IA01 : Rapport du TP1

Auteurs :
Camille Gerin-Roze
Thomas Perrin

Le 2 Octobre 2015

Table des matières

Introduction

Ce TP a pour but de nous initier à la programmation avec Lisp. Ce langage qui est un langage de programmation fonctionnelle nous présente un nouveau paradigme totalement différent de la programmation impérative que nous avons utilisé auparavant.
Dans ce TP nous nous sommes réparti les tâches de manière équitable de manière à pouvoir tous les deux apprendre à nous servir du langage Lisp, nous avons également pu nous entraider et partager nos travaux grâce à un dépôt privé GIT sur le site Github.

Chapitre 1

Exercice 1

1.1 Ecriture préfixée

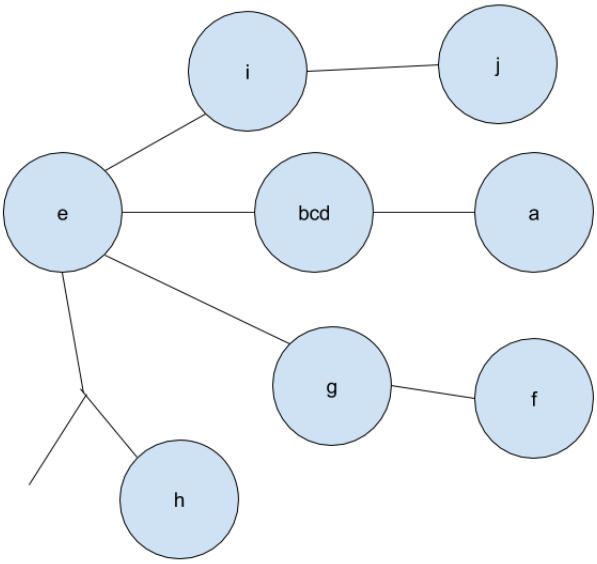
$$4x^3–5x^2 + 3x + 1$$

devient

(+ (* 4 x x x) (* -5 x x) (* 3 x) 1)

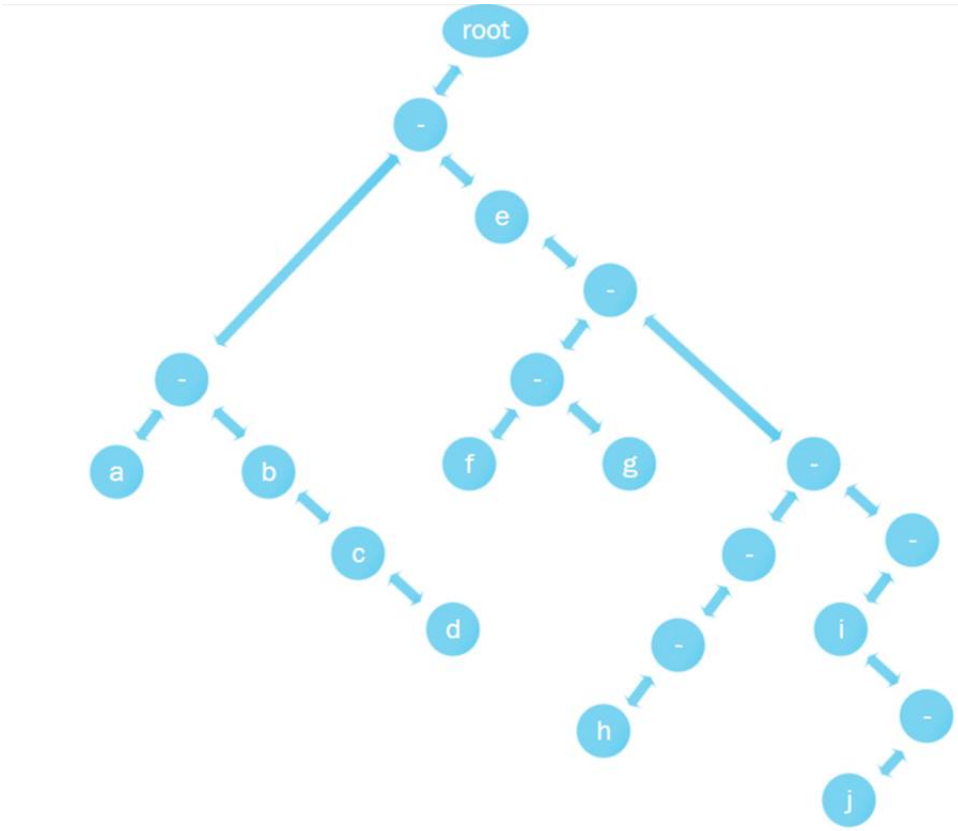
1.2 Représentation sous forme d'arbre

Arbre généalogique



Voici la représentation sous forme d'arbre généalogique. Tous les éléments qui sont au même niveau dans la liste, sont au même niveau dans l'arbre.

Arbre binaire



L'arbre binaire est représenté de la façon suivante : le lien gauche est un fils, le lien de droite est un frère. Concernant le caractère " - ", il représente le début d'une sous-liste, tandis que "root" représente la racine, c'est-à-dire le début de la liste principale.

1.3 Donner les premiers N éléments

1.3.1 Version récursive

```
(defun firstn (n l)
  (if (and (> n 0) (car l))
  (append (list (car l)) (firstn (- n 1) (cdr l)))))
```

La fonction prend deux arguments :
— Le nombre de **n** d'éléments à prélever dans la liste
— La liste **l** dans laquelle nous allons prélever les éléments

On prend le premier argument de la liste l, et on construit une nouvelle liste avec un appel récursif de la fonction avec **n-1** éléments, jusqu'à ce que n soit égale à 0 ou que la liste soit finie (On aurait pu également déclencher une erreur si la longueur de la liste est inférieur à n).

1.3.2 Version itérative

```
(defun firstn-ite (n l)
  (let ((listeEntiere l))
    (setq liste ())
  (dotimes (x n liste)
    (if (not (null listeEntiere) ) (progn
    (setq liste (append liste (list (car listeEntiere))))
    (setq listeEntiere (cdr listeEntiere))
    )))))
```

La fonction itérative se déroule différement. On crée une nouvelle liste dans laquelle on va copier les n premiers éléments. On doit donc utiliser deux listes différentes. La fonction dans son format itératif utilise donc plus de mémoire.

1.3.3 Jeu de tests

```
(firstn 3 '( 1 5))
(1 5)
(firstn 3 '( 1 5 6 4 8))
(1 5 6)
(firstn -3 '( 1 5 6 4 8))
NIL
(firstn-ite -3 '( 1 5 6 4 8))
NIL
(firstn-ite 3 '( 1 5 6 4 8))
(1 5 6)
(firstn-ite 3 '( 1 5))
(1 5)
```

1.4 Intersection de 2 listes

La méthode member ne prend pas les sous-liste en compte car elle utilise EQL et non EQUAL. On définit donc un méthode **mymember** qui utilisera equal à la place de eql :

```
(defun myMember (x l)
  (if (not (null l))
      (if (equal x (car l)) l (myMember x (cdr l)))))
```

1.4.1 Version récursive

```
(defun inter (l1 l2)
  (if (not (null (car l1)))
      (if (mymember (car l1) l2)
         (append (list (car l1)) (inter (cdr l1) l2)) ;;; Si
        (inter (cdr l1) l2)) ) ;;; Sinon
```

La fonction prend en paramètre deux listes (l1 et l2). Les appels récursif continue jusqu'à ce que le car de la première liste (celle sur laquelle nous allons itérer) Si le car de l1 est un membre de la liste l2, alors on va concaténer la liste contenant le car de l1 et le résultat de l'appel récursif sur le cdr de l1 et l2. Sinon on fera juste l'appel récursif.

1.4.2 Version itérative

```
(defun inter-iteratif (l1 l2)
    (mapcan #'(lambda (x) (if (mymember x l2) (list x))) l1))
```

La version itérative est plus simple car en utilisant mapcan (qui renvoie une nouvelle liste) on peut tout simplement itérer sur la première liste et vérifier que chaque élément et dans la deuxième. Si l'élément est dans la liste, alors on retourne l'élément, sinon on retourne nil. Grâce au mapcan qui utilise **cons** pour concaténer les résultats, les nil ne seront pas membre de la liste retourner par la fonction.

1.4.3 Jeu de tests

```
(inter '(1 2 3) '())
NIL
(inter '(1 2 3) '(2 3))
(2 3)
(inter '(1 2 3 (4 5 6)) '(1 2 3 (4 5 6)))
(1 2 3 (4 5 6))
(inter-iteratif '(1 2 3) '())
NIL
(inter-iteratif '(1 2 3) '(2 3))
(2 3)
(inter-iteratif '(1 2 3 (4 5 6)) '(1 2 3 (4 5 6)))
(1 2 3 (4 5 6))
```

1.5 Elimination des doublons d'une liste

On utilisera encore une fois la fonction mymember aulieu de member pour pouvoir gérer les sous-listes.

1.5.1 Version récursive

```
(defun elim (l1)
(if (null (car l1)) ()
  (if (mymember (car l1) (cdr l1))
      (elim (cdr l1))
    (append (list (car l1)) (elim (cdr l1))))))
```

La fonction prend en argument une liste l1, il faut alors vérifier que cette liste ne soit pas vide. Si cette liste n'est pas vide alors on vérifie que le premier élément n'est pas présent dans le reste de la liste, si il est présent on va rappeller la fonction sans ajouter cet élément de manière à le supprimer. Dans le cas ou l'élément est absent du reste de la liste, on garde l'élément et on lui ajoute le résultat de l'appel récursif sur la suite de la liste (cdr l1).

1.5.2 Version itérative

```
(defun elim-ite (l1)
  (let ((listResult ()))
    (dolist (x l1 listResult)
      (if (and (mymember x l1) (not (mymember x listResult)))
          (setq listResult (cons x listResult))))))
```

pour la version itérative de la fonction, on va simplement définir une nouvelle liste contenant le résultat. Puis on va itérer sur tous les éléments de notre list passée en paramètre en ajoutant à notre listResult uniquement les éléments ne s'y trouvant pas déjà.
Cet algorithme est plus simple à réaliser mais il demande plus de mémoire car on va devoir stocker le résultat dans une autre liste.

1.5.3 Jeu de tests

```
(elim '(1 2 3 4 5 6 6 5 4 3 2))
(1 6 5 4 3 2)
(elim '(1 2 3 4))
(1 2 3 4)
(elim '(1 2 3 4 (1 2) (1 2)))
(1 2 3 4 (1 2))
(elim-ite '(1 2 3 4 (1 2) (1 2)))
((1 2) 4 3 2 1)
(elim-ite '(1 2 3 4 5 6 6 5 4 3 2))
(6 5 4 3 2 1)
(elim-ite '(1 2 3 4))
(4 3 2 1)
```

1.6 Compteur de feuille

```
(defun nbfeuilles (l1) (if (null (car l1))
          0
        (if (listp (car l1))
           (+ (nbfeuilles (car l1)) (nbfeuilles (cdr l1)))
           (+ 1 (nbfeuilles (cdr l1))))))
```

Ici nous avons développé seulement une version récursive de l'algorithme car la représentation d'un arbre sous forme de liste entraine forcément la formation de sous-arbres et donc l'utilisation de la même fonction pour les traiter, un algorithme itératif ici ne serait donc pas naturel. Si la liste est nulle, la fonction retourne 0 et ne fait pas d'appel récursif, si elle contient une liste, on additionera les résultats des appels récursifs sur le car et le cdr. Si l'élément n'est pas une liste, alors nous avons atteint une feuilles donc on retourne le résultat de la fonction appliquée au cdr plus un pour la feuille que nous avons atteint.

1.6.1 Jeu de tests

```
(nbFeuilles '( r (( t )) y ( g h ) ( j m l ) p ))
9
(nbFeuilles '( a b c ))
3
'( a b c (d e) nil ))
5
```

1.7 Implémentation de la méthode MONEQUAL

```
(defun monEqual (e1 e2)
  (cond
    (( and (listp e1) (listp e2) (not (null e1)) (not (null e2)))
       (and (monEqual (car e1) (car e2)) (monEqual (cdr e1) (cdr e2))))
    ((and (atom e1) (atom e2))
                  (eq e1 e2))
    ((and (null e1) (null e2))
                  T)
    ('T 'nil)))
```

La différence fondamentale entre EQ et EQUAL est le fait que EQUAL va analyser le contenu des listes qu'on peut lui passer en paramètre. Nous avons donc des tests différents à faire en fonction de la nature des différents arguments. Nous avons déterminer 4 cas :

- Si les deux éléments sont des listes non nulles, alors la fonction doit utiliser l'opérateur **AND** entre la comparaison des car des deux listes, et le résultat de l'appel récursif sur le cdr.
- Si les deux éléments sont des atomes, alors on vérifie juste qu'ils sont égaux avec EQ.
- Si les deux éléments sont nil, alors ils sont égaux.
- Si il ne rentrent pas dans les cas précédent, cela veut dire qu'ils ne sont pas de la même nature (nil, atome ou liste) et donc ils sont différents.

1.7.1 Jeu de tests

```
(monequal 'LUC 'LUC)
T
(monequal 'LUC 'DANIEL)
NIL
(monequal '(d p f t r) '(d p f t r))
T
(monequal (car '(do re)) (cadr '(mi do sol)))
T
```

Chapitre 2

Exercice 2

La finalité de cet exercice est de créer une fonction list-paire retourner une liste des éléments par paire de deux listes précisées en paramètre.

2.1 Implémentation

```
-- Premiere solution : Avec le mapcar
-- Les deux arguments sont les deux listes.
(defun list-paire (x y)
                (if (and (listp x) (listp y))
                  (mapcar #'(lambda (x y) (list x y)) x y)
                  "Les deux arguments doivent etre des listes !"
                  )
)
```

L'utilisation de la fonction mapcar est le but de cette exercice c'est ainsi pourquoi cette première solution l'utilise. On utilise une fonction lambda qui sera appliquée sur chacun des éléments des listes afin de créer une sous liste à chaque itération.

```
-- Seconde solution : recursivite
-- Les deux arguments sont les deux listes.
(defun list-paire-r cursif (x y)
  (if (not (or (null (cdr x)) (null (cdr y))))
        (cons (list (car x) (car y)) (list-paire-r cursif (cdr x)
(cdr y)))
        (list (list (car x) (car y)))
  )
)
```

La seconde version de cette fonction met en avant l'avantage du mapcar en comparaison de la récursivité en terme de syntaxe. La condition d'arrêt de cette fonction est lorsque la paire actuelle est la dernière en vérifiant la suivante. Le principe est la concaténation des différentes listes.

2.2 Jeu de tests

2.2.1 Tests basiques

```
(list-paire '(0 2 3 11) '(6 10 20 30))
(list-paire-r cursif '(0 2 3 11) '(6 10 20 30))
```

2.2.2 Gestion de listes de différentes tailles

```
(list-paire '(0 2 3 11 15) '(6 10 20 30))
(list-paire-r cursif '(0 2 3 11 15) '(6 10 20 30))
```

Chapitre 3

Exercice 3

Le but de l'exercice est de développer une fonction permettant de récupérer la valeur d'une clé dans une liste de couple (clé valeur).

3.1 Implémentation

```
(defun my-assoc (list elem)
  (setq value nil)
  (dolist (x list value)
    (if ( equal (car x) elem) (setq value (cdr x)))))
```

Nous allons donc parcourir la liste grâce à la méthode **dolist** et on remplira la valeur de retour **value** si la clé du couple est celle passée en paramètre de la fonction.

3.2 Jeu de tests

```
(setq base '((a 1) (b 2) (c 3) (abc (1 2 3)) (abab ((1 2) (1 2)))))
(my-assoc base 'a)
1
(my-assoc base 'b)
2
(my-assoc base 'c)
3
(my-assoc base 'abab)
((1 2) (1 2))
(my-assoc base 'abc)
(1 2 3)
(my-assoc base 'notHere)
NIL
```

Pendant les tests, on voit que la valeur retournée est la bonne même si celle-ci est une liste. Dans le cas où la clé n'existe pas, la valeur NIL sera retournée.

Chapitre 4

Exercice 4

Dans ce dernier exercice, on utilisera la base de données suivante définit dans l'énoncé :

```
(setq BaseTest '((" Le Dernier Jour d'un condamne " Hugo 1829 50000)
(" Notre-Dame de Paris " Hugo 1831 3000000)
(" Les Miserables " Hugo 1862 2000000)
("Le Horla " Maupassant 1887 2000000)
(" Contes de la becasse " Maupassant 1883 500000)
("Germinal " Zola 1885 3000000)
))
```

4.1 Fonctions de service

```
(defun auteur (ouvrage)
  (cadr ouvrage))

(defun titre (ouvrage)
  (car ouvrage))

(defun annee (ouvrage)
  (caddr ouvrage))

(defun nombre (ouvrage)
  (cadddr ouvrage))
```

Ces différentes fonctions permettent de récupérer une information précise d'un ouvrage. Elles vont être réutilisées dans les fonctions suivantes.

4.2 Afficher tous les ouvrages

```
(defun affichAllOuvrage (base)
  (mapcar #'(lambda (l) (titre l)) base))
```

L'affichage des ouvrages se fait par une simple itération via un mapcar, en utilisant une fonction de base.

4.3 Afficher tous les ouvrages de Hugo

```
(defun affichAllOuvrageOfHugo (base)
  (remove nil
  (mapcar #'(lambda (l)
  (if (EQ (auteur l) 'Hugo) (titre l) ())) base)))
```

L'affichage des ouvrages spécifique à Victor Hugo se filtre grâce à la fonction if et à la fonction auteur réalisée précédemment. On retire les NIL via la fonction remove.

4.4 Afficher tous les ouvrages d'un auteur

```
(defun affichAllOuvrageOfAutor (base autor)
  (remove nil
  (mapcar #'(lambda (l)
  (if (EQ (auteur l) autor) (titre l) ())) base)))
```

Le filtrage par un auteur en paramètre se réalise de la même manière en utilisant un argument supplémentaire au lieu d'utiliser un mot brut.

4.5 Afficher le premier ouvrage d'une année

```
(defun affichFirstOuvrageOf (base year)
  (dolist (l base) (if (EQ (annee l) year) (return (titre l)))))
```

Dans cette fonction, nous utilisons un dolist afin de pouvoir s'arrêter dès que la condition est satisfaite, à l'aide encore une fois d'une fonction de service.

4.6 Afficher les ouvrages vendue à plus de 100k exemplaires

```
(defun affichOuvrageWithMore100k (base)
  (remove nil
  (mapcar #'(lambda (l)
  (if (>= (nombre l) 100000) (titre l) ())) base)))
```

Retour à l'utilisation du mapcar puisqu'on itère sur toute la liste. Nous vérifions le nombre d'exemplaires grâce à une fonction de service.

4.7 Afficher la moyenne du nombre de ventes de l'ouvrage d'un auteur

```
defun moyenne (base aut)
  (let ((nbExemplaire 0.0)(sum 0.0))
   (/ (dolist (livre base sum)
   (if (equal aut (auteur livre))
     (progn
     (setq nbExemplaire (+ nbExemplaire 1))
     (setq sum (+ sum (cadddr livre)))))
   ) nbExemplaire )))
```

Enfin, la fonction de la moyenne est réalisée à l'aide d'un dolist et de variable. Le let au départ permet de définir les variables sum et nbExemplaire afin de retenir le nombre d'exemplaires et d'ouvrages pour finalement réaliser à la fin la division. La variables sum retournée à la fin sera divisée par le nbExemplaire.

Jeu de tests

```
(affichAllOuvrage BaseTest)
(affichAllOuvrageOfHugo BaseTest)
(affichAllOuvrageOfAutor BaseTest 'Maupassant)
(affichFirstOuvrageOf BaseTest 1831)
(affichOuvrageWithMore100k BaseTest)
(moyenne BaseTest 'Maupassant)
(moyenne BaseTest 'Hugo)
```

Conclusion

Ce TP nous a permis de mieux comprendre le fonctionnement du langage Lisp. Il nous a également permis de mieux comprendre le paradigme de programmation fonctionnelle qui met l'accent sur la récursivité. Au final nous avons également pu faire une première gestion d'une base de connaissance avec Lisp, ce que nous devrons sûrement refaire dans les prochains TPs.