IA01 : Rapport du TP1

Auteurs :
Camille Gerin-Roze
Thomas Perrin

Le 2 Octobre 2015

Table des matières

Chapitre 1

Exercice 1

1.1 Donner les premiers N éléments

1.1.1 Version récursive

```
(defun firstn (n l)
(if (> n 0) (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.

1.1.2 Version itérative

```
(defun firstn-ite (n l)
  (let ((listeEntiere l))
    (setq liste ())
  (dotimes (x n liste)
    (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.

1.2 Intersection de 2 listes

1.2.1 Version récursive

```
(defun inter (l1 l2)
  (if (not (null (car l1)))
      (if (member (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.

1.2.2 Version itérative

```
(defun inter-iteratif (l1 l2)
    (mapcan #'(lambda (x) (if (member 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.

1.3 Elimination des doublons d'une liste

1.3.1 Version récursive

```
(defun elim (l1)
(if (null (car l1)) ()
  (if (member (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).

1.3.2 Version itérative

```
(defun elim-ite (l1)
  (let ((listResult ()))
    (dolist (x l1 listResult)
      (if (and (member x l1) (not (member 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.

1.4 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.

1.5 Implémentation de la méthode EQUAL

```
(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.