

12. LES FONCTIONS EVAL ET APPLY

En LISP les listes sont à la fois des données à manipuler et des programmes. Ainsi, rien ne s'oppose à la manipulation des programmes comme des listes tout à fait ordinaires. Par exemple, la liste :

(CDR '(HEAD HEAL TEAL TELL TALL TAIL))¹

peut être aussi bien *évaluée*, ce qui ramène la valeur

(HEAL TEAL TELL TALL TAIL)

qu'être argument d'une autre fonction, comme dans :

(CDR (CDR '(HEAD HEAL TEAL TELL TALL TAIL)))

ce qui ramène la valeur

(TEAL TELL TALL TAIL)

voire même, être manipulée comme une liste ordinaire, comme, par exemple, dans :

(CONS 'CDR (CONS '(CDR '(HEAD HEAL TEAL TELL TALL TAIL)) ()))

Ce dernier produisant comme résultat la liste :

(CDR (CDR '(HEAD HEAL TEAL TELL TALL TAIL)))

qui est elle même un autre appel de fonction ! En LISP il n'y a pas de distinction entre la représentation des données et la représentation des programmes. Il est donc possible de construire automatiquement, par programme, d'autres programmes.

Le problème qui se pose alors, est de savoir comment *lancer* ces programmes construits automatiquement, c'est à dire : comment aller de l'expression LISP :

(CDR (CDR '(HEAD HEAL TEAL TELL TALL TAIL)))

à sa valeur :

(TEAL TELL TALL TAIL)

Deux fonctions LISP qui nous permettent de calculer des valeurs LISP à partir d'expressions construites en LISP s'appellent **EVAL** et **APPLY**. Ce sont ces deux fonctions qui seront examinées dans ce chapitre.

¹ C'est une des solutions au problème de Lewis Carroll de savoir comment aller de la tête à la queue.

12.1. LA FONCTION EVAL

La fonction **EVAL** prend un argument et retourne la valeur de l'évaluation de la valeur de cet argument ! C'est à l'air compliqué. Explicitons à travers un exemple. La valeur de l'expression

(CONS 'CDR (CONS '(CDR '(HEAD HEAL TEAL TELL TALL TAIL)) ()))

est la liste

(CDR (CDR '(HEAD HEAL TEAL TELL TALL TAIL)))

Considérant cette liste comme une expression à évaluer, sa valeur est, comme nous l'avons dit ci-dessus,

(TEAL TELL TALL TAIL)

ce qui est également la valeur de l'expression :

(EVAL (CONS 'CDR (CONS '(CDR '(HEAD HEAL TEAL TELL TALL TAIL)) ())))

Ce qui veut dire que la fonction **EVAL** nous permet de *forcer* une évaluation supplémentaire. Prenons encore un exemple. Si nous donnons à LISP l'expression

'(+ (1+ 4) (* 4 4))

LISP nous répond avec la valeur de cette expression, qui, puisque toute l'expression est *quotée*, est l'expression elle-même, c'est-à-dire :

(+ (1+ 4) (* 4 4))

Si maintenant nous donnons à LISP l'expression

(EVAL '(+ (1+ 4) (* 4 4)))

LISP nous répond en ramenant la valeur **21**, la valeur de l'évaluation de cette expression. **EVAL** est donc le contraire de la fonction **QUOTE** : pendant que **QUOTE** empêche l'évaluation de son argument, **EVAL** force cette évaluation.

Voici donc la définition de **EVAL** :²

(EVAL expression) → *valeur (expression)*
EVAL évalue la valeur de son argument.

Prenons encore un exemple. Supposons que nous voulions écrire un petit calculateur simple. Il faut donc pouvoir donner à LISP des expressions de la forme :

(1 + 2 + 3)
(1 + (2 * 3))
((1 + 2) * 3)
(1 * 2 * 3)
...

et, naturellement, le programme doit répondre avec la valeur de ces expressions. Une manière de construire ce calculateur consiste à commencer par traduire les expressions en une forme compréhensible pour LISP, pour les évaluer ensuite. C'est-à-dire : les expressions arithmétiques ci-dessus doivent être traduites en les

² La notation *valeur (x)* se lit comme : calcule la valeur de la valeur de x.

formes ci-dessous :

(+ 1 (+ 2 3))
(+ 1 (* 2 3))
(* (+ 1 2) 3)
(* 1 (* 2 3))
...

Voici une fonction qui traduit les expressions arithmétiques écrites en notation *infixe* en expressions arithmétiques écrites en notation *polonaise prefixée* (sans toutefois prendre en compte les priorités standard des opérateurs), donc en des formes évaluables par LISP :

```
(DE PREFIX (L)
  (IF (ATOM L) L
    (CONS (CADR L)
      (CONS (PREFIX (CAR L))
        (IF (MEMQ (CADDR (CDR L)) '(+ * - /))
          (CONS (PREFIX (CONS (CADDR L)
            (CONS (CADDR (CDR L))
              (CDDR (CDDR L)))))) ()
          (CONS (PREFIX (CADDR L))(CDDR L))))))
```

Voici deux exemples d'appels de cette fonction :

(PREFIX '(1 + 2 + 3)) → (+ 1 (+ 2 3))
(PREFIX '(1 + 2 + (3 * 4) + (80 / 10))) → (+ 1 (+ 2 (+ (* 3 4) (/ 80 10))))

Ce sont bien des *formes*, donc des appels de fonction LISP. Afin de pouvoir livrer un résultat, tout ce qui reste à faire est d'*évaluer* ces formes, ce qui peut être fait avec un simple appel de la fonction **EVAL** . Voici alors la fonction calculatrice :

```
(DE CALCUL (L) (EVAL (PREFIX L)))
```

Si nous appelons cette nouvelle fonction avec les exemples ci-dessus, nous obtenons :

(CALCUL '(1 + 2 + 3)) → 6
(CALCUL '(1 + 2 + (3 * 4) + (80 / 10))) → 23
et finalement
(CALCUL '(2 * 3 * (1 - 5) + 10)) → 36

Notez que l'appel (**EVAL** *x*) engendre *deux* évaluations de *x* :

1. d'abord *x* est évalué parce que cette expression est argument d'une fonction,
2. ensuite, la valeur calculée est évaluée encore une fois à cause de l'appel explicite de **EVAL** .

Afin de rendre cela plus clair, regardez l'expression suivante :

```
(LET ((LE 'THE) (X 'LE)) (PRINT X (EVAL X)))
```

qui imprime

LE THE

et ramène l'atome **THE** en valeur.

D'abord la variable **LE** est liée à l'atome **THE** et la variable **X** à l'atome **LE** . La valeur de l'expression (**EVAL X**) est **THE** , ce qui est la valeur de la variable **LE** , elle-même étant la valeur de la variable **X** .

12.2. LA FONCTION APPLY

L'autre fonction d'évaluation explicite disponible en LISP est la fonction **APPLY**. *Apply* est le mot anglais pour *appliquer* : on applique une fonction à une suite d'arguments.

APPLY est syntaxiquement définie comme :

(**APPLY** fonction liste)

Elle a donc deux arguments qui sont, tous les deux évalués. Le premier argument doit être une expression qui retourne une fonction et le deuxième doit produire une liste dont les éléments seront les arguments de la fonction passée en premier argument. **APPLY** applique alors la fonction à la liste des arguments.

Donnons quelques exemples :

```
(APPLY '+ '(20 80))           → 100
(APPLY 'CAR '((DO RE MI)))    → DO
(APPLY (CAR '(CONS CAR CDR)) (CONS '(DO RE) '(MI FA))) → ((DO RE) MI FA)
```

Tout se passe *comme* si la fonction **APPLY** faisait deux choses :

1. d'abord, elle construit, à partir des deux arguments, un appel de fonction. Par exemple :

(**APPLY** '+ '(20 80)) se transforme en (+ 20 80)

et

(**APPLY** 'CAR '((DO RE MI))) se transforme en (CAR '(DO RE MI))

2. ensuite elle évalue cette expression nouvellement construite.

Il faut avoir à l'esprit que cette fonction :

1. évalue ses deux arguments,
2. attend une *liste* d'arguments : il y a donc toujours un paire de parenthèses supplémentaire autour des arguments, et
3. qu'après avoir évalué les deux arguments elle *applique* la fonction, résultat de l'évaluation du premier argument, à des arguments qui se trouvent dans une liste, qui - elle - est le résultat de l'évaluation du deuxième argument. *Pendant cette application de fonction, chacun des arguments est éventuellement encore une fois évalué !*

Afin de rendre ce dernier point très explicite, regardez l'exemple ci-dessous :

```
(LET ((A 1) (B 2) (C 'A) (D 'B))
  (APPLY '+ (LIST C D))) → 3
```

Voici ce qui se passe : d'abord les variables **A**, **B**, **C** et **D** sont respectivement liées à **1**, **2**, **A** et **B**. Ensuite, **APPLY** évalue ses deux arguments, ce qui donne en position fonctionnelle l'atome + et comme liste d'arguments la liste (**A B**). Finalement, la fonction + est appliquée à la liste (**A B**), ce qui est la même chose que si, au lieu de (**APPLY** '+ (LIST C B)), nous avions écrit (+ **A B**). Etant donné que la fonction + évalue ses arguments, elle calcule la somme des valeurs de **A** et **B**, donc de **1** et **2**, ce qui nous livre le résultat **3**.

La relation entre la fonction **EVAL** et la fonction **APPLY** peut s'exprimer, fonctionnellement, par l'équation

(**APPLY** fonction liste) ≡ (**EVAL** (CONS fonction liste))

La grande utilité de cette fonction est qu'elle résout le problème que nous avons au chapitre précédent, à savoir : comment appeler récursivement une fonction utilisateur de type NEXPR.

Rappelons-nous la fonction que nous avons donnée en exemple :

```
(DE PPLUS (L)
  (IF (NULL L) 0
    (+ (CAR L) (PPLUS (CDR L)))))
```

```
(DE PLUS L (PPLUS L))
```

Nous avons besoin de la fonction auxiliaire **PPLUS** à cause de la liaison de la variable **L** à la *liste* des arguments. Grâce à la fonction **APPLY**, qui fournit les arguments de la fonction passée en premier argument dans une liste, nous pouvons maintenant réécrire cette fonction de la manière suivante :

```
(DE PLUS L
  (IF (NULL L) 0
    (+ (CAR L) (APPLY 'PLUS (CDR L)))))
```

Bien évidemment, nous aurions aussi bien pu écrire :

```
(DE PLUS L
  (IF (NULL L) 0
    (+ (CAR L) (EVAL (CONS 'PLUS (CDR L)))))
```

Mais cette deuxième forme demande l'appel de la fonction **CONS** supplémentaire, et risque donc d'être plus chère en temps d'exécution. Notons toutefois que si nous voulons faire des appels récursifs avec des fonctions de type FEXPR (donc des fonctions utilisateurs définies par **DF**), nous sommes obligés d'écrire l'appel sous cette forme, puisque **APPLY** *n'admet pas* de telles fonctions comme argument fonctionnel.

Le premier argument de **APPLY** peut être une expression LISP quelconque. Il suffit que cette expression retourne une fonction (avec la restriction que nous venons de mentionner). Ainsi, si **P** est définie comme :

```
(DE P (X Y) (+ X Y))
```

toutes les formes ci-dessous sont correctes et calculent la somme de 1 et 2 :

```
(APPLY '+ '(1 2))
(APPLY (CAR '(+ - * /)) '(1 2))
(LET ((X '+)) (APPLY X '(1 2)))
(APPLY 'P '(1 2))
```

12.3. EXERCICES

1. Reprenez chacun des premiers trois exercices du chapitre précédent et transformez les solutions de manière à inclure des appels récursifs utilisant la fonction **EVAL** ou la fonction **APPLY**.
2. Ecrivez un petit traducteur du français vers l'anglais. Pour l'instant ignorez toute question de grammaire : traduisez mot par mot.

Pour faire cela, écrivez pour chaque mot que vous voulez pouvoir traduire une fonction qui ramène le mot traduit. Voici, par exemple, la fonction **CHAT**, qui traduit le mot français 'chat' en son correspondant anglais 'cat' :

```
(DE CHAT () 'CAT)
```

La fonction de traduction devra ensuite utiliser ces fonctions-mots, pour faire des petites traductions

comme :

le chat mange	→	the cat eats
le chat mange le souris	→	the cat eats the mouse
le souris a volé le fromage	→	the mouse has stolen the cheese

Bien évidemment, il faut absolument éviter de définir des fonctions de traduction pour des mots qui correspondent aux noms des fonctions standard, comme, par exemple, *DE*.