6. LA REPETITION

6.1. LA FONCTION REVERSE

Tout langage de description de processus, donc tout langage de programmation, doit comporter des possibilités de décrire des activités répétitives. LISP exprime une répétition de manière particulièrement élégante, facile et efficace. Regardons d'abord un exemple d'une fonction qui inverse les éléments d'une liste quelconque :

```
(DE REVERSE (L RES)
    (IF (NULL L) RES
        (REVERSE (CDR L) (CONS (CAR L) RES))))
```

et voici quelques appels :

(REVERSE '(DO RE MI FA) NIL)	→	**(FA MI RE DO)**
(REVERSE '(COLORLESS GREEN IDEAS) NIL)	→	**(IDEAS GREEN COLORLESS)**
(REVERSE '(A B (C D) E F) NIL)	→	**(F E (C D) B A)**

Pour comprendre l'évaluation d'un appel de cette fonction, donnons-en d'abord une description de son comportement : imaginons que nous ayons deux lieux, l'un nommé **L**, l'autre nommé **RES**, chacun dans un état initial :

L	**RES**
(A B C D)	()

Pour inverser la liste **(A B C D)** de **L**, on va transférer un élément après l'autre de **L** vers **RES** en introduisant à chaque instant l'élément sortant de **L** en tête de **RES**. Voici graphiquement une *trace* des états successifs de **L** et **RES** :

	L	**RES**
état initial	**(A B C D)** ↓	(↑)
opération		
première transaction	**(B C D)** ↓	(↑ **A)**
opération		
deuxième transaction	**(C D)** ↓	(↑ **B A)**
opération		
troisième transaction	**(D)** ↓	(↑ **C B A)**
opération		
dernière transaction	()	**(D C B A)**

Le programme **REVERSE** se comporte exactement de la même façon : il transfère élément par élément de

L vers l'accumulateur **RES**, et s'arrête quand la liste **L** est vide.

L'écriture de cette fonction peut être facilement comprise si l'on raisonne de la manière suivante : pour inverser une liste il faut d'abord regarder si la liste est vide. Si oui, alors il n'y a rien à faire et le résultat sera la liste accumulée au fur et à mesure dans la variable **RES**. D'ailleurs, ceci est naturellement vrai au début, puisque **RES** sera égal à **NIL**, la liste vide, au premier appel. La liste vide est bien l'inverse de la liste vide ! Ensuite, il faut prendre le premier élément de la liste **L** et le mettre en tête de la liste **RES**.

Reste ensuite à répeter la même chose avec les éléments restants, *jusqu'à ce que la liste* **L** *soit vide*. Si maintenant on trouvait une fonction pour faire ce travail, il suffirait de l'appeler. Rappelons nous que nous avons une fonction qui satisfait à cette demande : la fonction **REVERSE** qu'on est en train de définir. Donc, appelons la avec le reste de la liste **L** et l'accumulation, dans **RES**, du premier élément de **L** et de **RES**. Remarquons qu'il est nécessaire de modifier l'argument **L** de telle manière qu'il converge vers la satisfaction du test d'arrêt **(NULL L)**.

L'exécution de l'appel

(REVERSE '(DO RE MI) NIL)

se fait donc comme suit :

1. d'abord **L** est liée à la liste **(DO RE MI)** et **RES** à **NIL**. Ensuite le corps de la fonction est évalué avec ces liaisons. Première chose à faire : tester si **L** est vide. Pour l'instant ce n'est pas le cas. Il faut donc évaluer l'*action-si-faux* du **IF**, qui dit qu'il faut calculer **(REVERSE (CDR L)(CONS (CAR L) RES)**. Avec les liaisons valides pour l'instant, cela revient au même que de calculer

   (REVERSE '(RE MI) '(DO))

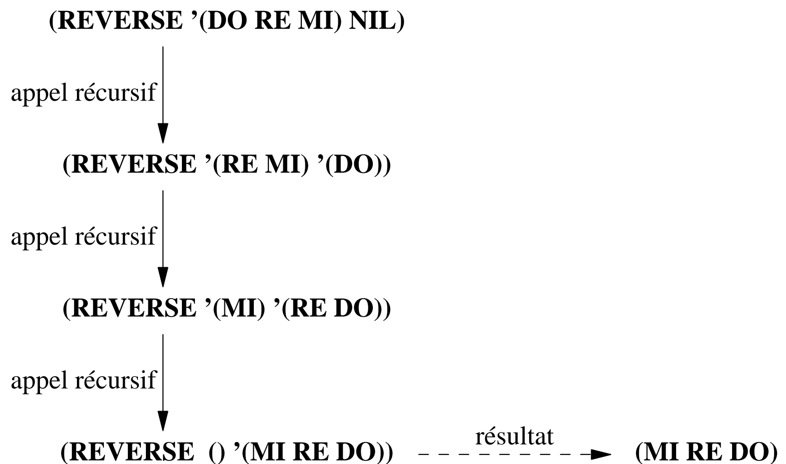
   Pour connaiˆtre le résultat de l'appel initial, il suffit donc de connaitre le résultat de ce nouvel appel.
2. On entre alors de nouveau dans la fonction **REVERSE**, liant la variable **L** à la nouvelle valeur **(RE MI)** et la variable **RES** à **(DO)**. Comme **L** n'est toujours pas égale à **NIL**, il faut, pour connaiˆtre le résultat de cet appel, une fois de plus évaluer l'*action-si-faux* de la sélection. Cette fois ce sera

   (REVERSE '(MI) '(RE DO))

   c'est-à-dire le **CDR** de **L** et le résultat de **(CONS (CAR L) RES)** avec les liaisons actuelles.
3. la même opération se répète : nouvelles liaisons des variables, **L** n'étant toujours pas égale à **NIL** il faut re-appeler **REVERSE**

   (REVERSE () '(MI RE DO))
4. cette fois-ci la variable **L** est liée à **NIL** et le test **(NULL L)** est donc vérifié. Le résultat du dernier appel de **REVERSE** est donc la valeur de **RES**, la liste **(MI RE DO)**. Mais rappelons nous que cet appel n'avait lieu que pour connaitre le résultat de l'appel précédent, c'est donc également le résultat de l'appel précédent. Ainsi de suite, jusqu'à ce qu'on arrive à l'appel initial, et la machine peut ramener cette liste en valeur.

Graphiquement, on peut représenter l'évaluation de cet appel comme suit :



Des fonctions qui contiennent à l'intérieur de leur *corps* un appel à elles-mêmes sont dites des *fonctions-récursives*. La fonction **REVERSE** est donc une fonction récursive.

Voici une *trace* de l'exécution du même appel :

```
? (REVERSE '(DO RE MI) ())
  ---> REVERSE : ((DO RE MI) NIL)
    ---> REVERSE : ((RE MI) (DO))
      ---> REVERSE : ((MI) (RE DO))
        ---> REVERSE : (() (MI RE DO))
  <------------ REVERSE = (MI RE DO)
= (MI RE DO)
```

NOTE

Notons, qu'en VLISP il est possible d'omettre, à l'appel, les derniers arguments si l'on veut qu'ils soient liés à la valeur **NIL**. Les appels exemples de la fonction **REVERSE** peuvent donc également s'écrire de la manière suivante :

(REVERSE '(DO RE MI FA))	→	**(FA MI RE DO)**
(REVERSE '(COLORLESS GREEN IDEAS))	→	**(IDEAS GREEN COLORLESS)**
(REVERSE '(A B (C D) E F))	→	**(F E (C D) B A)**

En LE_LISP, par contre, tous les arguments doivent être *impérativement* fournis, même quand ceux-ci sont égaux à ().

6.2. LA FONCTION APPEND

Prenons un deuxième exemple d'une fonction récursive : la fonction **APPEND** qui doit concaténer deux listes comme dans les exemples suivants :

(APPEND '(A B) '(C D))	→	**(A B C D)**
(APPEND '(A) '(B C))	→	**(A B C)**
(APPEND NIL '(A B C))	→	**(A B C)**

Cette fonction possède deux paramètres (pour les deux listes passées en argument) et elle retourne une liste

qui est la concaténation de ces deux listes. Cela nous donne déjà le début de l'écriture de la fonction :

```
(DE APPEND (LISTE1 LISTE2)
   . . .
```

Si nous regardons les exemples de la fonction **APPEND** (ces exemples peuvent être compris comme sa *spécification*), on observe que si le premier argument, **LISTE1**, est vide, le résultat est la liste passée en deuxième argument, c.à.d.: **LISTE2**, et si **LISTE1** a un seul élément, le résultat est la liste **LISTE2** avec comme nouveau premier élément, l'élément unique du premier argument. Malheureusement, nous savons déjà que nous ne pouvons pas *conser*[1] un élément après l'autre de la première liste avec la deuxième, puisque cette méthode était utilisée dans la fonction **REVERSE** : elle introduirait les éléments dans l'ordre inverse. Il faudrait pouvoir commencer avec le dernier élément de la liste **LISTE1**, l'introduire dans la liste **LISTE2**, pour ensuite y introduire l'avant dernier élément et ainsi de suite, jusqu'à ce que nous soyons au premier élément de **LISTE1**. On pourrait inverser d'abord la liste **LISTE1** et ensuite appliquer la fonction reverse avec comme arguments cette liste inversée et la liste **LISTE2**. Ce qui nous donne :

```
(DE APPEND (LISTE1 LISTE2)
      (REVERSE (REVERSE LISTE1) LISTE2))
```

Mais, bien que l'écriture de cette fonction soit très brève, elle est terriblement inefficace, puisque la liste **LISTE1** est parcourue 2 (*deux* !) fois dans toute sa longueur. Imaginez le temps que cela prendrait si la liste était très, très longue.

On peut faire exactement la même chose plus efficacement, si on sépare le parcours de la liste **LISTE1** de la construction de la liste résultat : d'abord parcourons la liste premier argument, en mémorisant les differents éléments rencontrés, ensuite *consons*[2] chaque élément, en commençant par le dernier, à la liste **LISTE2**. Ce qui nous donne cette deuxième définition :

```
(DE APPEND (LISTE1 LISTE2)
   (IF (NULL LISTE1) LISTE2
      (CONS (CAR LISTE1) (APPEND (CDR LISTE1) LISTE2))))
```

qui peut être paraphrasée comme suit :

Pour concaténer deux listes, il faut distinguer deux cas :

1. si la première liste est vide, le résultat sera la deuxième liste
2. sinon, il suffit d'introduire le premier élément en tête de la liste résultant de la concaténation du reste de la première liste et de la deuxième liste, tout naturellement, puisqu'on avait déjà vu que la concaténation d'une liste à un seul élément est justement le résultat du **CONS** de cet unique élément et de la deuxième liste.

Graphiquement, l'exécution de l'appel

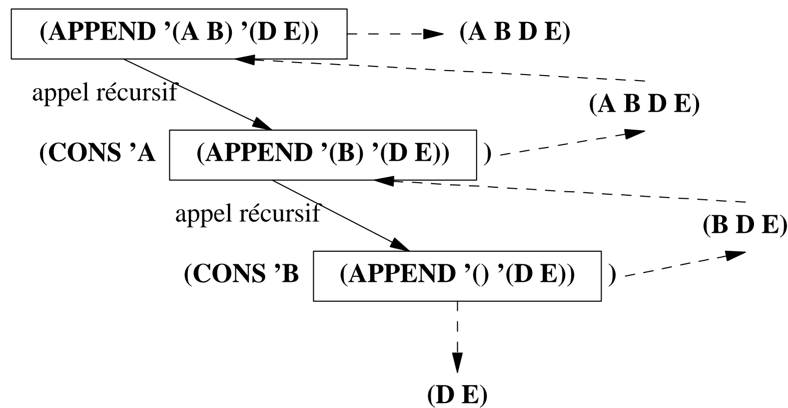
(APPEND '(A B) '(D E))

peut être représentée de la manière suivante :

---

[1] du verbe *"conser"* (du premier groupe), qui se conjugue tout normalement : je conse, tu conses, .. etc et qui est synonyme de 'faire un **CONS**', ou 'allouer une cellule´.

[2] voir la note précédente.



On voit clairement qu'en *descendant* dans les appels récursifs successifs, la machine accumule des **CONS**'s qu'elle ne peut calculer qu'après avoir évalué les arguments (ici, un des arguments est toujours un appel récursif de **APPEND**). Le calcul des expressions **CONS** est effectué pendant la *remontée* de la récursion.

Voici une *trace* d'un appel de la fonction **APPEND** :

```
? (APPEND '(A B C) '(D E F))                 ; l'appel initial ;
   ---> APPEND : ((A B C) (D E F))                 ; sa trace ;
     ---> APPEND : ((B C) (D E F))    ; le deuxième appel, donc: ;
                                        ; le 1-er appel récursif ;
       ---> APPEND : ((C) (D E F))          ; le troisième appel ;
         ---> APPEND : (NIL (D E F))        ; le quatrième appel ;
         <--- APPEND = (D E F)       ; le résultat du 4-ième appel ;
       <--- APPEND = (C D E F)       ; le résultat du 3-ième appel ;
     <--- APPEND = (B C D E F)       ; le résultat du 2-ième appel ;
   <--- APPEND = (A B C D E F)        ; le résultat du 1-er appel ;
= (A B C D E F)                  ; le résultat de l'appel initial ;
```

En comparant les deux *traces* de **REVERSE** et de **APPEND**, nous constatons qu'il existe deux sortes de boucles récursives :

1. les boucles récursives qui font tout le calcul en *descendant*, c'est-à-dire : comme dans la fonction **REVERSE**, le calcul est effectué entièrement à travers les appels récursifs. Une fois arrivé au *test de sortie* de la récursivité (ici, le test est l'expression **(NULL L)**), le calcul est terminé et la fonction peut directement ramener cette valeur.
2. les boucles récursives qui font une partie du calcul en *descendant* les appels récursifs et l'autre partie en *remontant* les appels récursifs successifs. C'est le cas dans la fonction **APPEND** : en descendant, la fonction isole les différents éléments qu'elle *conse* ensuite, pendant la remontée, aux résultats des divers appels récursifs. Remarquez que les éléments isolés pendant la descente sont utilisés, pendant la remontée, dans l'ordre inverse, c'est-à-dire : le premier élément isolé sera le dernier élément utilisé.

Nous allons revenir plus tard sur la différence entre ces deux types de fonctions récursives. Pour l'instant, observons juste, que ces deux types sont aisés à distinguer dans l'écriture même des fonctions : dans le premier type (calcul exclusivement en descendant) le résultat de l'appel récursif n'est pas utilisé par une autre fonction, ou, plus précisément, l'appel récursif ne se trouve pas en position d'argument d'une fonction; dans le deuxième type (calcul en descendant et en remontant), le résultat est utilisé par une autre fonction. Dans la fonction **APPEND**, par exemple, le résultat de chaque appel récursif est utilisé comme deuxième

argument d'un **CONS**.

Remarquons que les appels qui n'engendrent un calcul qu'en descendant s'appellent des *appels récursifs terminaux*.

6.3. LA FONCTION EQUAL

Notre troisième exemple de fonction récursive sera la fonction **EQUAL**, une fonction qui doit tester l'égalité de ses deux arguments. A cette fin nous avons d'abord besoin d'apprendre un nouveau prédicat : le prédicat **EQ** qui teste l'égalité de deux *atomes*. Voici la définition de **EQ** :

(EQ *arg1 arg2***)**	→	**T** si *arg1* et *arg2* sont le même atome
	→	**NIL** si *arg1* et *arg2* sont des atomes différents ou si au moins l'un des deux est une liste

Ci-dessous quelques exemples d'utilisation de la fonction standard **EQ** :

(EQ 'LUC 'LUC)	→	**T**	
(EQ 'LUC 'DANIEL)	→	**NIL**	
(EQ (CAR '(DO RE))(CADR '(MI DO SOL)))	→	**T**	
(EQ '(D P F T R) '(D P F T R))	→	**NIL**	*; !! ;*[4]

Remarquons que **EQ** ramène **NIL**, donc *faux*, si les arguments sont des listes, même si les deux listes sont égales.[5] A priori, il n'existe pas une fonction testant l'égalité de deux listes. D'autre part, nous savons que deux listes peuvent être considérées comme égales si (et seulement si) les éléments des deux listes sont égaux un à un. Tout ce que nous avons à faire alors, c'est comparer les éléments successifs des deux listes en y appliquant la fonction **EQ**. Ce raisonnement nous amène à la définition suivante :

```
(DE EQUAL (ARG1 ARG2)
   (IF (ATOM ARG1)(EQ ARG1 ARG2)
      (IF (ATOM ARG2) NIL
         (IF (EQ (CAR ARG1)(CAR ARG2))
            (EQUAL (CDR ARG1)(CDR ARG2))
            NIL))))
```

Visiblement, cette fonction teste aussi bien l'égalité de deux atomes (la première ligne dit que si **ARG1** est un atome, il suffit de tester son égalité - par un appel de **EQ** - avec **ARG2**) que l'égalité de deux listes (en testant d'abord l'égalité des deux premières éléments et en répétant ce processus avec les **CDR**'s des deux listes).

Notons l'utilisation du prédicat **ATOM** : il sert à la fois à tester si l'on a affaire à un atome *et* de test d'arrêt de la récursivité. Ceci est possible puisque la liste vide est considérée comme l'atome **NIL**. Voici quelques exemples d'appels de cette fonction :

---

[3] "D.P.F.T.R." est un acronyme pour "Du Passé Faisons Table Rase".

[4] L'exception à cette règle est discutée dans le chapitre 15.

(EQUAL '(DO RE MI) '(DO RE MI)) → **T**
(EQUAL 'TIENS 'TIENS) → **T**
(EQUAL '(DO RE MI) (CDR '(DO RE MI))) → **NIL**
(EQUAL '(LE NEZ (DE CLEOPATRE)) '(LE NEZ (DE CLEOPATRE))) → **NIL**[7]

Evidemment la fonction ne marche pas si les listes données en argument contiennent des sous-listes. Regardez voici une trace de ce dernier exemple :

```
? (EQUAL '(LE NEZ (DE CLEOPATRE)) '(LE NEZ (DE CLEOPATRE)))
  ---> EQUAL : ((LE NEZ (DE CLEOPATRE)) (LE NEZ (DE CLEOPATRE)))
    ---> EQUAL : ((NEZ (DE CLEOPATRE)) (NEZ (DE CLEOPATRE)))
      ---> EQUAL : (((DE CLEOPATRE)) ((DE CLEOPATRE)))
  <--------- EQUAL = NIL
=  NIL
```

Le test **(EQ (CAR ARG1)(CAR ARG2))** ramène **NIL** dès que le premier élément est une liste ! Pour réparer cette insuffisance, il suffit de tester non pas l'égalité **EQ**, mais l'égalité **EQUAL** des **CAR**'s successifs. Ce qui nous donne la version modifiée que voici :

```
(DE EQUAL (ARG1 ARG2)
     (IF (ATOM ARG1)(EQ ARG1 ARG2)
        (IF (ATOM ARG2) NIL
           (IF (EQUAL (CAR ARG1)(CAR ARG2))
              (EQUAL (CDR ARG1)(CDR ARG2))
              NIL))))
```

et voici une trace de cette nouvelle fonction sur le même exemple :

```
? (EQUAL '(LE NEZ (DE CLEOPATRE)) '(LE NEZ (DE CLEOPATRE)))
  ---> EQUAL : ((LE NEZ (DE CLEOPATRE)) (LE NEZ (DE CLEOPATRE)))
    ---> EQUAL : (LE LE)
    <--- EQUAL = T
    ---> EQUAL : ((NEZ (DE CLEOPATRE)) (NEZ (DE CLEOPATRE)))
      ---> EQUAL : (NEZ NEZ)
      <--- EQUAL = T
      ---> EQUAL : (((DE CLEOPATRE)) ((DE CLEOPATRE)))
        ---> EQUAL : ((DE CLEOPATRE) (DE CLEOPATRE))
          ---> EQUAL : (DE DE)
          <--- EQUAL = T
          ---> EQUAL : ((CLEOPATRE) (CLEOPATRE))
            ---> EQUAL : (CLEOPATRE CLEOPATRE)
            <--- EQUAL = T
            ---> EQUAL : (NIL NIL)
        <--------- EQUAL = T
        ---> EQUAL : (NIL NIL)
  <------------ EQUAL = T
=  T
```

Remarquez que cette fonction contient deux appels récursifs : le résultat du premier appel est utilisé comme

[5] pour en savoir plus sur "le nez de Cléopâtre" regardez donc l'excellent article de Raymond Queneau sur *la littérature définitionnelle* dans *Oulipo, la littérature potentielle*, pp. 119-122, idées/Gallimard.

test du **IF**, et le deuxième appel nous livre le résultat. Cette fonction contient donc un appel récursif terminal (le deuxième appel récursif) et un qui ne l'est pas (le premier appel récursif).

Une dernière remarque sur cette fonction : très souvent il peut arriver que vous ayez à écrire une fonction avec un ensemble de **IF**s imbriqués, comme ici, où vous aviez trois **IF**s les uns dans les autres. Pour de tels cas il existe une abréviation, à la fois plus lisible et plus puissante : c'est la fonction **COND**, une fonction de sélection généralisée. Elle est syntaxiquement définie comme suit :

(COND $clause_1$ $clause_2$ ... $clause_n$**)**

et chacune des *clauses* doit être de la forme :

(*test* {$action_1$ $action_2$ ... $action_n$}**)**

L'évaluation de la fonction **COND** procède comme ceci :

le *test* de la première clause ($clause_1$) est évalué

[1] si son évaluation ramène la valeur **NIL**
alors s'il y a encore des clauses
on continue avec l'évaluation du *test* de la clause suivante et on continue en [1]
sinon on arrête l'évaluation du **COND**
en ramenant **NIL** en valeur
sinon on évalue en séquence les act_1 à act_n de la clause courante et on sort du **COND** en ramenant en valeur la valeur de l'évaluation de act_n. (s'il n'y a pas d'*act*ions à évaluer on sort du **COND** en ramenant en valeur la valeur de l'évaluation de *test*)

[2] si aucun des *test*s n'est satisfait, la valeur du **COND** est **NIL**.

Ceci signifie que l'expression :

```
(IF test1 act1
   (IF test2 act2
     (IF test3 act3
         . . .
       (IF testn actn1 actn2 .. actnm) . . . )))
```

est équivalente à l'expression :

```
(COND
   (test1 act1)
   (test2 act2)
   (test3 act3)
      . . .
   (testn actn1)
   (T actn2 ... actnm))
```

La structure du **COND** est beaucoup plus lisible que celle des **IF**'s imbriqués. Nous pouvons ainsi écrire la fonction **EQUAL** dans une forme plus élégante (mais faisant exactement la même chose) :

```
(DE EQUAL (ARG1 ARG2)(COND
    ((ATOM ARG1)(EQ ARG1 ARG2))
    ((ATOM ARG2) NIL)
    ((EQUAL (CAR ARG1)(CAR ARG2))(EQUAL (CDR ARG1)(CDR ARG2))) ))
```

6.4. LA FONCTION DELETE

Regardons en détail un dernier exemple d'une fonction récursive. Cette fois le problème sera d'écrire une fonction qui enlève toutes les occurrences d'un élément donné d'une liste. Cette fonction, que nous nommerons **DELETE**, aura donc deux arguments : un premier argument donnant l'élément à éliminer, et un deuxième argument donnant la liste à l'intérieur de laquelle on veut éliminer cet élément. Sa construction sera tout à fait *immédiate*.

Rappelons que dans toutes les fonctions à structure répétitive, il existe *toujours* au moins un test d'arrêt et au moins un appel récursif à l'intérieur duquel au moins un des arguments doit être modifié de manière telle que les valeurs successives de cet argument convergent vers la satisfaction du test d'arrêt.

Pour pouvoir *enlever* d'une liste un élément donné, il faut naturellement d'abord *chercher* cet élément. Nous venons juste de voir un algorithme de recherche d'un élément dans la fonction **EQUAL**. Rien ne nous empêche de l'utiliser à nouveau. La seule différence ici est que nous ne nous satisferons pas avec le fait d'avoir trouvé *une* occurrence de l'élément en question, mais que nous voulons en trouver toutes les occurrences. D'autre part, un algorithme uniquement de recherche n'est pas suffisant, puisque chaque fois que nous avons trouvé une occurrence nous voulons l'éliminer. De plus, nous voulons garder tous les autres éléments de la liste. De cette réflexion s'ensuit pratiquement le programme : nous allons comparer élément par élément de la liste avec l'élément particulier, si nous avons affaire à un élément différent, nous le gardons et continuons notre parcours dans le reste de la liste, sinon nous l'enlevons (en fait, nous ne le gardons pas) et nous continuons également à chercher (d'autres occurrences) dans le reste de la liste. Naturellement nous arrêtons quand la liste est vide.

Cet algorithme se traduit aisément en LISP :

```
(DE DELETE (ELE LISTE)(COND
    ((NULL LISTE) NIL)
    ((EQUAL ELE (CAR LISTE))(DELETE ELE (CDR LISTE)))
    (T (CONS (CAR LISTE)(DELETE ELE (CDR LISTE)))) ))
```

Si l'élément **ELE** ne fait pas partie de la liste **LISTE**, cette fonction livre une copie de **LISTE**. Notez également l'utilisation de la fonction **EQUAL** comme prédicat.

Voici quelques exemples d'appel :

```
(DELETE 'A  '(A A H))                  →    (H)
(DELETE '(A) '(A A H))                 →    (A A H)
(DELETE '(A) '(A (A) H))               →    (A H)
(DELETE '(DO RE) '(FA SOL (DO RE)))    →    (FA SOL)
```

et voici une trace de l'exécution d'un appel de la fonction **DELETE**. Regardez à nouveau le mélange entre appels récursifs terminaux et appels récursifs normaux :

```
? (DELETE 'A '(A A H A H)))
  ---> (DELETE A (A A H A H))
    ---> (DELETE A (A H A H))
      ---> (DELETE A (H A H))
        ---> (DELETE A (A H))
          ---> (DELETE A (H))
            ---> (DELETE A NIL)
            <--- DELETE NIL
        <------ DELETE (H)
  <--------- DELETE (H H)
= (H H)
```

6.5. EXERCICES

1. Si l'on appelle **DELETE** comme suit :

(DELETE 'A '(A (B A (C A) A) A))

le résultat sera **((B A (C A) A))**. Changez la fonction de façon qu'elle livre le résultat **((B (C)))**, c'est-à-dire : modifiez la fonction pour qu'elle élimine *toute* occurrence d'un élément donné, indépendamment de la profondeur à laquelle cet élément se situe à l'intérieur de la liste.

2. Ecrivez une fonction qui double chacun des éléments de la liste donnée en argument. Exemples d'appels possibles :

(DOUBLE '(A B C))	→	**(A A B B C C)**
(DOUBLE '(DO (RE MI) FA))	→	**(DO DO (RE MI)(RE MI) FA FA)**
(DOUBLE '(JE BE GAYE))	→	**(JE JE BE BE GAYE GAYE)**

3. Modifiez la fonction **DOUBLE** de l'exercice 3 de manière à doubler tous les atomes d'une liste, indépendamment de la profondeur à laquelle ils se trouvent.

4. Ecrivez une fonction prédicat, à deux arguments, qui ramène *vrai* si le premier argument a une occurrence à l'intérieur de la liste deuxième argument, et qui ramène **NIL** si le premier argument n'a aucune occurrence dans la liste deuxième argument. Ci-dessous quelques exemples d'applications de cette fonction :

(MEMQ 'B '(A B C D))	→	**(B C D)**
(MEMQ 'Z '(A B C D))	→	**()**
(MEMQ 3 '(1 2 3 4))	→	**(3 4)**

5. Modifiez la fonction précédente de manière telle que l'élément cherché puisse se trouver à une profondeur quelconque et puisse être d'un type arbitraire (c'est-à-dire : une liste ou un atome).

6. Ecrivez une fonction qui groupe les éléments successifs de deux listes. Voici quelques exemples montrant ce que la fonction doit faire :

(GR '(A B C) '(1 2 3))	→	**((A 1)(B 2)(C 3))**
(GR '(M N O) '(13 14 15 16))	→	**((M 13)(N 14)(O 15)(16))**
(GR '(M N O P) '(13 14 15))	→	**((M 13)(N 14)(O 15)(P))**

7. Que fait la fonction suivante :

```
(DE FOO (X)
    (IF (NULL X) NIL (APPEND (FOOBAR X X) (FOO (CDR X)))))
```

avec la fonction **FOOBAR** que voici :

```
(DE FOOBAR (X Y)
    (IF (NULL Y) ()
        (CONS X (FOOBAR X (CDR Y)))))
```

8. Que fait la fonction suivante :

```
(DE F (X Y)
    (IF X (CONS (CAR X)(F Y (CDR X))) Y))
```

9. Et que fait la fonction suivante :

```
(DE BAR (X)
    (IF (NULL (CDR X)) X
        (CONS (CAR (BAR (CDR X)))
            (BAR (CONS (CAR X)
                    (BAR (CDR (BAR (CDR X)))))))))
```