suivant pour tester :

```
(creation '(a b c) '(1 2 3)) ; renvoie ((a 1) (b 2) (c 3))
(creation '(a b c d) '(1 2 3)) ; affiche "Les listes sont de tailles
   differentes"
```

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

4.1 Compléter BaseTest avec les conflits commençant avant l'an 1100

A l'aide du lien Wikipédia proposé, on peut compléter la base de données comme suit avec les conflits ayant débuté avec 1100 (attention, afin que le texte soit formaté correctement, nous avons enlevé les accents uniquement pour le rapport) :

```
(setq BaseTest
'(
    ("Campagnes de Clovis 1er" 486 508 (("Royaume Franc") ("Domaine
    gallo-romain de Soissons" "Royaume alaman" "Royaume des Burgondes"
    "Royaume wisigoth")) ("Soissons" "Zulpich" "Dijon" "Vouille" "Arles
    "))
    ("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume des
    Burgondes")) ("Vezeronce" "Arles"))
    ("Conquete de la Thuringe" 531 531 (("Royaume Franc") ("Thuringes")
    ) ("Thuringe"))
    ("Guerre des Goths" 535 553 (("Royaume ostrogoth" "Alamans" "
    Royaume Franc" "Royaume wisigoth" "Burgondes") ("Empire byzantin"))
    ("Peninsule italienne"))
    ("Conquete de l'Alemanie" 536 536 (("Royaume Franc") ("Alamans")) (
    "Alemanie"))
    ("Conquete de la Baviere" 555 555 (("Royaume Franc") ("Bavarii")) (
    "Baviere"))
    ("Campagnes de Breatagne" 560 578 (("Royaume Franc") ("Royaume de
    Vannetais")) ("Vannetais"))
    ("Guerre de succession merovingienne" 584 585 (("Royaume Franc") ("
    Royaume d'Aquitaine")) ("Comminges"))
    ("Guerre franco-frisonne" 600 793 (("Royaume Franc") ("Royaume de
    Frise")) ("Pays-Bas" "Allemagne"))
    ("Guerre civile des Francs" 715 719 (("Neustrie") ("Austrasie")) ("
    Royaume Franc"))
    ("Invasion omeyyade en France" 719 759 (("Royaume Franc") ("Califat
    omeyyade")) ("Royaume d'Aquitaine" "Septimanie"))
    ("Guerre des Lombards" 755 758 (("Royaume Franc") ("Lombards")) ("
    Lombardie"))
    ("Guerre d'Aquitaine" 761 768 (("Royaume Franc") ("Aquitains")) ("
    Vasconie" "Aquitaine"))
    ("Guerre des Saxons" 772 804 (("Royaume Franc") ("Saxons")) ("
    Germanie"))
    ("Guerre des Lombards" 773 774 (("Royaume Franc") ("Lombards")) ("
    Lombardie"))
    ("Guerre des Avars" 791 805 (("Royaume de France") ("Avars")) ("
    Pannonie"))
    ("Invasions sarrasines en Provence" 798 990 (("Royaume de France" "
    Comte de Provence") ("Sarrasins")) ("Provence"))
    ("Guerre civile entre les fils de Louis le Pieux" 830 842 (("
    Francie occidentale" "Francie orientale")) ("Fontenoy"))
    ("Guerre franco-bretonne" 843 851 (("Royaume de France") ("Royaume
    de Bretagne" "Vikings")) ("Royaume de Bretagne"))
    ("Luttes inter-dynastiques carolingiennes" 876 946 (("Francie
    occidentale" "Francie orientale") ("Royaume de Bourgogne" "Francie
```

```
    orientale")) ("Ardennes" "Saone-et-Loire" "Rhenanie-Palatinat" "
    Aisne"))
     ("Invasions vikings en France" 799 1014 (("Royaume de France") ("
    Vikings")) ("Normandie" "Bretagne"))
     ("Premiere croisade" 1096 1099 (("Comte de Blois" "Comte de
    Toulouse" "Comte de Boulogne" "Marquisat de Provence" "Comte de
    Flandre" "Duche de Normandie" "Diocese du Puy-en-Velay" "Comte de
    Vermandois" "Republique de Genes" "Duche de Basse-Lotharingie" "
    Principaute de Tarente" "Empire byzantin" "Royaume de
    Petite-Armenie" "Croises" "Royaume de France") ("Sultanat de Roum"
    "Danichmendides" "Califat fatimide")) ("Terre sainte"))
 )
)
```

4.2 Définir les fonctions de service

En utilisant les propriétés du car et du cdr vues en cours, on en déduit facilement les fonctions de l'exercice.

- dateDebut (conflit) : retourne la date de début du conflit passé en argument

La date de début est le deuxième élément de la liste, d'où le code suivant :

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

On peut proposer le jeu de tests suivants pour tester :

```
(dateDebut '("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume
    des Burgondes")) ("Vezeronce" "Arles"))) ; renvoie 523
```

- nomConflit (conflit) : retourne le nom du conflit passé en argument

Le nom du conflit est le premier élément de la liste, d'où le code suivant :

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

On peut proposer le jeu de tests suivants pour tester :

```
(nomConflit '("Guerre de Burgondie" 523 533 (("Royaume Franc") ("
    Royaume des Burgondes")) ("Vezeronce" "Arles"))) ; renvoie "Guerre
    de Burgondie"
```

- allies (conflit) : retourne les alliés du conflit passé en argument

Les alliés forment le premier élément de la liste contenant les "acteurs" du conflit. Ces acteurs se trouvent en 4ème position dans la liste du conflit, d'où le code suivant :

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

On peut proposer le jeu de tests suivants pour tester :

```
(allies '("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume
    des Burgondes")) ("Vezeronce" "Arles"))) ; renvoie ("Royaume Franc
    ")
```

- ennemis (conflit) : retourne les ennemis du conflit passé en argument

Les ennemis forment le deuxième élément de la liste contenant les "acteurs" du conflit. Ces acteurs se trouvent en 4ème position dans la liste du conflit, d'où le code suivant :

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

On peut proposer le jeu de tests suivants pour tester :

```
(ennemis '("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume
    des Burgondes")) ("Vezeronce" "Arles"))) ; renvoie ("Royaume des
    Burgondes")
)
```

- lieu (conflit) : retourne le lieu du conflit passé en argument

Les lieux du conflit sont contenus dans une liste, qui est le 5 élément de la liste modélisant le conflit, d'où le code suivant :

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

On peut proposer le jeu de tests suivants pour tester :

```
(lieu '("Guerre de Burgondie" 523 533 (("Royaume Franc") ("Royaume des
    Burgondes")) ("Vezeronce" "Arles"))) ; renvoie ("Vezeronce" "Arles
    ")
```

4.3 C. En utilisant ces fonctions de service, définir les fonctions suivantes

Dans l'exercice, on considèrera que baseTest n'est pas une variable globale spéciale, et qu'il est donc nécessaire de la placer en paramètre des fonctions. Cela permettra d'avoir un code plus "portable" puisque l'utilisateur pourra donner le nom qu'il souhaite à sa variable baseTest sans que cela génère des erreurs.

- FB1 : affiche tous les conflits

Pour afficher tous les conflits d'une base de données placée en paramètre de la fonction, on peut procéder comme suit, en affichant chaque conflit de la base. Ainsi, un mapcar nous permet d'itérer sur les conflits pour les afficher.

```
(defun FB1(baseTest)
    (mapcar #'(lambda (conflit) (print conflit))baseTest)
)
```

On peut proposer le test suivant pour vérifier :

```
(FB1 BaseTest) ; affiche tous les conflits
(FB1 NIL) ; n'affiche rien et renvoie NIL car liste vide
```

- FB2 : affiche les conflits du "Royaume Franc"

Pour afficher les conflits du "Royaume Franc", on itère sur chacun des conflits de la base de données placée en paramètres avec un mapcar, et on itère sur les alliées pour voir si l'allié est bien celui recherché ou non. Dans le cas où c'est le "Royaume Franc", on ajoute le conflit à la liste que l'on retournera ensuite. En faisant un mapcar, nous avons rencontré un problème. Le "if", s'il est évalué à NIL, renvoie NIL, et est ajouté à la liste de retour. Ce qui ne donne pas le résultat souhaité puisque des NIL viennent s'insérer. Pour pallier ce problème, nous avons choisi de faire un mapcan, qui renvoie une concatenated-list correspondant aux attentes du problème.

```
(defun FB2(baseTest)
    ; Iteration sur les conflits de la base
    (mapcan #'(lambda (conflit)
        ; Iteration sur les allies du conflit
        (mapcan #'(lambda (allies)
            ; Si le Royaume Franc est un des allies, on l'ajoute a la
   liste a retourner
            (if (equal allies "Royaume Franc")
                (append (list conflit)))
        ) (allies conflit)
        )
    )baseTest
    )
)
```

Le test suivant nous permet de vérifier le fonctionnement de notre fonction :

```
(FB2 BaseTest) ; affiche tous les conflits du Royaume Franc
```

- FB3 : retourne la liste des conflits dont un allié est précisé en argument

La fonction étant similaire à FB2, on peut procéder de manière analogue en rajoutant un paramètre "allie" à la fonction. Et dans le test d'égalité, il nous reste plus qu'à remplacer le "Royaume Franc" par l'allié placé en paramètre.

```
(defun FB3(baseTest allie)
    (mapcan #'(lambda (conflit)
        (mapcan #'(lambda (allies)
            (if (equal allies allie)
                (append (list conflit)))
        ) (allies conflit)
        )
    )baseTest
    )
)
```

On peut proposer le test suivant pour vérifier le fonctionnement de la fonction :

```
(FB3 BaseTest 'Neustrie) ; renvoie ("Guerre civile des Francs" 715 719
   (("Neustrie") ("Austrasie")) ("Royaume Franc"))
```

- FB4 : retourne le conflit dont la date de début est 523

Pour cela, nous avons choisi de parcourir l'ensemble des conflits de la base de données avec un mapcar, et on regarde si la date de début est bien 523 ou non. Si c'est le cas, on retourne alors le conflit.

```
(defun FB4(baseTest)
    ; Iteration sur les conflits
    (mapcan #'(lambda (conflit)
        ; Si le conflit a debute en 523, on retourne le conflit
        (if (equal (dateDebut conflit) 523)
            conflit
        )
    )baseTest
    )
)
```

On peut proposer le test suivant pour vérifier :

```
(FB4 BaseTest) ; renvoie ("Guerre de Burgondie" 523 533 (("Royaume
   Franc") ("Royaume des Burgondes")) ("Vezeronce" "Arles"))
```

Cette méthode n'est cependant pas optimale dans le cas où les conflits sont rangés par ordre croissant de date de début (comme réalisé ci-dessus). Toutefois, si appliqué sur une base test où les conflits ne sont pas ordonnés par date de début, le code renvoie le résultat escompté.

- FB5 : retourne la liste des conflits dont la date de début est comprise entre 523 et 715

On procède de manière analogue. Néanmoins, au lieu de tester si la date de début est 523, on regarde si la date est comprise entre 523 et 715 (n'ayant pas d'indication plus précise, nous considérerons que les années 523 et 715 sont exclues de la recherche) à l'aide des opérateurs “>” et “<”.

```
(defun FB5(baseTest)
    (mapcan #'(lambda (conflit)
        (if (AND(>= (dateDebut conflit) 523)(<= (dateDebut conflit)
    715))
            conflit
        )
    )baseTest
    )
)
```

On peut proposer le test suivant pour vérifier :

```
(FB5 BaseTest) ; renvoie tous les conflits ayant debute entre 523 et
    715
```

- FB6 : calcule et retourne le nombre de conflits ayant pour ennemis les "Lombards"

Pour réaliser cette fonction, on initialise d'abord une variable locale “taille” qui contient le nombre de conflits ayant pour ennemis les “Lombards”. Puis on itère sur chacun des conflits comme précédemment, puis sur chacun des ennemis, et on teste si c'est les “Lombards” ou non. Dans le cas où les Lombards sont des ennemis du conflit, on incrémente de 1 la variable taille, que l'on retournera ensuite.

```
(defun FB6(baseTest)
    ; Creation de la variable taille
    (let ((taille 0))
    ; Parcours des conflits de la base
    (mapcar #'(lambda (conflit)
        ; Parcours des ennemis
        (mapcar #'(lambda (ennemi)
            ; Si les lombards sont ennemis
            (if (equal ennemi "Lombards")
                ; on rajoute 1 a la variable taille
                (setq taille (+ taille 1))
        ))(ennemis conflit))
    )baseTest
    )
    taille
    )
)
```

On peut vérifier le résultat avec le test suivant :

```
(FB6 BaseTest) ; retourne 2
```

Au cours de cet exercice, nous avons fait la supposition que l'utilisateur rentre des données correctes en paramètre pour baseTest, c'est-à-dire, une liste. Dans le cas où il rentrerait un autre type de données en paramètre, cela génèrera une erreur. Afin de gérer cette exception, il aurait été possible de rajouter dans chacune des fonctions un test "if" pour savoir si le type du paramètre était celui escompté. Dans le cas échant, nous aurions pu faire un affichage indiquant l'erreur. Nous avons toutefois choisi de ne pas implémenter cette gestion d'erreur. En effet, elle n'était pas imposée

par l'énoncé et n'apportait pas de concepts nouveaux pour monter en compétences Lisp.

Conclusion

Au travers de ces quatre exercices, ce TP nous a permis de nous familiariser avec le langage Lisp, précurseur dans le domaine de l'intelligence artificielle. L'étude et la manipulation de la structure des listes, des fonctions, de la récursivité et des symboles nous a permis de comprendre les bases du langage.

Ce mini-projet nous a permis de lever les doutes sur certains points du cours en appliquant de manière concrète toutes les fonctions de base de Lisp. Il nous a également permis de développer notre capacité d'adaptation dans certains cas, notamment pour le dilemme entre récursivité et parcours itératif dans le dernier exercice.