TP1 – INTRODUCTION AU LISP

Ce rapport a pour objectif de détailler le code rendu dans le fichier .lisp livré en annexe et de répondre aux questions préliminaires posées dans le sujet. Nous essayerons, dans la mesure du possible, d'expliciter notre raisonnement mais aussi nos éventuelles difficultés, notamment pour les fonctions plus complexes.

♦ Exercice 1

Questions préliminaires

- 4 x^3 - 5x^2 + 3x + 1 s'écrit en préfixé (+ (- (* 4 (^ x 3)) (* 5 (^ x 2))) (+ (* 3 x) 1))

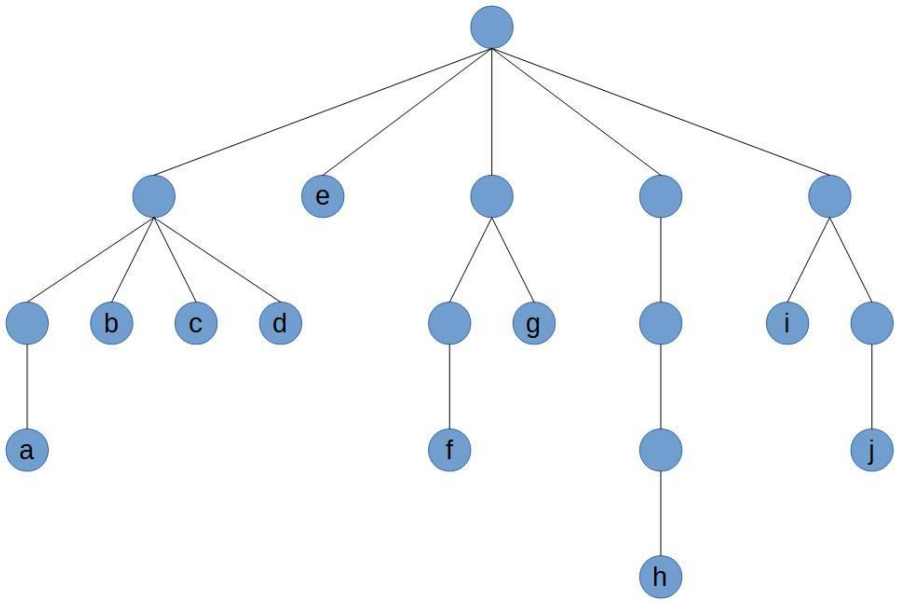

Figure 1 Représentation en arborescence de type arbre généalogique de (((a) b c d) e ((f) g) (((h))) (i (j)))

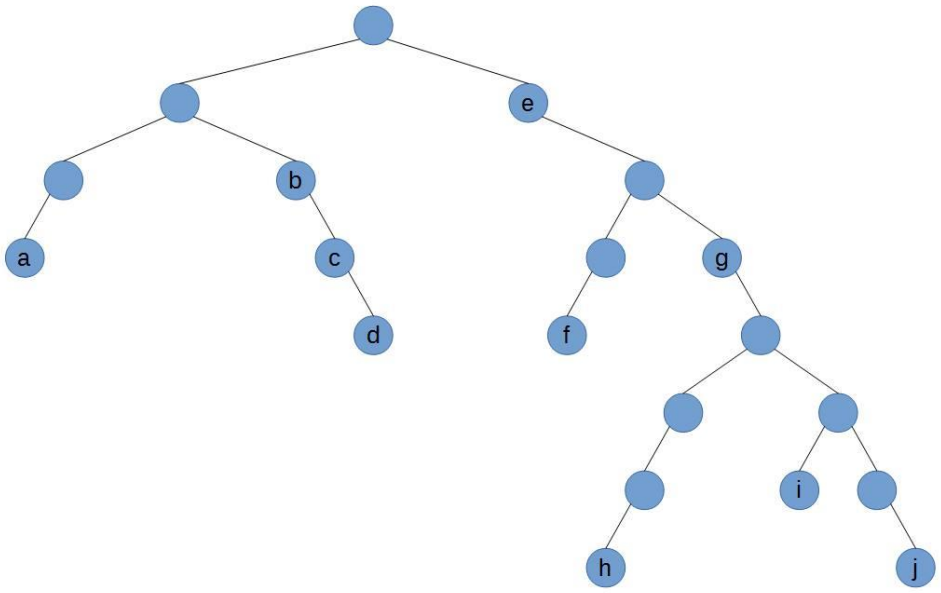

Figure 2 Représentation en arborescence de type arbre binaire de (((a) b c d) e ((f) g) (((h))) (i (j)))

Pour la fonction *firstn*, on construit la liste récursivement en concaténant le n-ième élément avec la liste des n-1 éléments suivants. On veille à construire une condition d'arrêt double : soit la liste est épuisée, soit n atteint 0.

La fonction *suppr* se contente de s'arrêter sur un élément passé en argument d'une liste donnée puis de le supprimer de la liste de retour.

La fonction *supprAll* repose sur le même principe que la précédente à ceci près qu'elle teste chaque élément puisqu'il faut éliminer toutes les occurrences.

La fonction *inter* renvoie une liste contenant tous les éléments communs à deux listes passées en arguments. Pour cela, elle teste l'égalité entre la tête de liste de L et chaque élément de M. Si le test est vrai, alors l'élément est ajouté à la liste résultat et supprimé de M (gain de temps car on ne testera plus cet élément pour la suite de L), sinon, on effectue un appel récursif sur L privée de sa tête. La fonction s'arrête lorsque l'une des deux listes est épuisée.

La fonction *elim* retourne une liste composée d'un premier élément et d'une suite d'où l'on a retiré toutes les occurrences grâce à une réutilisation de *supprAll*.

La fonction *nbFeuilles* calcule de façon simple : soit la liste est vide, auquel cas elle renvoit 0 (c'est un non-arbre sans feuille), soit l'élément testé n'est pas une liste et dans ce cas c'est une feuille, comptabilisée 1, soit c'est une liste et le nombre de feuilles est calculé récursivement en faisant la somme des feuilles de ses fils.

La fonction *monEqual* test les arguments qu'on lui donne : si ce sont des atomes, il suffit d'utiliser le *eq* standard (qui permet de tester l'égalité de deux atomes, mais renvoie NIL si l'une ou l'autre des éléments comparés sont des listes), sinon, pour tester l'égalité dans le cas de deux listes, on utilise une fonction *eqListe* que nous avons implémenté, détaillée ci-après.
La fonction *eqListe* teste l'égalité entre deux listes passées en argument en testant les égalités successives de leurs éléments suivant l'ordre ci-après :

→ si les deux listes sont vides, elles sont considérées comme égales. C'est aussi la condition d'arrêt.

→ si les deux têtes de liste sont des listes, on appelle récursivement *eqListe* pour déterminer leur égalités puis on rappelle *eqListe* pour tester la suite des listes

→ si au moins l'une des deux têtes de listes est un atome, on peut utiliser le *eq* standard ; on rappelle ensuite *eqListe* sur la fin des listes.

→ si aucune des conditions n'a été vérifiée, les listes ne sont pas égales et la fonction s'arrête.

♦ Exercice 2 – fonction list-paire

Ici, *mapcar* va chercher le premier élément de chaque liste A et B et donne ce couple d'élément comme arguments à la

fonction *lambda* qui se charge de les assembler en une liste. Ensuite *mapcar* continue avec les listes privées de leurs têtes. Enfin, la fonction s'arrête lorsqu'au moins une liste est épuisée (test *if*).

♦ Exercice 3 – fonction my-assoc

On commence par initialiser la variable retour à NIL. Ainsi, si la clé n'est pas trouvée, la fonction renverra une liste vide. Ensuite, on utilise le *dolist* pour parcourir la liste d'associations (appelée a-liste) association par association. Si la clé se trouve dans l'association, on remplace la liste vide par la liste (clé valeur) de l'association trouvée ; sinon, on ne fait rien et on passe à l'association suivante.

♦ Exercice 4 – gestion d'une liste d'ouvrages

On commence par définir les fonctions "de service" qui se contentent de renvoyer le i-ème élément d'une liste donnée comme ouvrage sous le format ("titre" auteur année nombre_d'ouvrages_vendus).

Pour afficher tous les ouvrages, on extrait une à une toutes les informations de chaque ouvrage (*fonctions de service*) puis on les recombine (*list*) en une liste que l'on affiche (*print*). Cette méthode permet, si on le souhaite, de changer le format d'affichage de la base de données.

Pour FB2, on parcourt ouvrage par ouvrage la liste de la base de données initiale grâce à un *dolist* puis on teste si l'auteur est HUGO. Si oui, alors on affiche son titre, sinon, on passe à l'ouvrage suivant. La fonction s'arrête lorsque la base de données est vide (pas d'entrée dans le *dolist*).

FB3 reprend le principe de la fonction précédente mais ajoute deux contraintes : l'auteur doit être décidé par l'utilisateur (il faut donc ajouter une variable *nom_auteur*) et la fonction doit retourner une liste. On teste ensuite l'égalité entre le nom de l'auteur de l'ouvrage et celui entré en argument par l'utilisateur. Si le test est vrai, on ajoute son titre à une liste de retour, sinon on relance récursivement le fonction FB3 sur l'ouvrage suivant. L'absence de *dolist* contraint à créer une condition d'arrêt (liste épuisée).
Nous avons également crée FB3bis qui renvoie la liste des caractéristiques de tous les ouvrages de l'auteur passé en argument (pas seulement les titres). Nous avions prévu de l'utiliser pour FB6 mais nous avons finalement une autre méthode. Nous laissons volontairement la fonction pour un éventuel usage futur.

FB4 reprend la structure de FB3 à ceci près qu'elle s'arrête au premier ouvrage trouvé qui correspond au critère (pas d'appel récursif).

FB5 reprend la structure de FB3 mais avec un changement dans la condition du test (nombre d'exemplaires vendus < 1.000.000[1]).

[1]Signalons notre étonnement quant au nombre d'exemplaires du *Dernier jour d'un condamné* d'Hugo …

FB6 est la fonction qui nous aura demandé le plus de réflexion, quand bien même sa structure générale est simple à comprendre. On commence par initialiser les variables nécessaires (*nombre_ouvrage* et *explaires_vendus*). On parcourt ensuite la base de données ouvrage par ouvrage via un *dolist* et on incrémente les variables de façon appropriée (*nombre_ouvrage* +1 et *exemplaires_vendus* + nombre d'exemplaires vendus de l'ouvrage). On teste ensuite le nombre d'ouvrage comptabilisés : s'il est supérieur à zéro, on effectue la moyenne, sinon on ne renvoie rien (*NIL*).