17. LES DIVERES FORMES DE REPETITION

L'unique manière de faire des répétitions que nous connaissions jusqu'à maintenant est la récursivité, i.e.: l'appel explicite d'une procédure pendant que cette procédure est déjà active. Dans ce chapitre, nous examinerons d'autres manières pour répéter une suite d'instructions et appeler des fonctions.

17.1. APPEL DE FONCTION IMPLICITE (OU LE BRANCHEMENT)

Une première forme d'appel de fonction, propre au langage LISP interprété, est d'utiliser la structure de la représentation du programme comme structure de contrôle. Les programmes LISP étant représentés sous forme de listes, nous pouvons alors utiliser les possibilités des listes pour exprimer des répétitions. Comment ? Simplement en se rappelant qu'une répétition correspond à faire *la même chose* plusieurs fois et qu'une liste peut contenir plusieurs fois *la même chose*, comme nous l'avons vu au chapitre 15.

Prenons comme exemple un programme qui renverse une liste (non circulaire) :[1]

```
(SETQ E '(IF (NULL L) M (SETQ M (CONS (NEXTL L) M)) *))
(RPLACA (LAST E) E)
```

Nous avons ici construit une liste **E** circulaire de la forme :

```
(IF (NULL L) M (SETQ M (CONS (NEXTL L) M))
    (IF (NULL L) M (SETQ M (CONS (NEXTL L) M)) . . . .
```

E peut alors être considéré comme le nom d'un programme, et un exemple d'appel est :

```
(SETQ M () L '(A B C D))     ; pour donner des valeurs aux variables ;
(EVAL E)                     ; lancement de l'exécution du corps ;
```

Ce qui donne la liste **(D C B A)** en résultat et provoque l'affectation de ce résultat à la variable **M**.

La répétition, et donc l'appel de la fonction, est simplement acquis par une lecture répétitive d'une liste circulaire.

Un autre exemple pour exprimer la structure de contrôle dans la structure de la représentation interne des programmes, est la version de **REVERSE** ci-dessous :[2]

```
(SETQ REVERSE
    '(IF (NULL L) ()
         (LET ((X (NEXTL L))) (NCONC1 (EVAL *) X))))
(SETQ * REVERSE)
```

Comme préalablement, l'appel de cette fonction **REVERSE** se fait par une initialisation de la variable **L** et

[1] Cet exemple est duˆ à Daniel Goossens

[2] La fonction **NCONC1** peut être définie par :

```
(DM NCONC1 (L) (RPLACB L '(NCONC ,(CADR L) (LIST ,(CADDR L)))))
```

un appel de **(EVAL REVERSE)**.

La différence avec la fonction **E** consiste dans le fait qu'en **E** nous avons eu une liste circulaire, ici, en **REVERSE**, la circularité est obtenue à travers l'évaluation explicite d'une variable qui pointe vers la structure même : c'est une sorte de circularité indirecte.

Naturellement, sauf dans quelques types de macro-fonctions, il arrive rarement de construire des programmes de cette manière, elle n'est exposée ici qu'à titre de curiosité (et, également, à titre d'exposition de la puissance de la représentation des programmes sous forme de listes !).

17.2. LES ARGUMENTS FONCTIONNELS

Mise à part les deux exemples d'appels de fonctions que nous venons de voir, tous ceux que nous avons rencontrés jusqu'à maintenant, sont des appels où le *nom* de cette fonction apparaît explicitement, comme, par exemple, dans l'appel : **(FOO 1)**. Même les appels de fonctions à travers la fonction **SELF** peuvent être considérés comme des appels explicites, car il ne font rien d'autre qu'appeler récursivement la fonction à l'intérieur de laquelle on se trouve. Autrement dit : dans la plupart des programmes que nous avons vus, on peut déterminer les fonctions appelées par une simple lecture statique.

LISP donne également une possibilité de calculer les fonctions qu'on veut appeler : nous en avons déjà rencontré une manière à travers l'utilisation des fonctions **EVAL** et **APPLY**. Dans le reste de ce chapitre nous examinerons d'autres manières de calculer les fonctions appelantes.

Comme nous l'avons dit au début de ce livre, LISP existe dans une multitude de dialectes. Normalement, les différences d'un dialecte à l'autre sont mineures, sauf en ce qui concerne la manière de traiter l'évaluation des appels de fonctions calculées. Là, nous pouvons distinguer deux types de dialectes LISP : les interprètes LISP s'adaptant aux normes de Common-LISP, dont LE_LISP est un excellent exemple, et les autres, comme, par exemple, VLISP. Dans la suite nous donnons pour chaque programme deux versions : une dans le style de LE_LISP et une dans le style de VLISP.

Dans la définition d'une *forme* au début de ce livre, tout ce que nous avons dit est que l'évaluateur LISP considère le premier élément d'une liste à évaluer comme la fonction, et le reste de la liste comme les arguments de cette fonction. En LE_LISP une erreur *fonction indéfinie* se produit si l'évaluateur, ne connait pas la fonction du début de la liste, i.e.: si elle n'est pas définie. En VLISP l'évaluateur cherche alors à évaluer le **CAR** de cette liste, pour regarder si *la valeur* du **CAR** est une fonction, ainsi de suite, jusqu'à ce que LISP trouve une erreur où rencontre une fonction. Ce sera alors cette fonction qui s'appliquera aux arguments de la liste originale.

Par exemple, si l'on tape :

(SETQ F '1+)

suivi de l'appel :

(F 3)

le résultat de l'évaluation sera **4** : VLISP a évalué l'atome **F**, puisque **F** n'est pas le nom d'une fonction, et a pris le résultat de l'évaluation de **F** comme fonction appliquée à l'argument **3**.

Pour pouvoir faire la même chose en LE_LISP, on doit explicitement appeler la fonction **FUNCALL**, une version spécial de **APPLY**. En LE_LISP, si la variable **F** est liée à **1+**, l'appel

(FUNCALL F 3)

applique la fonction **1+** à l'argument 3, et livre donc également le nombre **4**.

FUNCALL est une fonction standard à un nombre quelconque d'arguments qui peut être définie comme suit :

```
(DE FUNCALL L
    (APPLY (CAR L) (CDR L)))
```

Notons une particularité de VLISP qui permet un nombre arbitraire d'évaluations (donc un niveau d'indirection quelconque) comme le montre l'exemple suivant :

```
(SETQ F '1+)
(SETQ G 'F)
(SETQ H 'G)
(SETQ I 'H)
```

l'appel **(I 3)** donne également la valeur **4** : VLISP évalue successivement les variables **I**, **H**, **G** et **F** jusqu'à ce que l'évaluation livre une fonction (ou une erreur) !

La capacité de calculer la fonction peut être utilisée pour construire des fonctions avec des arguments fonctionnels. Ci-dessous une fonction VLISP qui applique la fonction **F** *deux* fois à son argument **X** :

```
(DE TWICE (F X) (F (F X))
```

En LE_LISP, cette même fonction doit évidemment expliciter les appels de **F** en utilisant **FUNCALL** :

```
(DE TWICE (F X) (FUNCALL F (FUNCALL F X))
```

Voici quelques exemples d'appels :

```
(TWICE '1+ 2)              →  4
(TWICE 'CAR '((A B C)))    →  A
(TWICE 'FACTORIELLE 3)     →  720
```

Cette technique sert à définir quelques fonctions fort utiles : très souvent, nous avons besoin de fonctions qui appliquent une autre fonction aux éléments successifs d'une liste, comme par exemple dans la fonction suivante :

```
(DE PLUS-UN (L)
  (IF (NULL L) ()
      (CONS (1+ (CAR L)) (PLUS-UN (CDR L)))))
```

qui livre une copie de la liste originale avec chacun des éléments incrémentés de 1. Un autre exemple du même schéma de programme est :

```
(DE LIST-EACH (L)
  (IF (NULL L) ()
      (CONS (LIST (CAR L)) (LIST-EACH (CDR L)))))
```

une fonction qui livre une copie de la liste avec chaque élément entouré d'une paire de parenthèses supplémentaires.

Ce type de fonction se présente si souvent, que nous pouvons construire une fonction instanciant ce schéma. La voici, d'abord en VLISP :

```
(DE MAPCAR (L F)
    (AND L
        (CONS (F (CAR L)) (MAPCAR (CDR L) F))))
```

ensuite en LE_LISP :

```
(DE MAPCAR (L F)
    (AND L
        (CONS (FUNCALL F (CAR L)) (MAPCAR (CDR L) F))))
```

A l'aide de cette fonction **MAPCAR**,[3] les deux fonctions ci-dessus peuvent être écrites simplement comme :

```
(DE PLUS-UN (L) (MAPCAR L '1+))

(DE LIST-EACH (L) (MAPCAR L 'LIST))
```

MAPCAR est donc une fonction qui permet l'application d'une fonction aux éléments successifs d'une liste, en ramenant comme résultat la liste des résultats successifs.

Revenons à présent aux divers types d'appel de fonctions, nous en connaissons deux : les appels explicites et les appels par évaluation d'une variable en position fonctionnelle. Evidemment, LISP ne nous limite pas aux atomes en position fonctionnelle : nous pouvons y trouver des expressions arbitraires. Leur évaluation doit livrer un objet fonctionnel. Voici quelques exemples :

VLISP

((CAR '(1+ 1-)) 3)	→	**4**
((LET ((F 'CAR)) F) '(A B C))	→	**A**
((TWICE 'CAR '((FACTORIELLE)(+)(-))) 6)	→	**720**
((LAMBDA (X) (LIST X X)) 2)	→	**(2 2)**

LE_LISP

(FUNCALL (CAR '(1+ 1-)) 3)	→	**4**
(FUNCALL (LET ((F 'CAR)) F) '(A B C))	→	**A**
(FUNCALL (TWICE 'CAR '((FACTORIELLE)(+)(-))) 6)	→	**720**
((LAMBDA (X) (LIST X X)) 2)	→	**(2 2)**

Regardez bien le dernier exemple : il s'écrit de manière identique dans les deux types de LISP. Examinons le : il utilise une λ-expression. Une λ-expression est une fonction sans nom dont l'évaluation livre cette λ-expression elle-même. Il n'est donc pas nécessaire de 'quoter' une λ-expression : elle est auto-quotée.

```
(LAMBDA (X) (LIST X X))
```

est une fonction - sans nom - à un argument, nommé **X**, et avec le corps **(LIST X X)**. Son appel s'effectue simplement en mettant cette λ-expression en position fonctionnelle. C'est très similaire à notre fonction **LET** qui combine la définition d'une λ-expression avec son appel. D'ailleurs, **LET** peut être définie en termes de **LAMBDA** comme suit :

```
(DMD LET L
     '((LAMBDA ,(MAPCAR (CADR L) 'CAR) ,@(CDDR L))
       ,@(MAPCAR (CADR L) 'CADR)))
```

Il est donc possible d'utiliser des fonctions plus complexes que des fonctions standard dans les appels de **MAPCAR**. Ci-dessous une fonction qui livre la liste de toutes les sous-listes après avoir enlevé toutes les occurrences de l'atome **A** :

---

[3] Les arguments de la fonction **MAPCAR** *standard* de LE_LISP sont dans l'ordre inverse : d'abord la fonction, ensuite l'argument.

```
(DE ENL-A (L)
    (MAPCAR L '(LAMBDA (X) (DELQ 'A X))))
```

Si on appelle :

```
(ENL-A '((A B C) (A C B) (B A C) (B C A) (C A B) (C B A)))
```

le résultat est la liste :

```
((B C) (C B) (B C) (B C) (C B) (C B))
```

et si on veut obtenir une liste sans répétitions, il suffit d'écrire les deux fonctions suivantes :

```
(DE ENL-A-SANS-REPETITIONS (L) (ENL-A-AUX (ENL-A L)))

(DE ENL-A-AUX (X)
      (IF (NULL X) ()
          (IF (MEMBER (CAR X) (CDR X))
              (CONS (CAR X) (ENL-A-AUX (DELETE (CAR X) (CDR X))))
              (CONS (CAR X) (ENL-A-AUX (CDR X))))))
```

Ce qui, en utilisant une autre fonction d'itération, **MAPC**, se simplifie en :

```
(DE ENL-A-SANS-REPETITIONS (L)
   (LET ((AUX  ()))
     (MAPC (ENL-A L)
        '(LAMBDA (X)
           (IF (MEMBER X AUX) ()
                (SETQ AUX (CONS X AUX)))))
     AUX))
```

La fonction **MAPC** est une simplification de **MAPCAR**, définie en VLISP, par :

```
(DE MAPC (L F)
   (IF (NULL L) () (F (CAR L)) (MAPC (CDR L) F)))
```

et en LE_LISP :

```
(DE MAPC (L F)
   (IF (NULL L) () (FUNCALL F (CAR L)) (MAPC (CDR L) F)))
```

L'appel :

```
(ENL-A-SANS-REPETITIONS '((A B C) (A C B) (B A C) (B C A) (C A B) (C B A)))
```

livre alors :

```
((C B) (B C))
```

Les fonctions du style **MAP**xxx, qui appliquent une fonction aux éléments successifs d'une liste, ouvrent la voie à tout un ensemble d'algorithmes conceptuellement simples (mais pas obligatoirement des plus efficaces). Ainsi, par exemple, pour trouver toutes les permutations des éléments d'une liste, un algorithme simple consiste à prendre un élément après l'autre de la liste et de le distribuer à toutes les positions possibles à l'intérieur de la liste *sans* cet élément. Le programme en résultant est :

```
(DE PERMUT (L) (PERMUT1 L NIL))

(DE PERMUT1 (L RES)
  (LET ((X NIL))
    (MAPC L
        '(LAMBDA (ELE)
            (SETQ X (DELQ ELE L))
            (IF X (PERMUT1 X (CONS ELE RES))
                (PRINT (CONS ELE RES)))))))
```

Un problème similaire à celui des permutations est le problème des 8 reines. Il s'agit de placer 8 reines sur un échiquer, de manière qu'aucune reine ne puisse prendre une autre.

Rappelons brièvement : un échiquier est un tableau de 64 cases répartis en 8 lignes et 8 colonnes. Une reine est une pièce du jeu dotée de la prérogative de pouvoir prendre toute pièce qui se trouve sur la même ligne, sur la même colonne, ou sur la même diagonale qu'elle.

Il est clair que toute solution comporte exactement une reine par ligne et par colonne. Sachant cela, nous pouvons simplifier la représentation, et au lieu de représenter l'échiquier comme un tableau à deux dimensions, nous allons le représenter comme une liste de 8 éléments, où la place d'un élément représente la colonne (ainsi, le premier élément représente la première colonne, le deuxième élément la deuxième colonne, etc) et où l'élément indique la ligne à l'intérieur de la colonne correspondant, où il faut placer une reine. Par exemple, la liste :

(1 7 5 8 2 4 6 3)

représente la position suivante :

R							
				R			
							R
					R		
		R					
						R	
	R						
			R				

Avec cette représentation, le problème revient à trouver toutes les permutations des nombres 1 à 8, satisfaisant quelques contraintes. Voici alors le programme résolvant le problème des 8 reines :[4]

[4] Une très bonne description de ce problème se trouve dans l'excellent livre *la construction de programmes structurés* de Jaques Arsac.

```
(DE REINE (LISTE) (REINE-AUX LISTE 1 NIL NIL NIL NIL))

(DE REINE-AUX (LISTE COLONNE RESUL DIAG1 DIAG2 AUX3)
   (MAPC L
      (LAMBDA (X)
         (SETQ AUX3 (DELQ X LISTE))
         (IF (AND (NULL (MEMQ (- X COLONNE) DIAG1))
                  (NULL (MEMQ (+ X COLONNE) DIAG2)))
            (IF (NULL AUX3) (PRINT (CONS X RESUL))
               (REINE-AUX AUX3
                     (1+  COLONNE)
                     (CONS X RESUL)
                     (CONS (- X COLONNE) DIAG1)
                     (CONS (+ X COLONNE) DIAG2)))))))
```

Un appel se fera alors par

```
(REINE '(1 2 3 4 5 6 7 8))
```

La fameuse fonction **LIT** est similaire à **MAPCAR**. Là voici d'abord en VLISP :

```
(DE LIT (LISTE FIN FONCTION)
   (IF LISTE
      (FONCTION (NEXTL LISTE) (LIT LISTE FIN FONCTION))
      FIN))
```

et ensuite en LE_LISP :

```
(DE LIT (LISTE FIN FONCTION)
   (IF LISTE
      (FUNCALL FONCTION (NEXTL LISTE) (LIT LISTE FIN FONCTION))
      FIN))
```

Avec cette fonction, l'écriture de quelques-unes des fonctions, étudiées dans ce livre, se simplifie considérablement. La fonction **APPEND**, par exemple, peut être réécrite comme :

```
(DE APPEND (L1 L2) (LIT L1 L2 'CONS))
```

et la définition de la fonction **MAPCAR** devient en VLISP :

```
(DE MAPCAR (L F) (LIT L NIL (LAMBDA (X Y) (CONS (F X) Y))))
```

et en LE_LISP :

```
(DE MAPCAR (L F) (LIT L NIL (LAMBDA (X Y) (CONS (FUNCALL F X) Y))))
```

Finalement, la fonction **MUL** ci-dessous calcule le produit des nombres de la liste donnée en argument :

```
(DE MUL (L) (LIT L 1 '*))
```

17.3. QUELQUES ITERATEURS SUPPLEMENTAIRES

En plus des possibilités de répétition par la récursivité, LISP dispose de quelques *itérateurs* : les fonctions

WHILE, **UNTIL** et **DO**.

La fonction **WHILE** est définie comme suit :

(WHILE *test corps***)** → **NIL**
corps est répétitivement exécuté tant que l'évaluation de *test* donne une valeur différente de **NIL**.

Voici la fonction **REVERSE** écrit en utilisant **WHILE** :

```
(DE REVERSE (L)
   (LET ((RES NIL))
      (WHILE L (SETQ RES (CONS (NEXTL L) RES)))
      RES))
```

et la fonction **APPEND** peut s'écrire :

```
(DE APPEND (L1 L2)
   (LET ((AUX (REVERSE L1)))
      (WHILE AUX (SETQ L2 (CONS (NEXTL AUX) L2))))
   L2)
```

La fonction **UNTIL** est très similaire à **WHILE**, elle est définie comme :

(UNTIL *test corps***)** → **NIL**
corps est répétitivement exécuté jusqu'à ce que l'évaluation de *test* donne une valeur différente de **NIL**.

En utilisant **UNTIL**, notre fonction **REVERSE** devient alors :

```
(DE REVERSE (L)
   (LET ((RES NIL))
      (UNTIL (NULL L) (SETQ RES (CONS (NEXTL L) RES)))
      RES))
```

Naturellement, si la fonction **WHILE** ou **UNTIL** n'existe pas dans votre LISP, vous pouvez les définir comme des macros. Voici, par exemple, une définition de **WHILE** :

```
(DMD WHILE CALL
      '(LET ()
         (IF (NULL ,(CADR CALL)) ()
            ,@(CDDR CALL) (SELF)))))
```

Il est possible d'utiliser à l'intérieur des définitions de macro-fonctions la technique exposée au début du chapitre, i.e.: la répétition par évaluation séquentielle d'une liste circulaire. Ce qui donne pour cette même fonction **WHILE** la nouvelle définition ci-après :

```
(DMD WHILE CALL
      '(IF (NULL ,(CADR CALL)) ()
         ,@(CDDR CALL) ,CALL))
```

MACLISP, un dialecte de LISP devéloppé au M.I.T., a introduit l'itérateur **DO** qui est défini comme suit :

(DO ((var_1 *init repeat*) **. . . (**var_n *init repeat*)**)**
(*test-de-fin valeur-de-sortie***)**
*corps***)**

Cet itérateur se compose donc de trois parties :

1. Une liste de déclaration de variables, ou pour chaque variable on indique - comme dans un **LET** - sa valeur initiale ainsi que - c'est l'originalité de la fonction **DO** - l'expression du calcul de la nouvelle valeur pour chaque itération.
2. Une clause du style **COND**, où le **CAR** est donc un test et le reste la suite des instructions à évaluer si le test donne vrai. Ce test détermine l'arrêt de l'itération.
3. Un corps, indiquant la suite des instructions à évaluer à chaque itération.

Prenons comme exemple la fonction **COMPTE** qui imprime tous les nombres de 0 à **N** :

```
(DE COMPTE (N)
    (DO ((I 0 (1+ I)))
        ((= I N) (TERPRI) 'FINI)
        (PRINT I)))
```

La fonction **REVERSE** s'écrit, en utilisant **DO**, comme ci-dessous :

```
(DE REVERSE (L)
   (DO ((L L (CDR L)) (RES NIL (CONS (CAR L) RES)))
       ((NULL L) RES)))
```

Dans cette forme, la partie 'corps' du **DO** est vide : tout le travail se fait dans l'itération.

La définition de la macro **DO** ci-après utilise la fonction **PROGN**, une fonction standard qui évalue séquentiellement chacun de ses arguments et ramène en valeur le résultat de l'évaluation du dernier argument.

```
(DMD DO CALL
    `((LAMBDA ,(MAPCAR (CADR CALL) 'CAR)
        (IF ,(CAAR (CDDR CALL)) (PROGN ,@(CDAR (CDDR CALL)))
            ,@(CDDDR CALL)
            (SELF ,@(MAPCAR (CADR CALL) 'CADDR))))
      ,@(MAPCAR (CADR CALL) 'CADR)))
```

La fonction **REVERSE** définie en utilisant la macro **DO** se transforme donc, après une exécution, en :

```
(DE REVERSE (L)
   ((LAMBDA (L RES)
       (IF (NULL L)
           (PROGN RES)
           (SELF (CDR L) (CONS (CAR L) RES))))
     L NIL))
```

C'est quasiment la même fonction que celle que nous avons définie dans le chapitre 6.

17.4. EXERCICES

1. Ecrivez la fonction **MAPS**, qui applique une fonction à chaque sous-structure d'une liste. Voici un exemple de son utilisation :

```
? (MAPS '(A (B C) D) 'PRINT)
  (A (B C) D)
```

A
((B C) D)
(B C)
B
(C)
C
(D)
D
= NIL

2. Ecrivez la macro-fonction **UNTIL**, aussi bien en utilisant la fonction **SELF** qu'avec une liste circulaire.
3. Ecrivez la fonction **FACTORIELLE** en utilisant l'itérateur **DO**
4. Ecrivez la macro-fonction **DO** en faisant l'itération avec la fonction **WHILE**.

chapitre précédent

solutions

chapitre suivant