TP1 – Compte-Rendu

Exercice 1 – Mise en condition

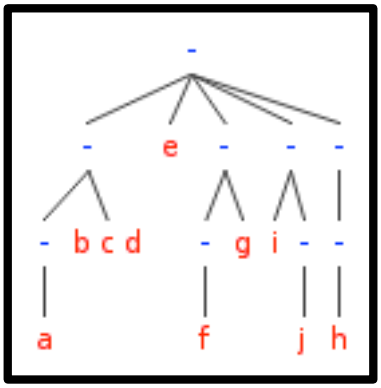
Forme préfixée de l'expression : $4x^3 - 5x^2 + 3x + 1$

```
-> (+ (- (* 4 (^ x 3) (* 5 (^ x 2)) (+ (* 3 x) 1))
```

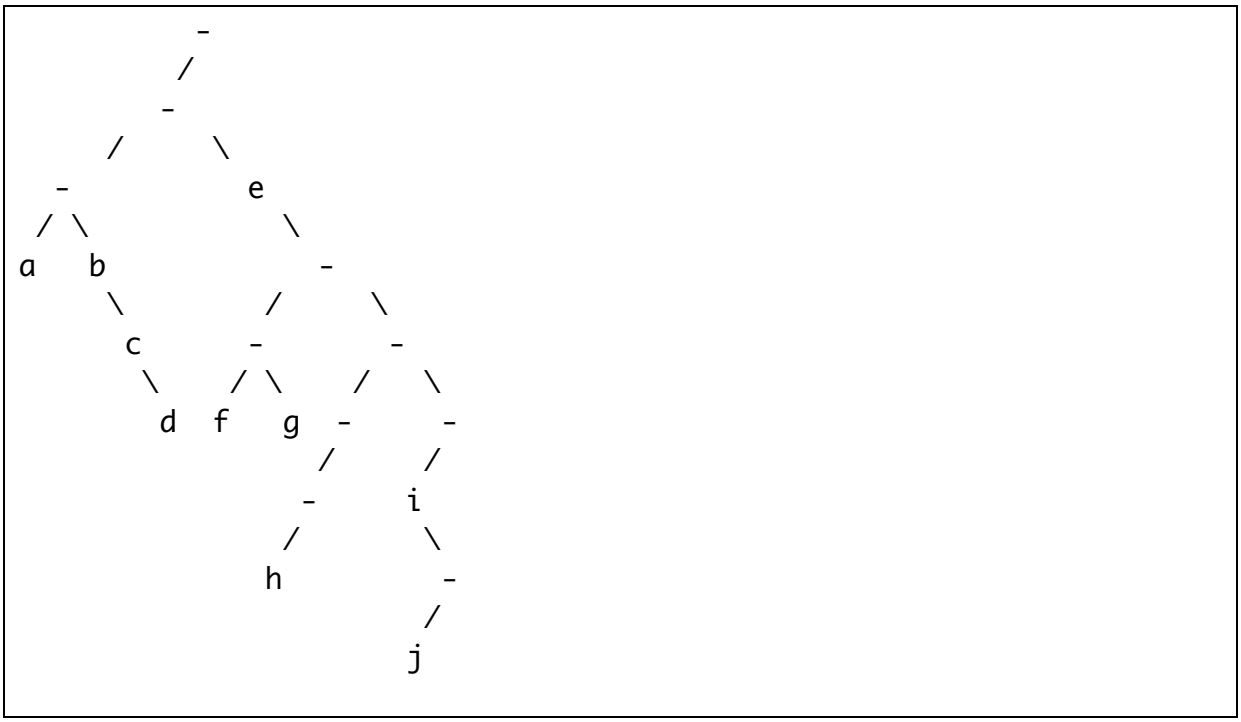
Représenter par une arborescence de type arbre généalogique, puis par un arbre binaire la liste :

```
(((a) b c d) e ((f) g) (((h))) (i (j)))
```

Arbre généalogique : On différencie facilement les fils des frères



Arbre binaire : Une branche gauche désigne un fils, une branche droite désigne un frère.



Firstn(N L) : retourne la tête formée par les N premiers éléments de la liste L :

```
(defun firstn (n l)
```

```
      (if (and (> n 0) l)
            ;; construction du résultat a partir du 1er élément & récursion
            (cons (car l) (firstn (- n 1) (cdr l)))))

(defun firstnr (n l)
      (loop for i from 0 to (- n 1)
            collect (nth i l)))

(firstni 2 '(2 3 4 5)
-> (2 3)
(firstnr 3 '(2 3 4 5 6))
-> (2 3 4)
```

Pour la méthode récursive, on utilise la méthode cons, qui construit une paire, à partir du premier élément de la liste et du cdr. Pour cela, on décrémente n.

Pour la méthode itérative, on parcourt la liste via loop, de 0 à n-1. On part de 0 car la fonction nth admet que le premier élément de la liste soit l'élément 0.

Inter(l m) : retourne l'intersection des deux listes L et M :

```
; recursive
(defun interr (l m)
      (if l
            ;;on regarde si le premier élem de l est membre de m
            (if (member (car l) m)
                  ;; on ajoute le premier elem de l au début de la liste
                  (cons (car l) (interr (cdr l) m))
                  ;; on teste l'intersection du reste de l avec m
                  (interr (cdr l) m))))
; iterative
(defun interi (l m)
      (loop for i in l ; pour chaque elem de la liste l
            when (member i m) ; on regarde s'il existe dans m
            collect i)) ; si oui on l'ajoute

(interr '(2 3 4 5) '(1 4 5 6 7))
-> (4 5)
(interi '(2 3 4 5) '(1 4 5 6 7))
-> (4 5)
```

Dans les deux cas, on assume que les listes soient correctement formées et qu'il n'y a pas de doublons.

Pour la méthode récursive, on utilise la même méthode que précédemment. Si la liste n'est pas nulle, on compare si son premier élément est membre de m, si oui on construit une liste, sinon on passe à la suite de la liste via récursivité.

Pour la méthode itérative, on utilise la même méthode que précédemment sauf qu'on vérifie que l'élément traité est membre de m avant de le collecter.

elim(L) : retourne la liste L après élimination des répétitions :

```
(defun elimr (l)
      (if l
            (if (member (car l) (cdr l))
                  (elimr (cdr l))
```

```
                    (cons (car l) (elimr (cdr l))))))
(elimr '(a z g a s z d g z s))
-> (A D G Z S)
```

On vérifie d'abord que la liste ne soit pas vide, puis que son premier membre (car) soit bien membre du reste de la liste (cdr). Si c'est le cas on passe à la suite, sinon on l'ajoute.

nbFeuilles (L) : retourne le nombre de feuilles d'un arbre généalogique représenté par une liste :

```
(defun nbFeuilles(L)
      (cond
            ((null L) 0)
            ((atom L) 1)
            (T (+ (nbFeuilles (car L)) (nbFeuilles (cdr L))))))

(nbFeuilles '(r ((t)) y (g h) (j m l) p))
-> 9
```

On teste plusieurs cas :

- Si l'élément est NIL, on ajoute rien,
- Si l'élément est un atome (donc une feuille) on ajoute 1
- Si l'élément n'est ni null, ni un atome, alors c'est une branche de l'arbre et on traite ses fils

monEqual : retourne t ou nil selon l'égalité de ses deux arguments.

```
(eq 'LUC 'LUC)  -> T
(equal 'LUC 'LUC)  -> T

(eq 'LUC 'DANIEL)  -> NIL
( equal 'LUC 'DANIEL)  -> NIL

(eq (car '(do re)) (cadr '(mi do sol)))  -> T
( equal (car '(do re)) (cadr '(mi do sol)))  -> T

(eq '(d p f t r) '(d p f t r))  -> NIL
( equal '(d p f t r) '(d p f t r))  -> T
```

La fonction eq retourne T si les deux arguments sont exactement les mêmes objets (même adresse).

La fonction equal retourne T uniquement si la représentation (print) des deux arguments est identique. Cette fonction est équivalente au mot "identique" en français. Lorsque deux objets sont dit identiques, c'est que l'un ressemble à l'autre.

```
(defun monEqual(arg1 arg2)
      ;; Si on a deux atomes
      (if (and (atom arg1) (atom arg2))
      ;; alors on regarde si il sont eq
            (eq arg1 arg2)
            ;; Sinon il faut continuer à descendre pour arriver jusqu'aux atomes
            ;; On regarde si nombre égaux d'éléments, sinon c'est pas la peine
            (if (not(= (length arg1) (length arg2)))
                  ;; si pas le même nombre d'elt alors c'est terminé
                  nil
                  ;; sinon on peut continuer
                  (if (and
```

```
                        (monEqual (car arg1) (car arg2))
                        (monEqual (cdr arg1) (cdr arg2)))
                        T))))

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

Exercice 2 – Objets fonctionnels

Pour cet exercice, il suffit juste de montrer que l'on sait utiliser la fonction mapcar

```
(defun list-paire (L M) (mapcar (lambda (a b) (list a b)) L M))
(liste-paire '(0 2 3 11) '(6 10 20 30))
-> ((0 6) (2 10) (3 20) (11 30))
```

mapcar opère sur les éléments successifs de listes. La fonction passée en paramètre (ici (list a b)) est appliquée aux premiers éléments de chaque liste, puis au second, etc.. Cette itération se termine après avoir traité le dernier élément de la plus petite liste. La valeur retournée par mapcar est la liste des résultats des appels successifs à la fonction.

Exercice 3 – a-list

```
(defun my-assoc(cle a-liste)
      (cond
            ((null a-liste) nil)
            ((equal cle (car(car a-liste))) (car a-liste))
            ((my-assoc cle (cdr a-liste)))))

(my-assoc 'Pierre '((Yolande 25) (Pierre 24) (Julie 45)))
-> (PIERRE 24)
(my-assoc 'Yves '((Yolande aime Pierre) (Pierre aime Julie) (Julie aime Pierre)))
-> NIL
```

On regarde d'abord si la clé de la 1ere paire est égale à celle que l'on cherche, si oui alors on retourne cette paire, sinon on passe à la paire suivante via le cdr de la liste.

Exercice 4 – Gestion d'une base de connaissances en Lisp

A. Fonctions de service

Définition de BaseTest :

```
(setq base '(
      ("Le dernier jour d'un condamne" "Hugo" 1829 50000)
      ("Les Miserables" "Hugo" 1862 2000000)
      ("Le Horla" "Maupassant" 1887 2000000)
      ("Contes de la becasse" "Maupassant" 1883 500000)
      ("Germinal" "Zola" 1885 3000000)))
```

NB : Ces fonctions étant triviales, il est inintéressant de reporter les tests.

```
(defun auteur (ouvrage) (cadr ouvrage))
(defun titre (ouvrage) (car ouvrage))
(defun annee (ouvrage) (caddr ouvrage))
(defun nombre (ouvrage) (cadddr ouvrage))
(setq ouvrage-test '("Les Miserables" "Hugo" 1862 2000000))
(titre ouvrage-test) -> "Les Misérables"
(auteur ouvrage-test) -> "Hugo"
(annee ouvrage-test) -> 1862
(nombre ouvrage-test) -> 2000000
```

B. Autres fonctions

- FB1 : affiche tous les ouvrages

```
(defun FB1 (liste-ouvrages)
      (loop for x in liste-ouvrages
            do (print x)))

(fb1 base)
-> ("Le dernier jour d'un condamne" "Hugo" 1829 50000)
   ("Les Miserables" "Hugo" 1862 2000000)
   ("Le Horla" "Maupassant" 1887 2000000)
   ("Contes de la becasse" "Maupassant" 1883 500000)
   ("Germinal" "Zola" 1885 3000000)
```

Nous utilisons ici une simple boucle for x in liste pour parcourir tous les éléments de la liste, que l'on affiche au fur et à mesure.

- FB2 : affiche les ouvrages dont l'auteur est Hugo

Itératif :

```
(defun FB2 (liste-ouvrages)
      (loop for x in liste-ouvrages do
            (if (equal (auteur x) "Hugo")
                  (print x))))
(fb2 base)
-> (("Le dernier jour d'un condamne" "Hugo" 1829 50000)
    ("Les Miserables" "Hugo" 1862 2000000))
```

Ici, la fonction est presque identique à FB1, sauf que nous rajoutons un test afin de vérifier si l'auteur correspond bien à celui passer en paramètre.

Récursif :

```
(defun fb2 (b)
      (if b
            (if (equal "Hugo" (auteur (car b)))
                  (cons (car b) (fb2 (cdr b)))
                  (fb2 (cdr b)))))
(fb2 base)
-> (("Le dernier jour d'un condamne" "Hugo" 1829 50000)
    ("Les Miserables" "Hugo" 1862 2000000))
```

- FB3 : retourne la liste des titres d'ouvrage dont l'auteur est précisé en argument

```
(defun FB3 (liste-ouvrages nomAuteur)
      (if liste-ouvrages
            (if (equal (auteur (car liste-ouvrages)) nomAuteur)
```

```
                    (cons (titre (car liste-ouvrages)) (FB3 (cdr liste-ouvrages)
nomAuteur))
                    (FB3 (cdr liste-ouvrages) nomAuteur))))
(fb3 base "Maupassant")
-> (("Le Horla" "Maupassant" 1887 2000000)
    ("Contes de la becasse" "Maupassant" 1883 500000))
```

Nous parcourons en récursif tous les ouvrages. Si l'auteur de l'ouvrage actuel (c'est à dire car liste-ouvrages) correspond à celui dont on veut les ouvrages, on construit une liste avec les prochains appels de FB3. Sinon, on appelle simplement la fonction sur le reste des ouvrages (cdr liste-ouvrages).

- FB4 : retourne le premier ouvrage paru en année X (X = un argument) ou nil

```
(defun FB4 (liste-ouvrages pAnnee)
      (if liste-ouvrages
            (if (equal (annee (car liste-ouvrages)) pAnnee)
                  (car liste-ouvrages)
                  (FB4 (cdr liste-ouvrages) pAnnee))))
(fb4 base 1862)
-> ("Les Miserables" "Hugo" 1862 2000000)
(fb4 base 1863)
-> NIL
```

Etant donné que l'on cherche à retourner le premier élément qui correspond à la date, il suffit d'appeler en récursif FB4 jusqu'à trouver cet élément, puis de retourner cet élément. Sinon, la fonction retourne nil.

- FB5 : retourne la liste des ouvrages dont le nombre d'exemplaires vendus dépasse 1000000

```
(defun FB5 (liste-ouvrages)
      (if liste-ouvrages
            (if (> (nombre (car liste-ouvrages)) 1000000)
                  (cons (car liste-ouvrages) (FB5 (cdr liste-ouvrages)))
                  (FB5 (cdr liste-ouvrages)))))
(fb5 base)
-> (("Les Miserables" "Hugo" 1862 2000000) ("Le Horla" "Maupassant" 1887 2000000)
    ("Germinal" "Zola" 1885 3000000))
```

Le principe ici reste identique aux autres fonctions. Nous parcourons de manière récursive la liste passée en argument. Nous testons si le nombre d'exemplaire vendus est supérieur à 1000000. Si le résultat est vrai, alors on construit la liste avec cet ouvrage et le résultat de l'appel de FB5. Sinon, nous ne construisons pas (= nous n'ajoutons pas) la liste avec cet élément, mais nous continuons à appeler FB5.

- FB6 : calcule et retourne la moyenne du nombre d'exemplaires vendus de l'auteur X

```
(defun FB6 (liste-ouvrages pAuteur)
      (setq nbOuvrages 0)
      (setq nbOuvragesVendus 0)
      (loop for x in liste-ouvrages do
            (if (equal (auteur x) pAuteur)
                  (progn
                  (setq nbOuvrages (+ nbOuvrages 1))
                  (setq nbOuvragesVendus (+ nbOuvragesVendus (nombre x))))))
      (if (= nbOuvrages 0)
            0
            (/ nbOuvragesVendus nbOuvrages)))
```

```
(fb6 base "Hugo")
-> 1025000
```

Nous retrouvons une autre fonction itérative.

Ici, nous initialisons nbOuvrages et nbOuvragesVendus à 0 pour nous permettre de comptabiliser le nombre d'ouvrages vendus par un auteur, et combien d'ouvrages nous avons analysé.

En parcourant les ouvrages, nous testons si l'auteur est bien celui qui nous intéresse. Si c'est le cas, on incrémente de un ces deux variables. Notons ici l'utilisation de l'instruction 'progn' afin de créer un bloc et de pouvoir exécuter deux actions dans l'if.

Lorsque nous avons parcouru tous les ouvrages et si nous en avons trouvé au moins un, alors on calcule la moyenne des ouvrages vendus par ouvrage édité. Sinon, nous retournons 0.