16. LES MACRO-FONCTIONS

Pendant la construction de grands programmes, il arrive souvent que l'on construise des fonctions relativement simples juste pour augmenter la lisibilité des programmes. Ainsi, si vous cherchez tre‘s souvent dans un programme le quatrie‘me élément d'une liste, au lieu d'écrire à chaque fois :

(CADR (CDDR *la-liste*))

vous allez probablement définir une fonction nommée **4-IEME**, définie comme suit :

(DE 4-IEME (L) (CADR (CDDR L)))

et remplacer toutes les occurrences des instructions ci-dessus par :

(4-IEME *la-liste*)

rendant ainsi le programme plus lisible.

Le proble‘me avec cette manie‘re de programmer est que l'exécution du programme devient de plus en plus lente : chaque appel d'une fonction utilisateur lance les processus de sauvegarde, liaison et restauration des valeurs des variables. A cause de cette dichotomie entre la lisibilité et le temps d'exécution d'un programme, LISP permet - en plus des fonctions **DE** et **DF** - un type de fonctions supplémentaire : ce sont les *macro-fonctions*, ou - plus simplement - les *macros*. Ce chapitre introduit un aspect de leur utilisation.

Les macro-fonctions se définissent avec la fonction **DM** (pour *d*efine *m*acro-fonction). Cette fonction est syntaxiquement de la même forme que **DF** : un nom, une liste de variables et un corps :

(DM *nom* **(***variable***)** *corps***)** → *nom*
définie une fonction de type *macro*.

La particularité des macro-fonctions est

1. à l'appel, la variable est liée à *la forme de l'appel tout entier*,
2. ensuite le corps est évalué et *le résultat de cette évaluation est encore une fois évalué*. L'évaluation d'une macro se fait donc en deux temps.

Insistons sur le fait que la liaison des variables parame‘tres en LE_LISP est plus générale qu'en VLISP. De même que les fonctions de type **DF** permettent les *mêmes* mécanismes de liaison que les fonctions du type **DE**, les macro-fonctions en LE_LISP permettent également ces mêmes mécanismes.

Ainsi, si une macro-fonction est définie avec plusieurs parame‘tres, à l'appel les différentes variables parame‘tres sont liées aux éléments succéssifs de l'*appel* de cette macro : la premie‘re variable étant liée au premier élément de l'appel, c'est-à-dire au nom de la macro-fonction, la deuxie‘me variable au premier argument de cet appel de macro-fonction, et ainsi de suite.

L'unique définition de macro-fonction que nous considérons ici est celui ou *une* variable est liée à la forme de l'appel entier, ce qui correspond en LE_LISP à des définitions du style :

(DM *nom variable corps***)** → *nom*
définie une fonction de type *macro* où la variable *variable* est liée à l'appel entier.

Ainsi, si nous définissons en VLISP une macro-fonction **FOO** comme :

(DM FOO (X) . . .)

cette même macro-fonction se définit en LE_LISP comme :

(DM FOO X . . .)

Ce sont donc des définitions où la liste de variables est réduite à un atome. Par un mécanisme similaire à celui de la liaison des NEXPRs cet atome est lié à la forme de l'appel de la macro.

Allons pas-à-pas et prenons d'abord un exemple tre‘s simple. Voici la macro-fonction **EX1** :

```
(DM EX1 (L)
    (PRINT "L =" L)
    '(1+  99))
```

Cette macro ne fait rien d'autre qu'éditer son argument et de ramener dans un premier temps la liste **(1+ 99)**. Cette liste est, dans un deuxie‘me temps, réévaluée, ce qui donne pour tout appel de **EX1** le résultat **100**. Essayons :

```
? (EX1)        ; l'appel de la macro-fonction
  L = (EX1)    ; l'impression de l'argument (qui est bien l'appel entier !)
= 100          ; le résultat de l'évaluation de (1+ 99)
```

Mais revenons vers notre fonction **4-IEME** et définissons une macro résolvant le proble‘me. Voici la premie‘re version :

(DM 4-IEME (L) (LIST 'CADR (LIST 'CDDR (CADR L))))

ou, en LE_LISP :

(DM 4-IEME L (LIST 'CADR (LIST 'CDDR (CADR L))))

Notons que le corps de cette macro construit une expression à évaluer, laquelle exprime le calcul du **CADR** du **CDDR** de l'argument donné. Evidemment, en utilisant le macro-caracte‘re *back-quote* (‘), nous pouvons écrire le corps plus simplement comme :

‘(CADR (CDDR ,(CADR L)))

Si nous appelons cette macro comme suit :

(4-IEME '(A B C D E F))

le résultat de la premie‘re évaluation de cet appel est l'expression :

(CADR (CDDR (QUOTE (A B C D E F))))

et l'évaluation supplémentaire de cette expression livre l'atome **D**, résultat de l'appel de notre macro **4-IEME**.

Naturellement, on peut construire des macro-fonctions récursives. Par exemple, la macro **MCONS** ci-dessous s'appelle récursivement jusqu'à ce qu'elle aboutisse à la fin de la liste passée en argument :

```
(DM MCONS (L)
    (IF (NULL (CDDR L)) (CADR L)
        '(CONS ,(CADR L) (MCONS ,@(CDDR L)))))
```

Voici une trace d'un appel de **MCONS** :

```
? (MCONS 1 2 3 4 5 NIL)                    ; l'appel initial ;
  ---> MCONS (MCONS 1 2 3 4 5 NIL)         ; la trace de cet appel initial ;
  <--- MCONS (CONS 1 (MCONS 2 3 4 5 NIL))  ; son résultat ;
  ---> MCONS (MCONS 2 3 4 5 NIL)           ; le premier appel récursif ;
  <--- MCONS (CONS 2 (MCONS 3 4 5 NIL))    ; son résultat
  ---> MCONS (MCONS 3 4 5 NIL)
  <--- MCONS (CONS 3 (MCONS 4 5 NIL))
  ---> MCONS (MCONS 4 5 NIL)
  <--- MCONS (CONS 4 (MCONS 5 NIL))
  ---> MCONS (MCONS 5 NIL)
  <--- MCONS (CONS 5 (MCONS NIL))
  ---> MCONS (MCONS NIL)
  <--- MCONS NIL
= (1 2 3 4 5)
```

A chaque appel, la variable est liée à cet appel en entier et son évaluation s'éxécute en deux temps : chaque appel récursif est effectué *apre's* la sortie de l'appel précédent. Ce n'est pas réellement une exécution récursive !

Toutefois, bien que nous ayons introduit les macro-fonctions en argumentant sur les temps d'exécution, nous ne pouvons pas réellement dire que nous avons amélioré quoi que ce soit en les introduisant : tout au contraire, les macro-fonctions sont encore plus lentes que les fonctions utilisateurs normales. Revenons alors à notre définition de **DM** : nous disions que la variable d'une macro-fonction est liée à *l'appel même* de cette macro-fonction. C'est là le point important ! Ceci nous permet de modifier physiquement l'appel de cette macro, à la premie're évaluation, par la suite d'instruction qui définit le corps de la macro. Ainsi, les appels de macro-fonctions se détruisent eux-mêmes, assurant par là l'élimination de ces appels.

Revenons vers notre macro **4-IEME** et modifions-la de manie're à remplacer son appel par le corps de la macro même :

```
(DM 4-IEME (L)
    (RPLACB L '(CADR (CDDR ,(CADR L)))))
```

La fonction **RPLACB** est évidemment celle que nous avons définie à la fin du chapitre précédent : la fonction **DISPLACE** de LE_LISP.

Afin de voir clairement l'effet de cette macro, construisons une fonction **FOO** qui calcule la somme du premier et du quatrie'me élément de sa liste argument. Ce n'est pas difficile :

```
(DE FOO (LISTE)
    (+ (CAR LISTE) (4-IEME LISTE)))
```

Quand nous appelons cette fonction, par exemple par :

```
(FOO '(1 2 3 4 5 6 7))
```

le résultat sera, bien évidemment, la valeur numérique **4**. Mais, si nous regardons maintenant la fonction, elle a l'allure suivante :

```
(DE FOO (LISTE)
    (+ (CAR LISTE) (CADR (CDDR LISTE))))
```

Tous les appels ultérieurs de la fonction **FOO** n'impliquerons plus l'appel de la fonction **4-IEME**, puisque le premier appel a d'abord été remplacé par le résultat de l'évaluation du corps de la macro-fonction.

Des telles macro-fonctions s'appellent des macros *destructives*, puisqu'elles détruisent la forme de leur appel.

Construisons alors la macro-fonction **DMD** qui nous permet de définir de telles macro-fonctions *sans* écrire chaque fois l'appel explicite de **RPLACB**. La voici :

```
(DM DMD (L)
    (LET ((NOM (CADR L))
          (VARIABLE (CADDR L))
          (CORPS (CDR (CDDR L))))
        '(DM ,NOM (,VARIABLE)
             (RPLACB ,VARIABLE ,@CORPS))))
```

La macro-fonction **DMD**, qui est une fonction standard en LE_LISP, construit et évalue un appel de la fonction **DM**. Avec cette fonction, la définition de la fonction **4-IEME** ci-dessus peut s'écrire :

```
(DMD 4-IEME L
     '(CADR (CDDR ,(CADR L))))
```

La fonction **NEXTL** que nous avons définie dans les exercices du chapitre précédent, peut maintenant être définie en tant que macro-fonction comme suit :

```
(DMD NEXTL L
     '(LET ((VAL ,(CADR L)))
        (SETQ ,(CADR L) (CDR VAL)) (CAR VAL)))
```

Avec cette définition de **NEXTL**, la fonction **PREVERSE** du fin du chapitre précédant devient, après une exécution :

```
(DF PREVERSE (L RES)
    (LET ((LISTE (EVAL (CAR L))))
        (IF (NULL LISTE) (SET (CAR L) RES)
            (SETQ RES
                (CONS
                    (LET ((VAL LISTE))
                        (SETQ LISTE (CDR VAL))
                        (CAR VAL))
                RES))
            (SELF LISTE))))
```

L'utilisation des macro-fonctions s'impose chaque fois que vous voulez écrire un programme bien lisible sans pour cela diminuer son efficacité. La première exécution d'une fonction contenant une macro destructive sera un peu plus lente que l'exécution d'une fonction normale. Par contre, toutes les exécutions suivantes seront considérablement accélérées.

16.1. EXERCICES

1. Définissez les macro-fonctions **INCR** et **DECR** qui, similaires à la macro-fonction **NEXTL**, prennent en argument le nom d'une variable, et livrent comme résultat la valeur de cette variable plus ou moins 1. Ces deux macros ont comme effet de bord d'incrémenter ou de décrémenter la valeur de la variable donnée en argument.
2. Définissez la macro-fonction **SETNTH** qui est définie par les exemples suivants :

   (SETNTH '(A B C D) 3 'G) → **(A B G D)**
   (SETNTH '(1 2 3 4) 4 5) → **(1 2 3 5)**

   et si la variable **X** est liée à la valeur **(1 2 3 4)**, après l'appel :

   (SETNTH X 1 'A)

   la valeur de **X** est la liste **(A 2 3 4)**.

chapitre précédent | solutions | chapitre suivant