I . N . R . I . A .
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay
Tél : 954 90 20

L e — L i s p 6 8 K

L e M a n u e l

Première partie : l'interprète

Jérôme CHAILLOUX

Novembre 1981

AVANT–PROPOS

Le_Lisp 68K [Chailloux 82] est un système LISP qui s'implante sur tout ordinateur possédant une unité centrale de type MC68000 de Motorola. Le_Lisp 68K contient l'interprète, le compilateur et les outils de développement d'un nouveau dialecte du langage Lisp [McCarthy 62], appelé Le_Lisp, fils spirituel de Vlisp [Greussay 77, Chailloux 80] auquel il a emprunté sa concision et sa rapidité dans l'interprétation, fils naturel de Maclisp (et plus précisément des *Post–Maclisp Lisps* tels que le Lisp de la machine Lisp du M.I.T. [Weinreb et Moon 79], Spice Lisp [Steele 81], NIL [White 79] et Franz Lisp [Foderaro 79]) à qui il doit ses performances en tant que langage d'écriture de systèmes sûrs et performants. Le_Lisp a été augmenté de nombreuses facilités, toutes hautement interactives, pour le développement incrémental de systèmes de haut niveau.

Le langage et le système sont destinés au développement et à la réalisation de systèmes de haut niveau axés sur la conception assistée par ordinateur, la simulation, la synthèse d'images colorées, l'informatique musicale et d'une manière générale les problèmes d'intelligence artificielle et les domaines qui y sont rattachés.

Le manuel est composé de 3 volumes décrivant respectivement :

- (I) l'interprète Le_Lisp
- (II) la machine virtuelle LLM3 et le compilateur
- (III) les outils de développement du système Le_Lisp

Ce manuel décrit en principe tous les aspects courants des différents systèmes Le_Lisp 68K, mais n'engage l'auteur que sur les principes de base. Le manuel sera sujet à des remaniements, au fur et à mesure de la création de nouveaux systèmes Le_Lisp 68K et de l'évolution normale et souhaitée des systèmes actuels et de leurs utilisateurs.

Ce manuel n'est pas réellement destiné aux débutants, mais aux utilisateurs ayant déjà une certaine expérience du langage LISP. Le lecteur ne doit donc pas espérer trouver une introduction progressive et ordonnée aux constructions du langage. Toutefois la très grande quantité d'exemples de tout niveau doit permettre à ce manuel d'être tout à la fois un manuel de référence et un manuel d'utilisation. Nous ne saurions que trop conseiller aux tout débutants le livre de C. Queinnec intitulé *LISP, langage d'un autre type,* édité chez Eyrolles [Queinnec 82] ou bien celui de H. Winston et B. Horn, intitulé *LISP* [Winston et Horn 81] (bien que ce dernier ait légèrement tendance à prendre Lisp pour Fortran).

Si vous détectez une situation très anomalique, une erreur manifeste d'un des systèmes, une omission dans le manuel ou si vous désirez parler de choses et d'autres, contactez–moi à l'adresse ci–dessous :

Jérôme CHAILLOUX
I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay, Cédex

tél : 954–90–20 poste : 456

ou par courrier électronique :

Chailloux.Vlsi at INRIA

ou bien

Chailloux@SRI–CSL
Chailloux.INRIA@mit–multics

Je crois à propos de remercier :

G. BAUDET et J.J. LEVY qui ont réalisés l'interface avec le système VERSAdos de l'EXORmacs, la connection de l'EXORmacs au système Multics (à partir duquel a été amorcé le système) et les routines de COLORIX et de la tablette graphique;

J.M. HULLOT qui a conçu et implanté les RECORD, LUCIFER et a participé à l'implantation de l'éditeur EMACS.

G. BERRY, P. COINTE, C. COSLADO, G. HUET, E. NEIDL, E. StJAMES, qui ont participés à l'élaboration du nouveau dialecte Le_Lisp;

J. VUILLEMIN et G. KAHN pour leur appui efficace et amical tout au long du projet qui, du moins à ses débuts, était hautement spéculatif;

G. NOWAK, P. COINTE et J.J. LEVY qui ont relu la version préliminaire de ce manuel;

ainsi que tous les autres utilisateurs qui ont influencé, par leurs excellentes suggestions, la réalisation et l'implantation du système Le_Lisp 68K.

La composition de ce manuel a été réalisée à l'INRIA avec le programme COMPOSE disponible sur le système Multics, l'impression finale du document a été réalisée sur photocomposeuse VIP grâce à la compétence de G. HUET et G. COUSINEAU.

BIBLIOGRAPHIE

[Chailloux 80]
Le modèle Vlisp : description, évaluation et interprétation, Thèse de 3ème cycle, Université de Paris VI, Avril 1980.

[Chailloux 82]
Le_Lisp : transports et performances, (à paraître)

[Cointe 81]
Fermetures dans les lambda–interprètes. Application aux langages LISP, PLASMA et SMALLTALK, thèse de 3ème cycle, Université de Paris VI, 1982.

[Foderaro 79]
Franz Lisp Manual, Univ. of California, Berkeley, Ca., 1979.

[Greussay 77]
Contribution à la définition interprétative et à l'implémentation des lambda–langages, thèse, Université de Paris VI, Novembre 1977.

[McCarthy 62]
LISP 1.5 Programmer's manual, the M.I.T. Press, Cambridge, Mass., 1962.

[Queinnec 82]
LISP : langage d'un autre type, Eyrolles, Paris, 1982.

[Steele 81]
Spice Lisp Reference Manual, Spice Document S061, Carnegie Mellon University, July 1981.

[Teitelman 78]
Interlisp Reference Manual, XEROX PARC, October 1978.

[Weinreb et Moon 79]
Lisp Machine Manual, Artificial Intelligence Laboratory, M.I.T., Cambridge, Mass., 79.

[Winston 81]
LISP, Addison Wesley, 1981.

[White 79]
NIL – a perspective, Proc. of the Macsyma User's conf., Washington D.C., June 1979.

PLAN DE LECTURE

Ce premier volume du manuel est exclusivement réservé à la description du fonctionnement de l'interprète et des fonctions qui lui sont rattachées. Le_Lisp 68K est un système de taille respectable, de l'ordre de 1000 symboles y sont pré–définis, leur description ne pourra se faire que par grandes familles. Quelques 400 symboles vont être abordés dans ce pr'sent volume.

Le chapitre 1 contient la mise en œuvre des systèmes. Sa lecture est indispensable pour le premier contact avec un des différents systèmes ou pour l'adapter à une utilisation particulière.

Le chapitre 2 traîte de la représentation des données et du fonctionnement interne de l'interprète. Ce chapitre, notoirement opaque, peut être sauté en première lecture par les utilisateurs ayant déjà une idée de l'interprétation de LISP en général. Ne s'y référer que pour les détails d'implantation et d'interprétation.

Le chapitre 3 contient la description des fonctions standards du système. A lire dans le sens de la marche à ses moments perdus, y accèder au moyen de la table d'index finale en utilisation courante.

Le chapitre 4 traîte des fonctions spécialisées dans les entrées/sorties simples, la manipulation des fichiers et la gestion des périphériques spécialisés.

Le chapitre 5 traîte des fonctions systèmes. Sa lecture est reservée à ceux qui désirent réaliser des sous–systèmes spécialisés.

Enfin, le plus important dans un manuel de ce type, deux tables d'index qui permettent un accès direct aux concepts et aux fonctions.

CHAPITRE 1

UTILISATION

1.1 Les différentes versions du système

Le_Lisp 68K est un système Lisp destiné à fonctionner sur toute machine à base d'unité centrale de type MC68000 de Motorola. Il est de fait portable très rapidement sur toute machine de ce type, dotée ou non d'un système d'opération, car la majeure partie du système est écrite dans le langage de la machine virtuelle LLM3 ou bien en Lisp lui-même ce qui facilite l'amorçage et le transport des systèmes les uns les autres. Le premier transport a été réalisé en utilisant la machine Multics de l'INRIA et un EXORmacs.

Actuellement (Novembre 1981) seul le système sur machine EXORmacs et système d'exploitation VERSAdos est totalement opérationnel. Seront opérationnels été 82 (si les délais annoncés par les fabricants sont respectés) les systèmes fonctionnant sur :

- la machine du CNET, SM90, sous système UNIX
- une machine nue à base de plaques VERSAmodule, fabriquée par Louis Audoire.
- la machine MicroMega

De nombreux contacts ont lieu actuellement avec les constructeurs d'ordinateurs personnels à base de 68000. En particulier il est envisagé de transporter le système sur les machines suivantes :

- le TRS Model II, équipé d'une carte 68000.
- la machine NU sous système TRIX au M.I.T.
- la machine SUN à l'Université de Stanford
- la machine APOLLO à l'Université de Brown

1.2 Pour débuter avec Le_Lisp 68K sous VERSAdos

Pour jouer Le_Lisp 68K, il suffit, sous le système VERSAdos, d'émettre sur le terminal la commande :

```
=lelisp
```

dans tous les cas, le système répond :

```
******* Le_Lisp 68 K (by INRIA) version 1.3 jj/mm/aa
= Systeme standard.
```

dans lequel jj/mm/aa est la date de la dernière modification de l'interprète. Le système rentre alors dans la boucle principale de l'interprète qui va, indéfiniment, lire une expression sur le terminal, l'évaluer puis imprimer la valeur de cette évaluation. Le système indique qu'il attend la lecture d'une expression en imprimant, sur le terminal, le caractère ? au début de chaque ligne. La valeur de l'évaluation est imprimée précédée du caractère =.

Voici un exemple de session réalisée avec Le_Lisp 68K :

```
=lelisp

******* Le_Lisp 68K (by INRIA) version 1.3 31-Nov-81
= Systeme standard du 31-Nov-81

? ()
= NIL

? (length (oblist))
= 942

? (de fib (n)
?    (cond ((= n 0) 1)
?          ((= n 1) 1)
?          (t (+ (fib (1- n)) (fib (- n 2))))))
= fib

? (fib 20)
= 6567          ; en moins de 12 secondes

? (end)
  Que Le_Lisp soit avec vous
```

1.3 Quelques trucs à savoir

Enfin voici quelques trucs en vrac, ils sont développés longuement dans les chapitres suivants, à savoir pour utiliser le système confortablement dès le début :

- pour effacer un caractère, utilisez la touche DELETE ou BACKSPACE le caractère à gauche du curseur doit s'effacer de l'écran.

- pour détruire une ligne, utilisez le caractère ^X (émis en appuyant simultanément sur les touches CTRL ou CONTROL et X). Un caractère | est imprimé, un saut de ligne est effectué et le système se remet à l'écoute d'une nouvelle ligne.

- pour revenir à la boucle principale de l'interprète alors que vous êtes en train de taper une expression, provoquez une erreur de lecture en tappant deux points les uns à la suite des autres.

- pour revenir à la boucle principale de l'interprète dans les cas désespérés appuyez sur la touche BREAK ou ATTN et repondez par un point si une question vous est posée.

- pour charger un fichier précédemment créé, dîtes simplement :

```
? ■ L fichier
```

i.e. le caractère ^L (appui sur la touche CNTRL et L simultanément) suivi du nom du fichier.

- pour créer ou éditer un fichier, appelez l'éditeur Emacs en évaluant :

```
? (emacs)
```

cet Emacs est compatible au niveau des clés et de la sémantique des fonctions attachées aux clés avec Emacs Multics. Les fonctions qui n'existent que sur Multics déclenchent la sonnette du terminal, et les fonctions qui n'existent que sur 68000 sont décrites dans le troisième volume du manuel. Pour obtenir un fichier récapitulatif des commandes de Emacs, utilisez la commande étendue :

```
esc X print-wall-chart
```

- pour insérer un commentaire, tapez un point-virgule; tout le reste de la ligne est ignoré jusqu'à la fin complète de la ligne.

- pour augmenter la taille mémoire allouée aux cellules de listes il suffit d'appeler Le_Lisp au moyen de la commande :

```
=lelisp In>
```

dans laquelle <n> est la taille de la zone liste exprimée en *8k cells*. Par défaut n=2 (i.e. il y a 16k cellules de liste). La limite théorique est de 128 (i.e. 1024 k cellules de liste (i.e. 1 M cellules de liste)) la limite pratique est la taille physique de la mémoire. A titre indicatif, une unité de *8k cells* correspond à toute la mémoire adressable d'un Z80 ou d'un PDP11.

- pour pouvoir charger automatiquement au départ de l'interprète un fichier quelconque il suffit d'appeler Le_Lisp au moyen de la commande :

```
=lelisp Ifichier>
```

dans laquelle <fichier> est un nom de fichier quelconque contenant du code Lisp qui sera entièrement chargé et exécuté avant d'entrer dans la boucle d'interaction sur le terminal. Bien évidemment les deux types de commandes peuvent se mélanger et il est possible

d'appeler Le_Lisp en spécifiant à la fois une taille et un fichier. Ex :

```
=lelisp 5 llib.init.ll
```

appelle le système Lisp en prévoyant une mémoire de 40k cellules de listes et en chargeant automatiquement le fichier LLIB.INIT.LL

- pour utiliser les fonctions de COLORY, il faut appeler Le_Lisp avec l'argument -MI. Cet argument indique au système d'allouer une zone supplémentaire de 64k octets pour y stocker la mémoire d'image de COLORY.

```
=lelisp 6 -mi
```

appelle le système Lisp en prévoyant une mémoire de 48k cellules de listes et 64k octets pour la mémoire image de COLORIX.

CHAPITRE 2

L'INTERPRèTE

2.1 Les Objets de base

L'interprète Le_Lisp 68K permet de manipuler des objets nommés S-expressions (pour Symbolic expressions).

Ces objets sont classés en 5 types distincts :

- les symboles
- les nombres
- les chaînes de caractères
- les listes
- les tableaux

Toute S-expression est représentée en machine au moyen d'un pointeur (ou d'une adresse). L'accès aux valeurs des différents objets sera donc toujours effectué au moyen d'une indirection. L'interprète Le_Lisp sera donc spécialisé dans la manipulation de pointeurs.

Le_Lisp 68K fonctionne avec une Unité Centrale MC68000 qui possède des pointeurs de 32 bits permettant d'adresser 4 Giga objets (2^32). Du fait d'une limitation hardware dûe au nombre de pattes du boitier, seuls 24 bits sont effectivement utilisés actuellement. Il en résulte que seuls 16 Mega objets sont adressables (ce qui correspond à 2 Mega cellules de liste). Dans tous les cas, l'espace adresse fourni par l'Unité Centrale est très confortable. A titre de comparaison, le PDP10, cheval de bataille des systèmes Lisp des quinze dernières années, ne pouvait adresser au maximum que 256K cellules de listes. Très récemment, Motorola a annoncé une nouvelle version de l'Unité Centrale 68000 qui donne accès à un espace adresse de 32 bits.

2.1.1 Les symboles

Ils jouent le rôle d'identificateurs et servent à nommer les variables et les fonctions. Ils sont créés implicitement dès leur lecture dans le flux d'entrée ou explicitement par les fonctions IMPLODE ou GENSYM ; nul besoin donc de les déclarer.

Leur nom externe (Print name ou P-NAME) est une suite de caractères quelconques (contenant au moins un caractère non numérique) de longeur limitée à 62 caractères. On peut insérer dans un P-NAME des caractères délimiteurs s'ils sont précédés du caractère / (slash) ou mieux, si le P-NAME contient des caractères spéciaux, on peut encadrer tout le P-NAME avec le caractère | *valeur absolue* (voir la section sur la lecture standard).

Un symbole est représenté dans l'interprète par un pointeur sur un descriptif stocké dans une zone spéciale de la mémoire.
Ce descriptif est constitué des 7 propriétés naturelles suivantes :

C-VAL (abréviation de Cell-VALue) qui contient à tout moment la valeur associée au symbole considéré comme une variable. L'accès à cette valeur est extrêmement rapide. A la création d'un symbole, sa C-val est *indéfinie*. Toute tentative de consultation d'un symbole qui n'a pas encore reçu de valeur provoque irrémédiablement une erreur.

P-LIST (abréviation de Properties LIST) qui contient à tout moment la liste des propriétés du symbole. Ces propriétés sont gérées par l'utilisateur au moyen des fonctions spéciales sur les P-LIST. Le système n'utilise jamais les P-LIST des symboles pour ses besoins propres.

F-VAL (abréviation de Function-VALue) qui contient à tout moment la valeur associée au symbole considéré comme une fonction. Cette valeur peut-être :
- une adresse machine dans le cas des SUBR
- une S-expression dans le cas des EXPR

L'accès à la F-VAL est réalisé au moyen de la fonction GETFN et GETDEF.

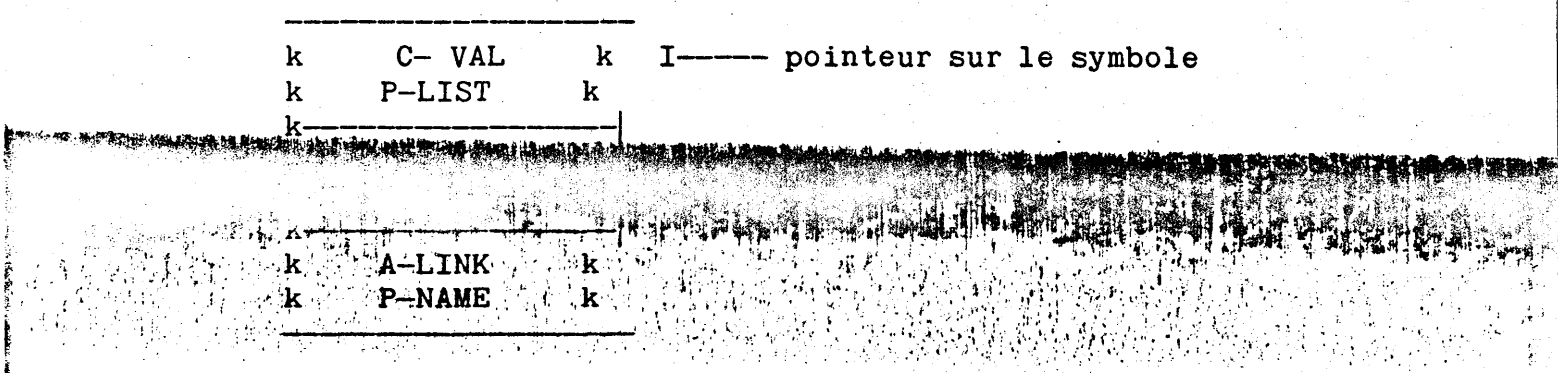
F-TYPE (abréviation de Function TYPE) qui contient le type (codé) de la fonction stockée dans la F-VAL. L'ensemble F-VAL, F-TYPE permet à l'interprète de lancer les fonctions d'un manière extrêmement rapide. La consultation, en clair, du F-TYPE des symboles est réalisée au moyen de la fonction TYPEFN.

P-TYPE (abréviation de Print TYPE) qui contient les informations nécessaires à l'édition de la représentation externe du symbole
- comme variable : en indiquant la restitution du caractère *valeur absolue* pour encadrer les caractères du P-NAME.
- comme fonction : pour permettre au PRETTY-PRINT de connaître le format d'édition à utiliser. L'accès au P-TYPE d'un symbole est effectué au moyen de la fonction spéciale PTYPE.

A-LINK (abréviation de Atom-LINK) qui contient l'adresse de l'atome suivant dans la table des symboles. Ce lien permet entre autre de gérer facilement le hachage (hash-coding) de la table des symboles. Cet attribut n'est pas accessible à l'utilisateur directement.

P-NAME (abréviation de Print-NAME) qui contient l'adresse de la chaîne de caractères représentant le nom du symbole.

Ces propriétés naturelles sont rangées en mémoire suivant le schéma :



Certains symboles sont déjà connus à l'appel du système :
- les constantes littérales (qui contiennent leur propre adresse en C-VAL) dont voici la liste : NIL T LAMBDA LAMBDA-NAMED SUBR NSUBR FSUBR EXPR FEXPR MACRO QUOTE.
- les fonctions standards

2.1.2 Les nombres

Le_Lisp manipule des nombres entiers sur 16 bits (permettant de calculer dans l'intervalle : [-2^15 ,, +2^15-1] et des nombres flottants sur 32 bits. Dans le système EXORmacs, toutes les routines traitant des nombres flottants proviennent de la bibliothèque Pascal PASCALIB

Le P-NAME d'un nombre est la représentation de sa valeur dans la base de conversion courante (e.g. 10).
La valeur d'un nombre est ce nombre lui-même. Un nombre n'a ni C-VAL ni P-LIST.

2.1.3 Les chaînes de caractères

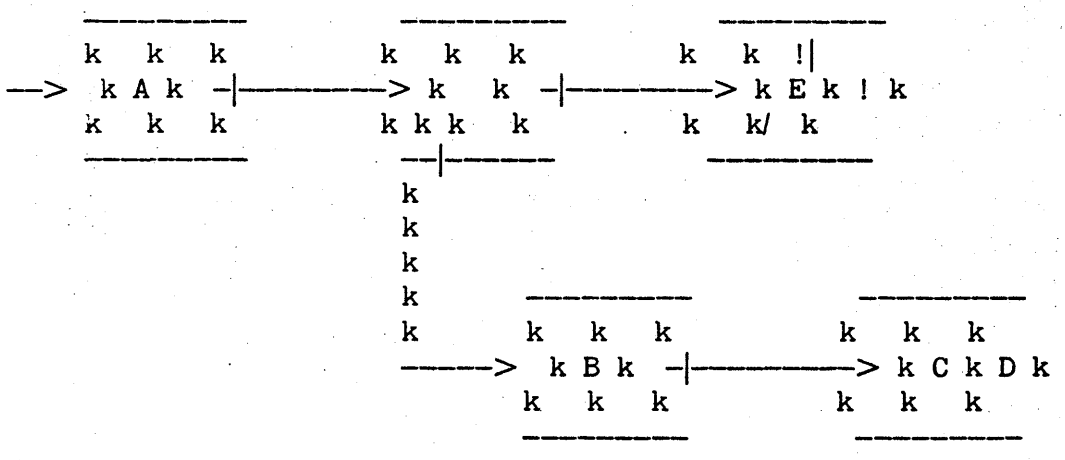
Le_Lisp possède des chaînes de caractères dont la représentation externe est la suite des caractères composant cette chaîne encadrée du caractère guillemet. Si le caractère guillemet doit apparaître dans une chaîne, il suffit de le doubler. Une chaîne de caractères ne peut pas dépasser 256 caractères actuellement.
La valeur d'une chaîne est cette chaîne elle-même, nul besoin de les *quoter* donc. Tout comme un nombre, une chaîne n'a ni C-VAL ni P-LIST.

```
ex :  "Foo Bar"     correspond a la chaine    Foo Bar
      """abc"""                                "abc"
      """""                                    "
```

2.1.4 Les listes

Les listes sont représentées d'une manière standard : les différents éléments d'une liste sont stockés dans une cellule de liste (constituée d'un doublet de pointeurs) dont la partie gauche (le CAR) contient l'élément, et la partie droite (le CDR) contient un pointeur sur l'élément suivant ou NIL pour le dernier élément de la liste.

Ex : la liste (A (B C . D) E) est stockée en mémoire sous la forme :



Une implantation extrêmement efficace (par trap hardware) des listes dynamiques est en cours de développement. Ces listes dynamiques réalisent un appel à l'interprète à chaque tentative d'accès à leurs différents composants au moyen des fonctions primitives internes CAR, CDR, RPLACA et RPLACD. L'appel de l'interprète pourra (et c'est là tout le jeu) modifier physiquement la liste dynamique elle-même.

2.1.5 Les tableaux

Le_Lisp permet de gérer des tableaux de deux types :
- les tableaux de pointeurs Lisp
- les tableaux de nombres entiers (sur 16 bits) correspondant à des mots machine. Ces tableaux permettent d'adresser toute la mémoire et sont donc utilisés par le chargeur LLM3 pour ranger en mémoire le code engendré par le compilateur.

Le nombre de dimensions ainsi que les bornes des indices sont variables et contrôlés durant l'exécution des programmes.

2.2 Fonctionnement de base de l'interprète

2.2.1 Evaluation d'un atome

La valeur d'un symbole (considéré comme une variable) est sa C-VAL. L'évaluation d'un symbole dont la C-VAL est indéfinie (c'est à dire auquel on a pas encore donné de valeur) provoque lors de son évaluation une erreur dont le libellé est :

```
** Ifnt> : variable indefinie : Isym>     ou bien
** <fnt> : undefined variable : <sym>
```

dans lequel le nom du symbole incriminé <sym> est imprimé ainsi que le nom de la fonction <fnt> qui a provoqué l'erreur.

Il existe de fait trois utilisations des variables en Lisp :

- comme variable globale. Ces variables ont une valeur toujours accessible par toutes les fonctions. Il est recommandé de les initialiser au moyen de la fonction DEFVAR (voir le chapitre suivant).

- comme variable locale. Danc ce cas les variables reçoivent une valeur qu'elles ne gardent que le temps de l'exécution d'une fonction (ce sont les paramètres formels des fonctions).

- comme variable rémanente *(own variable)*. Dans ce cas ces variables ne sont globales que le temps de l'évaluation d'une fonction donnée. Ces variables doivent être clôturées avec une fonction au moyen de la fonction CLOSURE.

La valeur d'un nombre ou d'une chaîne de caractères est ce nombre ou cette chaîne de caractères, nul besoin donc de les *quoter*.

2.2.2 Evaluation d'une liste

L'évaluateur considère toujours une liste comme un appel de fonction. Cette liste s'appelle une forme. Le CAR de cette forme est la fonction, le CDR de la forme les arguments de la fonction.
La valeur d'une forme est la valeur retournée par l'application de la fonction aux arguments.

2.3 Les fonctions

Une fonction (le CAR d'une forme) peut être n'importe quelle S-expression. Le CDR de la forme est la liste des paramètres actuels de la fonction.

Si la fonction est un symbole, la fonction à utiliser est celle qui a été associée à ce symbole
- soit à l'initialisation du système (c'est le cas des fonctions prédéfinies qui sont appelées également fonctions standards)
- soit par l'utilisateur au moyen des fonctions de définition statiques ou dynamiques.

Si aucune fonction n'a été associée à cet atome, une indirection est opérée sur la C-VAL (i.e. sur la valeur du symbole en tant que variable). Si cette C-VAL est indéfinie ou contient une constante littérale une erreur apparait dont le libellé est :

```
** Ifnt> : fonction indefinie : Isym>     ou bien
** <fnt> : undefined function : <sym>
```

dans lequel le nom du symbole incriminé <sym> est imprimé ainsi que le nom de la fonction <fnt> ayant provoqué l'erreur.

```
ex : (CONS 'A 'B)       -> (A . B)
     (SETQ KONS 'CONS)  -> CONS
     (KONS 1 2)         -> (1 . 2)
```

Dans le cas où la fonction est un nombre ou une chaîne de caractères, l'interprète provoque également une erreur dont le libellé est le même que dans le cas précédent.

```
ex : (3 '(1 2 3))  ->
     ** E VAL : fonction indefinie : 3
```

Dans le cas où la fonction est une liste, il peut s'agir :

- d'une déclaration explicite de fonction. Dans ce cas le premier élément de la liste doit être l'atome spécial LAMBDA (pour une fonction anonyme) ou LAMBDA-NAMED (pour une fonction avec nom).

```
ex : ((LAMBDA (x) (+ x x 1)) 5)   -> 11
     ((LAMBDA (x) x) (LAMBDA (x) x))  -> ???
     ; quelle peut etre la valeur de cette evaluation ?
```

- d'une nouvelle forme qui doit être évaluée et dont la valeur est la fonction à utiliser. On a donc affaire dans ce cas non pas à des fonctions constantes (anonymes ou non) mais à des fonctions variables. Lisp est l'un des rares langages dans lequel il est possible d'écrire des appels très agréables du genre :

```
((IF (< n 0) '* '+) val 2)
```

Le_Lisp possède 5 familles de fonctions qui vont être traitées différemment par l'interprète :

- les SUBR et les FSUBR écrites en langage machine.
- les EXPR, les FEXPR et les MACRO écrites en Lisp.

2.3.1 Les SUBR

Les SUBR sont des fonctions écrites en langage machine LLM3, résidentes dans l'interprète dès son lancement (les fonctions standards) ou après compilation (les fonctions compilées). L'exécution de ces fonctions est extrêmement rapide. Ces fonctions possèdent des arguments évalués. Il existe des SUBR à nombre fixe d'arguments (en général ce nombre est plus petit que 6) et des SUBR à nombre variable d'arguments (ces dernières sont parfois appelées NSUBR). Pour les SUBR à nombre fixe d'arguments l'interprète teste si le bon nombre d'arguments est fourni à l'appel. S'il ne l'est pas une erreur apparait dont le libellé est :

```
** Ifnt> : mauvais nombre d'arguments : Is>     ou bien
** <fnt> : wrong number of arguments : <s>
```

dans lequel le nom de la SUBR incriminée <fnt> est imprimé ainsi que le reste des arguments <s> s'il y en a de trop ou NIL s'il en manque.

Pour les NSUBR l'interprète teste parfois le nombre minimum d'arguments à l'appel.

Il est possible de rajouter des fonctions de ce type, en les écrivant directement en langage machine LLM3 ou en faisant compiler des EXPR.
Ces fonctions sont très nombreuses dans l'interprète standard, entre 400 et 500 en fonction du système utilisé.

```
Ex : (CONS 'A)    =>  ** CONS : mauvais nombre d'arguments : NIL
     (CONS 1 2 3) =>  ** CONS : mauvais nombre d'arguments : (3)
     (CONS 1 2)   =>  (1 . 2)
```

2.3.2 Les FSUBR

Les FSUBR sont également des fonctions écrites en langage machine (donc exécutées extrêmement rapidement), résidentes dans l'interprète dès son lancement. Ces fonctions possèdent des arguments en nombre variable qui ne sont pas évalués. Ces fonctions, très spéciales, sont principalement utilisées comme fonctions de contrôle de l'interprète ou comme fonctions de manipulation de noms. Elles sont peu nombreuses.
Tout comme pour les SUBR il n'est pas possible de définir des fonctions de ce type, sauf à les écrire directement en langage machine LLM3 ou en faisant compiler des FEXPR.

2.3.3 Les EXPR

Les EXPR sont des fonctions écrites en Lisp et interprétées par les fonctions standards d'évaluation (EVAL et APPLY). Ces fonctions possèdent un nombre quelconque de paramètres formels, représentés par une liste de variables <lvar> ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions <s1> ... <sN>.

On décrit une fonction de ce type en utilisant une liste, appelée LAMBDA-expression, de la forme :

(LAMBDA <lvar> <s1> ... <sN>)

dans laquelle le symbole LAMBDA est un indicateur de fonction, <lvar> est la liste des paramètres formels et <s1> ... <sN> le corps de la fonction.

```
ex : (LAMBDA (x y) (CONS (CAR x) (CDR y)))
```

Pour décrire des fonctions récursives, il est nécessaire de pouvoir donner un nom à une LAMBDA-expression. On utilise dans ce cas une liste de la forme :

(LAMBDA-NAMED <sym> <lvar> <s1> ... <sN>)

dans laquelle <sym> est le nom attaché à la lambda-expression, <lvar> et <s1> ... <sN> sont identiques à la description précédente.

```
ex : (LAMBDA-NAMED ELTLAST (l)
          (IF (NULL (CDR l))
              (CAR l)
              (ELTLAST (CDR l))))
```

La définition des fonctions de type EXPR (i.e. l'association d'une LAMBDA-expression à un symbole) est décrite dans la section suivante.

L'évaluation d'un appel de fonction de type EXPR s'opère en 3 étapes :

1 - liaison des paramètres actuels aux paramètres formels. Cette liaison est réalisée en Lisp après avoir sauver les anciennes valeurs des paramètres formels dans la pile.
2 - évaluation des différentes expressions du corps <s1> ... <sN>. La valeur retournée par l'appel de la fonction sera la valeur de la dernière évaluation (i.e. celle de <sN>)
3 - destruction des liaisons effectuées en 1 - et restitution des anciennes valeurs sauvées dans la pile.

Le type de liaison des paramètres est fonction de la forme de la liste des paramètres formels <lvar>.

- Si <lvar> est une liste de variables, la liaison s'effectue élément par élément entre la liste des variables (paramètres formels) et la liste des valeurs (paramètres actuels évalués). Si la liste des paramètres actuels est plus longue que celle des paramètres formels, il se produit une erreur. Si la liste des paramètres formels est plus longue que celle des paramètres actuels, les variables restantes sont liées à la valeur NIL par défaut.

- Si <lvar> est une variable simple, tous les paramètres actuels sont évalués puis rassemblés en une liste qui est liée à la variable. On appelle souvent ce type de fonction des LEXPR (terminologie Maclispienne).

- Si <lvar> est une liste pointée de variables, la liaison s'effectue suivant les 2 modes précédents (voir le dernier exemple).

exemples de liaison des EXPR

```
((LAMBDA (X Y Z) (LIST X Y Z))
   (PRIN 1) (PRIN 2) (PRIN 3) (PRIN 4))
123   et  ->  (1 2 3)

((LAMBDA (X Y Z) (LIST X Y Z))
   (PRIN 1))
1  et  ->  (1 NIL NIL)

((LAMBDA X X) (PRIN 1) (PRIN 2) (PRIN 3))
123  et  ->  (1 2 3)

((LAMBDA (X Y . Z) (LIST X Y Z))
   (PRIN 1) (PRIN 2) (PRIN 3) (PRIN 4))
1234  et  ->  (1 2 (3 4))
```

Si la liste des arguments est mal-formée, une erreur apparait dont le libellé est :

```
 ** Ifnt> : mauvaise liste d'arguments : Is>     ou bien
** <fnt> : wrong list of arguments : <s>
```

dans lequel <fnt> est le nom de la fonction ayant déclenchée l'erreur (EVAL ou APPLY) et <s> la liste d'arguments défectueuse.

```
ex : ((LAMBDA (x y) (+ (* x x) y)) '(1 . 2))  ->
     ** E VAL : mauvaise liste d'arguments : 2
```

Compatibilité : Maclisp ne possède pas cette forme généralisée de LEXPR mais une manipulation extrêmement fastidieuse des arguments variables au moyen de leur numéro. Vlisp, quant à lui, ne teste pas du tout le nombre d'arguments à l'appel d'une fonction et ne possède pas les LAMBDA-NAMED. Il permet toutefois la description de fonctions explicites récursives à un niveau au moyen de la fonction SELF qui applique la dernière fonction appelée dynamiquement.

2.3.4 Les FEXPR

Les FEXPR sont des fonctions écrites en Lisp et interprétées par les fonctions standards d'évaluation (EVAL et APPLY). Tout comme les EXPR ces fonctions possèdent un nombre quelconque de paramètres formels, qui sont rangés dans une liste <lvar> ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions <s1> ... <sN>.

Ces fonctions ne diffèrent des EXPR que par le mode de liaison des paramètres. Tous les paramètres actuels (i.e. la CDR de la forme elle-même) sont rassemblés sans être évalués en une liste qui est liée à la première (et seule) variable de la liste des paramètres formels. Toutes les autres variables de cette liste font office de variables locales et sont liées à la valeur NIL.

2.3.5 Les MACRO

L'interprète reconnait un autre type de fonctions, les MACRO. Tout comme les EXPR et les FEXPR, ces fonctions écrites en Lisp possèdent un nombre quelconque de paramètres formels, qui sont rangés dans une liste <lvar> ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions <s1> ... <sN>.

Pour évaluer une forme dont la fonction est une MACRO, l'interprète évalue tout d'abord la fonction associée à cette MACRO avec toute la forme comme paramètre actuel puis re-évalue la valeur retournée de cette première évaluation. L'évaluation d'une macro se fait donc en deux temps.
C'est l'appel de la macro tout entier qui est passé en argument et qui est lié à la première variable de la liste des paramètres formels (les autres variables servant de variables locales au corps de la MACRO), il est donc possible de réaliser des modifications physiques de la forme elle-même à la première évaluation de la macro (voir les exemples de la section suivante).

2.3.6 Les Fermetures

Le_Lisp permet de définir des fermetures [Cointe 1981] (CLOSURE en anglais). Une fermeture est un couple constitué d'une fonction et d'un certain nombre de variables rémanentes. Elles sont définies au moyen des fonctions CLOSURE et DEFCLOSURE.

```
; voici un generateur de la suite des nombres de Fibonacci

(setq fib (let ((x 1) (y 1))
            (closure '(x y)
                     '(lambda () (psetq y (+ x y) x y)
                                  y))))

? (fib)
= 2
? (fib)
= 3
? (fib)
= 5
? x
= ** EVAL : variable indefinie : X
```

2.4 Définition des fonctions

Il existe deux types d'utilisation des fonctions :

- une utilisation statique : les fonctions sont définies d'une manière globale et gardent leurs définitions tant qu'on ne les modifient pas explicitement. Ce type de définition ne permet pas de conserver les définitions antérieures (voir les fonctions DE/DF/DM)

- une utilisation dynamique : les fonctions peuvent changer de définition durant certaines évaluations et retrouver par la suite leurs anciennes définitions (voir les fonctions FLET et LET-NAMED).

Ces deux types d'utilisation s'appliquent aussi bien aux fonctions nommées (associées à un symbole) qu'aux fonctions anonymes (les lambda-expressions), de n'importe quel type.

Il faut se rappeler que les définitions des fonctions sont stockées dans les propriétés naturelles F-VAL et F-TYPE des symboles. Leur utilisation directe étant fort peu commode, il existe un certain nombre de fonctions standards qui permettent de réaliser des définitions statiques ou dynamiques à peu de frais (voir le paragraphe : définition des fonctions du chapitre suivant).

2.5 L'évaluateur

L'évaluateur Le_Lisp (i.e. les fonctions interprète EVAL et APPLY) a été spécialement étudié du point de vue de la vitesse d'exécution.

Toutes les fonctions atomiques sont lancées très rapidement au moyen d'un branchement indirect indexé (par la F-VAL et le F-TYPE).

L'évaluateur minimise le nombre de CONS internes, en n'utilisant jamais la fonction EVLIS pour ses besoins propres (même pour évaluer les arguments des lambda-expressions et des NSUBR) sauf en cas de demande explicite au moyen d'une fonction de type LEXPR.

Une particularité du système Le_Lisp empruntée au système Vlisp est d'interpréter itérativement les fausses récursivités.

Itérativement est ici défini en termes de ressources : un appel de fonction est itératif s'il ne demande pas plus de ressources que celles accordées à l'entrée de la fonction. Les ressources en question sont les tailles de piles, ainsi que le nombre de cellules de listes.
C'est ainsi que dans :

```
(DE LOOPEVAL (IT)
   (LOOPEVAL (PRINT (EVAL (READ)))))
```

la suite des appels internes de LOOPEVAL ne provoquera PAS un débordement de pile, et bouclera, comme il se doit pour une boucle-système, indéfiniment.
Cette propriété se conserve, quelque soit le niveau d'imbrication des appels dits itératifs dans les structures de contrôle mises en jeu dans le corps de fonction.

Toutefois seul le compilateur traite les cas de co-post-recursion [Greussay 77].

2.6 Définition méta-circulaire de l'interprète

Enfin voici une description méta-circulaire de l'interprète Le_Lisp. Cette description permet d'avoir une vue globale du fonctionnement de l'interprète mais ne représente pas la véritable mise en œuvre qui est beaucoup plus performante notamment au niveau du nombre de CONS réalisés dans les fonctions de l'interprète et au niveau de l'encombrement de la pile : l'interprète Le_Lisp ne réalise aucun CONS pour ses besoins propres.

```
(DE TOP-LEVEL ()
    ; boucle principale
    (TAG ERROR
        (PRINT "Toplevel")
        (SETQ IT (LET ((STACK)) (EVAL (READ))))
        (PRINT "= " IT))
    (TOP-LEVEL))


(DE SYSERROR (f m a)
    ; traitement des erreurs de l'interprete
    (PRINT "** " f " : " m " : " a)
    (EXIT ERROR))


(DE EVAL (forme)
    ; evaluation d'une forme quelconque
    (COND
```

```
(DE PUSH l (WHILE l (NEWL stack (NEXTL l))))

(DE POP () (NEXTL stack))


; et pour les fanatiques, un veritable APPLY

(DE APPLY (fonct larg)
    ; applique la fonction fonct aux arguments larg
    (COND
      ((SYMBOLP fonct)
          (APPLYINTERNAL (TYPEFN fonct)
                         (CDDR (GETDEF fonct))
                         larg))
      ((OR (STRINGP fonct) (NUMBERP fonct))
          (SYSERROR 'APPLY "fonction indefinie" fonct))
      ((EQ (CAR fonct) 'LAMBDA)
          (APPLYINTERNAL 'EXPR fonct larg))
      ((EQ (CAR fonct) 'LAMBDA-NAMED)
          (APPLYLABEL (CADR fonct) (CDDR fonct) larg))
      (T (APPLY (EVAL fonct) larg))))


(DE APPLYINTERNAL (ftyp fval larg)
   ;application d'une fonction suivant son type
   (ELECTQ ftyp
      (SUBR  (CALL fval (CAR larg) (CADR larg) (CADDR larg) () ))
      (SUBRN (CALLN fval larg))
      (FSUBR (CALL fval larg () () () ))
      (EXPR  (EVALAMBDA fval larg))
      (FEXPR (EVALAMBDA fval (LIST larg)))
      (MACRO (EVAL (EVALAMBDA fval (CONS fonct larg))))
      (T (SYSERROR 'APPLY "fonction indefinie" fval)))))
```

CHAPITRE 3

LES FONCTIONS STANDARDS

Toutes les fonctions qui vont être décrites dans ce chapitre sont toujours résidentes et sont indépendantes du système utilisé.

Pour chacune d'elles on donnera son type (SUBR ou FSUBR) ainsi que le nombre standard d'arguments, et pour chaque argument le type souhaité ou requis, ces types étant notés :

- <s> pour une S-expression quelconque
- <l> pour une liste
- <a> pour un atome quelconque (symbole, nombre ou chaîne de caractères)
- <sym> pour un symbole
- <n> pour un nombre
- <str> pour une chaîne de caractères
- <ch> pour un caractère (i.e. un atome dont le P-NAME n'a qu'un caractère)
- <cn> pour le code ASCII d'un caractère
- <fn> pour une fonction (un symbole, une lambda-expression anonyme ou nommée)

Dans la mesure du possible, les SUBR seront décrites en Lisp sous forme de DE, DF ou DM. Ces descriptions ne sont que les équivalents Lisp de ces fonctions et ne représenteront que leurs caractéristiques fondamentales. Les fonctions standards sont ordonnées par grandes familles, durant ces descriptions de nombreuses références avants apparaitront.

Egalement au niveau de chaque fonction nous donnerons des indications de transportabilité vers les autres grands systèmes lisp : Maclisp, Vlisp et Interlisp.

3.1 Les fonctions d'évaluation

(EVAL <s>) [SUBR à 1 argument]

C'est la fonction principale de l'interprète. EVAL retourne la valeur de l'évaluation de l'argument <s> (voir la description complète de cette fonction dans le chapitre précédent).

```
ex : (EVAL '(1+ 55))                    ->   56
     (EVAL (LIST '+ 8 '(1+ 3)))         ->   12
     (EVAL '((CAR '(CDR)) '(A B C)))    ->  (B C)
```

Compatibilité : indépendemment des différences de fonctionnement interne de l'interprète des différents systèmes lisp, Maclisp et certaines versions de Vlisp possèdent des arguments optionnels supplémentaires (pointeur de pile, indicateurs en tout genre, CHRONOLOGY ...). Il est préférable de s'en tenir à une fonction EVAL à 1 argument.

(EVLIS <l>) [SUBR à 1 argument]

retourne une liste composée des valeurs des évaluations de tous les éléments de la liste <l>.

EVLIS peut être défini en Lisp de la manière suivante :

```
(DE EVLIS (l)
    (IF (NULL l)
        ()
        (CONS (EVAL (CAR l)) (EVLIS (CDR l)))))

ex : (SETQ L '((1+ 5) (1+ 7) (1+ 9)))  ->  ((1+ 5) (1+ 7) (1+ 9))
     (EVLIS L)                          ->   (6 8 10)
```

(EPROGN <l>) [SUBR à 1 argument]

évalue séquentiellement tous les éléments de la liste <l>. EPROGN retourne la valeur de la dernière évaluation (i.e. celle du dernier élément de <l>).

EPROGN peut être défini en Lisp de la manière suivante :

```
(DE EPROGN (l)
    (IF (NULL (CDR l))
        (EVAL (CAR l))
        (EVAL (CAR l))
        (EPROGN (CDR  l))))

ex : (SETQ L '((PRIN 1)(PRIN 2)(PRIN 3))) -> ((!
PRIN 1)(PRIN 2)(PRIN 3))
     (EPROGN L)  1 2 3  ->  3
```

(PROG1 <s1> ... <sN>) [FSUBR]

évalue séquentiellement les différentes expressions <s1> ... <sN>. PROG1 retourne la valeur de la première évaluation (i.e. celle de <s1>).

PROG1 peut être défini en Lisp de la manière suivante :

```
(DF PROG1 (l)
    (LET ((result (EVAL (CAR l))))
         (EPROGN (CDR l))
         result))
```

ou bien sous la forme d'une EXPR :

```
(DE PROG1 l (CAR l))
```

```
ex : (PROG1 (PRIN 1)(PRIN 2)(PRIN 3))  1 2 3  ->  1
```

PROG1 est toujours utilisé pour réaliser des effets de bord. Voici une MACRO (EXCH var1 var2) qui réalise l'échange des valeurs des variables var1 et var2 sans utiliser de mémoire auxiliaire :

```
(DM EXCH (l)
    (LIST 'SETQ (CADR l)
         (LIST 'PROG1 (CADDR l)
               (LIST 'SETQ (CADDR l) (CADR l))))))

; avec cette MACRO un appel de type :  (EXCH var1 var2)

; est expanse en :   (SETQ var1 (PROG1 var2 (SETQ var2 var1)))
```

(PROGN <s1> ... <sN>) [FSUBR]

évalue en séquence les différentes expressions <s1> ... <sN>. PROGN retourne la valeur de la dernière évaluation (i.e. celle de <sN>).

PROGN est la forme FSUBRée de la fonction EPROGN et peut donc être décrite sous forme de FEXPR de la manière suivante :

```
(DF PROGN (l) (EPROGN l))

; ou bien sous forme d'EXPR :

(DE PROGN l (CAR (LAST l)))
```

```
ex : (PROGN (PRIN 1)(PRIN 2)(PRIN 3))  1 2 3  ->  3
```

(QUOTE <s>) [FSUBR]

retourne la S-expression <s> non-évaluée. Cette fonction est utilisée comme argument des fonctions de type SUBR ou EXPR dont on ne désire pas évaluer les arguments.
Il existe un macro-caractère standard qui facilite cette écriture, le caractère apostrophe (quote) ' . Ce caractère placé devant une expression quelconque <s> fabriquera à la lecture la liste (QUOTE <s>).
De même les fonctions de sortie impriment une liste de type (QUOTE <s>) en utilisant la notation '<s>, toujours pour des raisons de lisibilité.

QUOTE peut être défini en Lisp de la manière suivante :

```
(DF QUOTE (L) (CAR L))

ex : (QUOTE (1+ 4))    ->  (1+ 4)
     '(A (B C))        ->  (A (B C))
     'A                ->  A
     ''A               ->  'A
     '''A              ->  ''A
     '(QUOTE A B)      ->  (QUOTE A B)
```

(FUNCTION <s>) [FSUBR]

est identique à la fonction QUOTE. FUNCTION a été rajouté dans Le_Lisp pour des raisons de compatibilité avec certains Lisp qui utilisent cette fonction comme déclaration pour le compilateur, mais dont l'effet est identique à la fonction QUOTE.

(IDENTITY <s>) [SUBR à 1 argument]

comme son nom l'indique, cette fonction est la fonction identité et retourne comme il se doit son argument.

IDENTITY peut être défini en Lisp de la manière suivante :

```
(DE IDENTITY (S) S)

ex : (IDENTITY 'A)       ->  A
     (IDENTITY (1+ 5))   ->  6
```

(COMMENT <e1> ... <eN>) [FSUBR]

retourne le symbole COMMENT lui-même sans évaluer aucun des arguments. Cette fonction est très utile pour rendre ineffective une S-expression quelconque à l'intérieur d'un fichier mais est à déconseiller pour l'introduction de véritables commentaires qui sont plutot écrit au moyen du caractère spécial point-virgule.
ATTENTION : cette fonction ne peut être placé qu'à l'intérieur d'un PROGN (implicite ou explicite) sous peine de perturber l'évaluation.

COMMENT peut être défini en Lisp de la manière suivante :

```
(DF COMMENT (L) 'COMMENT)
```

```
ex : (COMMENT 'FOO BAR)     ->  COMMENT
     (COMMENT)              ->  COMMENT
     (COMMENT MAIS NON)     ->  COMMENT
```

(DECLARE <e1> ... <eN>) [FSUBR]

comme la fonction précédente COMMENT, DECLARE n'évalue rien et retourne toujours le symbole DECLARE. Cette fonction a été rajoutée dans le système Le_Lisp pour pouvoir lire des fichiers Maclisp qui contiennent des déclarations au compilateur Maclisp. Le_Lisp, quant à lui, possède un compilateur qui ne nécessite aucune déclaration et de ce fait est beaucoup plus facile à utiliser. Ce compilateur est décrit dans le deuxième volume du manuel.

3.2 Les fonctions d'application

(LAMBDA <l> <s1> ... <sN>) [FSUBR]

la valeur d'une lambda-expression est cette lambda-expression elle-même. Cette nouvelle fonction a été rajoutée pour éviter de *quoter* les lambda-expressions explicites anonymes dans les fonctions d'application.

LAMBDA peut être défini en Lisp de la manière suivante :

```
(DF LAMBDA (L) (CONS 'LAMBDA L))

; ou bien sous la forme d'une MACRO

(DM LAMBDA (L) L)

; cette definition est plus exacte mais fait boucler la fonction
; interprete EVAL (voyez-vous pourquoi ?)
```

```
ex : (LAMBDA (X) X)                    ->   (LAMBDA (X) X)
     (MAPC (LAMBDA (X) (PRIN X))
           '(A B C)) A B C             ->   NIL
```

(LAMBDA-NAMED <sym> <l> <s1> ... <sN>) [FSUBR]

la valeur d'une lambda-expression nommée est cette lambda-expression elle-même. Cette nouvelle fonction a été rajoutée pour éviter de *quoter* les lambda-expressions explicites nommées dans les fonctions d'application.

LAMBDA-NAMED peut être défini en Lisp de la manière suivante :

```
(DF LAMBDA-NAMED (L) (CONS 'LAMBDA-NAMED L))
```

```
ex : (LAMBDA-NAMED FOO (X) X)        ->    (LAMBDA-NAMED FOO (X) X)
```

Compatibilité : LAMBDA-NAMED est la version moderne de l'ancienne fonction de LISP 1.5 LABEL qui peut très aisemment se macro-générer sous la forme d'une LAMBDA expression nommée.

```
(DM LABEL (call)
    (DISPLACE call (LIST 'LAMBDA-NAMED (CADR call) (CDADDR c!
all))))
```

3.2.1 Les fonctions d'application simples

(APPLY <fn> <l>) [SUBR à 2 arguments]

retourne la valeur de l'application de la fonction <fn> à la liste d'arguments <l>. La fonction <fn> peut être une fonction de n'importe quel type SUBR NSUBR FSUBR EXPR FEXPR ou MACRO.

```
ex : (APPLY (LAMBDA (x y) (+ x y)) (LIST (1+ 8) (REM 10 3)))  ->  10
```

Compatibilité : de même que pour EVAL, Maclisp possède un 3ème argument à APPLY, qui n'est compatible avec aucun autre système Lisp.

(FUNCALL <fn> <s1> ... <sN>) [SUBR à N arguments]

est équivalente à la fonction APPLY. Toutefois les arguments de la fonction ne sont pas regroupés sous la forme d'une liste mais sont arguments de la fonction FUNCALL. FUNCALL est donc une autre forme de la fonction APPLY.
L'appel (FUNCALL <fn> <s1> ... <sN>) est équivalent à l'appel (APPLY <fn> (LIST <s1> ... <sN>)).

FUNCALL peut être défini en Lisp de la manière suivante :

```
(DE FUNCALL 1
    (APPLY (CAR 1) (CDR 1)))
```

```
ex : (FUNCALL (LAMBDA (x y) (CONS x y)) 'A 'B)   ->   (A . B)
```

Compatibilité : cette fonction s'appelle APPLYN en Vlisp et APPLY en Interlisp. Maclisp ne permet quant à lui d'appeler les FEXPR ou les FSUBR que si elles possèdent 1 seul argument.*

3.2.2 Les fonctions d'application de type MAP

Ces fonctions permettent d'appliquer une fonction successivement à un ensemble de listes arguments. Les fonction appliquées peuvent avoir un nombre quelconque d'arguments. Dans les descriptions Lisp accompagnant ces fonctions nous utiliserons les fonctions auxiliaires suivantes :

```
(DE ALLCAR (l)
    ; l est une liste de listes, retourne la liste de tous leur CAR
    (IF (NULL l) () (CONS (CAAR l) (ALLCAR (CDR l)))))

(DE ALLCDR (l)
    ; l est une liste de listes, retourne la liste de tous leur CDR
    (IF (NULL l) () (CONS (CDAR l) (ALLCDR (CDR l)))))
```

Compatibilité : pour toutes les fonctions de type MAPxx qui vont suivre, Vlisp en général n'autorise qu'une seule liste d'argument <l1> ce qui ne permet d'appliquer que des fonctions à un argument, Interlisp quant à lui, en plus de la limitation de Vlisp, possède les arguments inversés (MAP l fnt) et un 3ème argument optionnel

(MAP <fn> <l1> ... <lN>) [SUBR à N arguments]

applique successivement la fonction <fn> avec toutes les listes <li> comme arguments puis de nouveau avec tous les CDR de ces listes jusqu'a ce la première de ces listes argument devienne égale à NIL. MAP retourne toujours NIL en valeur.

MAP peut être défini en Lisp de la manière suivante :

```
(DE MAP l
    (LET ((f (CAR l)))
         (LET-NAMED MAPl ((l (CDR l)))
              (WHEN (CAR l)
                    (APPLY f l)
                    (MAPl (ALLCDR l))))))

ex : (MAP  (LAMBDA (x) (PRINT x)) '(A (B C) D))
     (A (B C) D)
     ((B C) D)
     (D)
     ->  NIL
```

(MAPC <fn> <l1> ... <lN>) [SUBR à N arguments]

applique successivement la fonction <fn> avec tous les CAR de toutes les listes <li> comme arguments puis avec tous les CADR puis avec tous les CADDR ... jusqu'a ce que la première liste argument soit égale à NIL. MAPC retourne NIL en valeur.

MAPC peut être défini en Lisp de la manière suivante :

```
(DE MAPC l
    (LET ((f (CAR l)))
```

```
                (LET-NAMED MAP1 ((l (CDR l)))
                     (WHEN (CAR l)
                           (APPLY f (ALLCAR l))
                           (MAP1 (ALLCDR l))))))

ex:  (MAPC (LAMBDA (x) (PRINT x)) '(A (B C) D))
     A
     (B C)
     D
     ->  NIL
```

(MAPLIST <fn> <l1> ... <lN>) [SUBR à N arguments]

applique successivement la fonction <fn> à toutes les listes <li> puis à tous leur CDR. MAPLIST est donc identique à la fonction MAP du point de vue application de fonction mais retourne en valeur la liste des valeurs de toutes les applications successives.

MAPLIST peut être défini en Lisp de la manière suivante :

```
(DE MAPLIST l
    (LET ((f (CAR l)))
         (LET-NAMED MAP1 ((l (CDR l)))
              (WHEN (CAR l)
                    (CONS (APPLY f l)
                          (MAP1 (ALLCDR l))))))))

ex : (MAPLIST  (LAMBDA (x) (PRINT x)) '(A (B C) D))
     (A (B C) D)
     ((B C) D)
     (D)
     ->  ((A (B C) D) ((B C) D) (D))
```

(MAPCAR <fn> <l1> ... <lN>) [SUBR à N arguments]

applique successivement la fonction <fn> avec tous les CAR de toutes les listes <li> comme arguments. MAPCAR est donc identique fonctionnellement à la fonction MAPC mais retourne en valeur la liste des valeurs des applications successives.

MAPCAR peut être défini en Lisp de la manière suivante :

```
(DE MAPCAR l
    (LET ((f (CAR l)))
         (LET-NAMED MAP1 ((l (CDR l)))
              (WHEN (CAR l)
                    (CONS (APPLY f (ALLCAR l))
                          (MAP1 (ALLCDR l))))))))

ex:  (MAPCAR (LAMBDA (x) (PRINT x)) '(A (B C) D))
     A
     (B C)
     D
```

```
->  (A (B C) D)
(MAPCAR 'CONS '(A B C) '(1 2))   ->  ((A . 1) (B . 2) (C))
```

(MAPCON <fn> <l1> ... <lN>) [SUBR à N arguments]

applique successivement la fonction <fn> avec toutes les listes <li> comme arguments puis avec tous leur CDR. MAPCON est identique fonctionnellement à la fonction MAP. Chaque évaluation doit retourner une liste en valeur. MAPCON retourne la liste des valeurs des évaluations qui sont rassemblées au moyen de la fonction NCONC (voir cette fonction).

MAPCON peut être défini en Lisp de la manière suivante :

```
(DE MAPCON l
    (LET ((f (CAR l)))
         (LET-NAMED MAP1 ((l (CDR l)))
              (WHEN (CAR l)
                   (NCONC (APPLY f l)
                          (MAP1 (ALLCDR l)))))))

ex:  (MAPCON (LAMBDA (x y) (cons x y))
             '(1 2 3)
             '(4 5 6))
->        ((1 2 3) 4 5 6 (2 3) 5 6 (3) 6)
```

(MAPCAN <fn> <l1> ... <lN>) [SUBR à N arguments]

applique successivement la fonction <fn> avec tous les CAR de toutes les listes <li> comme arguments. MAPCAN est identique fonctionnellement à la fonction MAPC. Chaque évaluation doit retourner une liste en valeur. MAPCAN retourne la liste des valeurs des évaluations qui sont rassemblées au moyen de la fonction NCONC (voir cette fonction).

MAPCAN peut être défini en Lisp de la manière suivante :

```
(DE MAPCAN l
    (LET ((f (CAR l)))
         (LET-NAMED MAP1 ((l (CDR l)))
              (WHEN (CAR l)
                   (NCONC (APPLY f (ALLCAR l))
                          (MAP1 (ALLCDR l)))))))

ex:  (MAPCAN (LAMBDA (x y) (list (1+ x) (1- y)))
             '(1 2 3)
             '(1 2 3))     ->      (2 0 3 1 4 2)
```

(MAPOBLIST <fn>) [SUBR à 1 argument]

applique successivement à la fonction <fn>, qui doit être à 1 argument, tous les éléments de la liste des symboles (l'OBLIST). Cette fonction est donc équivalente à :

(MAPC <fn> (OBLIST))

mais est beaucoup plus efficace en particulier parcequ'elle ne construit pas la liste complète des symboles, liste qui possède typiquement plus de 2000 éléments.

```
ex : fonction qui retourne la liste de toutes les fonctions
     de type FSUBR du systeme :

     (DE FINDFSUBR ()
         (LET ((l))
              (MAPOBLIST (LAMBDA (sym)
                                 (WHEN (EQ (TYPEFN sym) 'FEXPR)
                                       (NEWL l x))))
              l))
```

3.2.3 Les fonctions manipulant l'environnement

Ces fonctions vont pouvoir changer l'environnement temporairement. Le terme environnement est pris ici comme l'ensemble des couples variable-valeur. Au sortir de ces 4 fonctions l'environnement initial (i.e. celui avant l'appel de l'une de ces fonctions) est restitué automatiquement.

(LET <l> <s1> ... <sN>) [FSUBR]

permet d'appeler une fonction anonyme de type EXPR (i.e. une lambda-expression).
<l> est une liste de la forme :

-() dans le cas où il n'y a pas de variable
-((<var1> <val1>) ... (<varN> <valN>)) dans le cas de variables multiples.

<s1> ... <sN> est un corps de fonction.

La fonction LET va lier dynamiquement et en parallèle toutes les variables <vari> aux valeurs <vali> puis lancer l'exécution du corps de la fonction <s1> ... <sN>. Au sortir du corps de cette fonction les variables seront déliées et retrouveront leurs anciennes valeurs. LET permet de définir des variables locales très facilement.

LET est une forme abrégée d'appel de fonctions anonymes de type EXPR.

Ainsi la forme
(LET ((<var> Ival>)) Is1> ... IsN>)

correspond à l'appel
((LAMBDA (<var>) Is1> ... IsN>) Ival>)

et la forme
(LET ((<var1> Ival1>) ... (<varN> IvalN>)) Is1> ... IsN>)
correspond à l'appel
((LAMBDA (<var1> ... IvarN>) Is1> ... IsN>) Ival1> ... IvalN>)

LET
peut être défini en Lisp de la manière suivante :

```
(DM LET (ls)
    (DISPLACE ls
       (COND ((NULL (CADR ls))
                  (CONS (CONS LAMBDA (CONS () (CDDR ls)))))
             (T (CONS  (CONS LAMBDA (CONS (MAPCAR 'CAR (CADR ls))
                                          (CDDR ls)))
                       (MAPCAR 'CADR (CADR ls))))))))

; ou d'une maniere plus lisible :

(DEFMACRO (varl . body)
          (IF (NULL varl)
              ((LAMBDA () ,@body))
              ((LAMBDA ,@(MAPCAR 'CAR ,varl) ,@body)
               ,@(MAPCAR 'CADR ,varl))))
```

(LETV <lsym> <lval> <s1> ... <sN>) **[FSUBR]**

est identique à la fonction LET mais va permettre de calculer le nom des variables qui seront liées. <lsym> est un argument qui une fois évalué doit livrer une liste de symboles, <lval> est un argument qui une fois évalué doit livrer une liste de valeurs. LETV va lier les symboles de la liste <lsym> respectivement avec les valeurs de la liste <lval> puis, dans ce nouvel environnement évaluer les différentes expressions <s1> ... <sN>. LETV permet donc de lier des variables dont le nom est calculé et donc de ce fait d'écrire de nouveaux interprètes.

LETV peut être défini en Lisp de la manière suivante :

```
(DEFMACRO LETV (varl vall . body)
          ((LAMBDA ,varl ,@body) ,@vall))
```

Compatibilité : cette fonction n'existe pas en Vlisp et se nomme PROGV en Maclisp. Elle correspond à un dictionnaire smalltalk.

(LETSEQ <l> <s1> ... <sN>) [FSUBR]

cette fonction est identique à la fonction LET mais les arguments sont liés séquentiellement (et non en parallèle). Il y a donc la même différence qu'entre les fonctions PSETQ et SETQ ce qui permet d'utiliser dans le calcul d'une expression la valeur d'une variable précédemment liée dans le LETSEQ lui-même.

Ainsi la forme :

```
(LETSEQ ((<var1> Ival1>) (<var2> Ival2>) ... (<varN> IvalN>))
        Is1> ... IsN>)
```

correspond à :

```
(LET ((var1> Ival1>))
     (LET ((var2> Ival2>))
          .....
          (LET ((varN> IvalN>))
               Is1> ... IsN>) ... ))
```

et également à :

```
((LAMBDA (<var1>)
         ((LAMBDA (<var2>)
                  ........
                  ((LAMBDA (<varN>)
                           Is1> ... IsN>)
                   IvalN>) ...)
          Ival2>)
 Ival1>)
```

(LET-NAMED <sym> <l> <s1> ... <sN>). [FSUBR]

permet d'appeler une fonction nommée de type EXPR (i.e. une lambda-expression nommée LAMBDA-NAMED).

<sym> est le nom de la fonction à créer
<l> est une liste de couples variable valeur comme dans la fonction LET
<s1> ... <sN> est un corps de fonction.

La fonction LET-NAMED va lier dynamiquement et en parallèle toutes les variables avec toutes les valeurs de la liste <l> puis lancer le corps de la fonction <s1> ... <sN> après avoir associé (juste le temps de l'évaluation du corps) au nom <sym> la même lambda-expression que dans la fonction LET. LET-NAMED permet donc la définition dynamique et l'appel d'une fonction récursive.

3.3 Les fonctions de définition de fonctions

Ces fonctions vont permettre de définir de nouvelles fonctions. Elles testent la validité de leurs arguments :
- les noms des fonctions doivent être des symboles
- toutes les variables de la liste des paramètres formels doivent également être des symboles.

Si ce n'est pas le cas, une erreur apparait dont le libellé est :

** <fnt> : mauvaise définition : <sym> ou bien
** <fnt> : bad definition : <sym>

Il existe 2 types de définition de fonction : les définitions statiques et les définitions dynamiques (voir le chapitre précédent sur le fonctionnement de l'interprète).

3.3.1 Définitions des fonctions statiques

Toutes ces fonctions vont changer de façon permanente les fonctions associées aux symboles. Toute modification est précédée d'un sauvetage de l'ancienne définition associée au symbole (s'il en possédait une) sur la P-liste de ce symbole sous l'indicateur PREVIOUS-DEF. De plus l'indicateur STATUS-REDEF permet de valider ou non ces redéfinitions. Si STATUS-REDEF est faux, un message d'avertissement est lançé en cas de tentative de redéfinition de fonction. Si cet indicateur est vrai (ce qui est l'option par défaut) les redéfinitions ne provoquent pas d'impression de message. Le message à la forme :

** <fnt> : fonction redéfinie : <sym> ou bien
** <fnt> : redefined function : <sym>

dans lequel <fnt> est la fonction de définition qui a été appelé et <sym> est le symbole qui devait être redéfini.

(DE <sym> <lvar> <s1> ... <sN>) [FSUBR]

permet de définir statiquement de nouvelles EXPR. <sym> est le nom du symbole auquel sera rattaché une lambda-expression de type :

(LAMBDA <lvar> <s1> ... <sN>)

DE (prononcez *dé* comme en latin) retourne le nom de la fonction <sym> en valeur.

```
  ex : (DE FOO (1) (IF (NULL (CDR 1)) 1 (FOO (CDR 1)))  ->  FOO
       (GETFN 'FOO))    ->  (LAMBDA (1) (IF (NULL (CDR 1)) 1 (!
FOO (CDR 1))))
       (TYPEFN 'FOO)    ->  EXPR
       (FOO '(A B C))   ->  (C)
```

(DF <sym> <lvar> <s1> ... <sN>) [FSUBR]

permet de définir statiquement de nouvelles FEXPR. <sym> est le nom du symbole auquel sera rattaché une lambda expression de type :

(LAMBDA <lvar> <s1> ... <sN>)

Du fait de son type FEXPR, cette lambda-expression sera évaluée d'une manière spéciale par l'interprète (voir le chapitre précédent). DF retourne en valeur le nom <sym> de la fonction définie.

```
ex : (DF INCR (L) (SET (CAR L) (1+ (CAAR L))))   ->  INCR
     (TYPEFN 'INCR)                              ->  FEXPR
     (SETQ nb 7)                                 ->  7
     (INCR nb)                                   ->  8
```

(DM <sym> <lvar> <s1> ... <sN>) [FSUBR]

permet de définir statiquement de nouvelles MACRO. <sym> est le nom du symbole auquel sera rattaché une lambda expression de type :

(LAMBDA <lvar> <s1> ... <sN>)

Du fait de son type MACRO, cette lambda-expression sera évaluée d'une manière spéciale par l'interprète (voir le chapitre précédent). DM retourne en valeur le nom <sym> de la fonction définie.

```
ex : (DM DECR (1) (LIST 'SETQ (CADR 1) (LIST '1- (CADR 1))))
         ->  DECR
     (SETQ N 10)  ->  10
     (DECR N)     ->  9
```

(DS <sym> <adr> <type>) [FSUBR]

permet de définir statiquement une fonction de type SUBR ou FSUBR. <sym> est le nom d'un symbole auquel sera rattachée une fonction écrite en langage machine qui débute à l'adresse <adr>. Le type codé numériquement (et dépendant des implantations du système) est fourni par l'argument <type>. Cette fonction est utilisée principalement par le compilateur.

3.3.2 Définitions avancées des Fonctions

Ces fonctions de définition vont permettre de réaliser un nouveau type de liaison. Il ne s'agit plus de lier une liste de variables à une liste de valeurs mais un arbre de variables à un arbre de valeurs. Cette liaison d'arbre utilise toujours la fontion DESET.

Dans les fonctions qui vont suivre, <tvar> représente un arbre de valeurs.

(DEFEXPR <sym> <tvar> <s1> ... <sN>) **[FSUBR]**

permet de définir statiquement une nouvelle EXPR de nom <sym> qui accepte un arbre de variables <tvar>. La liaison s'effectue entre l'arbre des variables <tvar> et la liste des valeurs des arguments. DEFEXPR retourne <sym> en valeur.

```
ex : (DEFEXPR FOO ((L1 . L2) . L3)
                  (LIST L1 L2 L3))
     -> FOO
     (FOO '((1 2 3) (4 5 6)))  ->  (1 (2 3) ((4 5 6)))
```

(DEFFEXPR <sym> <tvar> <s1> ... <sN>) **[FSUBR]**

permet de définir statiquement une nouvelle EXPR de nom <sym> qui accepte un arbre de variables <tvar>. La liaison s'effectue entre l'arbre des variables <tvar> et le CDR (non évalué) de l'appel de la fonction. DEFFEXPR retourne <sym> en valeur.

```
ex : (DEFFEXPR WHEN (test . then)
                (IF (EVAL test)
                    (EPROGN then)
                    ()))
     -> WHEN
```

(DEFMACRO <sym> <tvar> <s1> ... <sN>) **[FSUBR]**

permet de définir statiquement une nouvelle MACRO de nom <sym> qui accepte un arbre de variables <tvar>. La liaison s'effectue avec l'arbre des variables <tvar> et l'appel de la MACRO tout entier. DEFMACRO retourne <sym> en valeur.

```
ex : (DEFMACRO IF (TEST THEN ELSE1 . ELSE)
                (COND ((,TEST ,THEN) (T ,ELSE1 ,@ELSEN))))
```

(MACROEXPAND1 <s>) **[SUBR à 1 argument]**

Si l'expression <s> est un appel de MACRO, MACROEXPAND1 retourne la valeur de l'expansion de cette MACRO. Très utile évidemment pour tester ses propres macros.

MACROEXPAND1 peut être défini en Lisp de la manière suivante :

```
(DE MACROEXPAND1 (x)
    ; expanse une macro juste un coup pour voir
    (IF (EQ (TYPEFN (CAR x)) 'MACRO)
        (APPLY (GETFN (CAR x)) (LIST x))
        x)))
```

(MACROEXPAND <s>) **[SUBR à 1 argument]**

permet d'expanser une expression tant que celle-ci le permet.

MACROEXPAND peut être défini en Lisp de la manière suivante :

```
(DE MACROEXPAND (x)
    ; expanse un appel tant que l'on tombe sur une macro
    (IF (EQ (TYPEFN (CAR x)) 'MACRO)
        (MACROEXPAND (APPLY (GETFN (CAR x)) (LIST x)))
        x))
```

3.3.3 Définition des fermetures

(CLOSURE <lvar> <fn>) **[SUBR à 2 arguments]**

permet de définir une fermeture composée d'une liste de variables <lvar> et d'une fonction <fn>. Toutes les variables de la liste <lvar> doivent avoir une valeur non indéfinie qui sera utilisée pour construire la fermeture.

CLOSURE peut être défini en Lisp de la manière suivante :

```
(de closure (lvarclot fnt)
    (let ((listval (mapcar '(lambda (val) (list 'quote (eval! val)))
                           lvarclot))
          (lvar (cadr fnt))
          (corps (cddr fnt)))
        (lambda ,lvar
            ((lambda ,lvarclot
                (unwind-protect (progn ,@corps)
                    ,@(mapcar '(lambda (slot var)
                                  (rplaca (cdr ,slot) (quote ,slot))
                              ,var))
```

(REVERT <sym>) [SUBR à 1 argument]

permet de retrouver l'ancienne définition associée au symbole <sym> (s'il en avait une).

REVERT peut être défini en Lisp de la manière suivante :

```
(DE REVERT (sym)
    (LET ((oldef (GET sym 'PREVIOUS-DEF)))
         (IF oldef
             (EVAL oldef)
             (SYSERROR 'REVERT "il n'avait pas de definition"!
sym))))
```

(SYNONYM <sym1> <sym2>) [SUBR à 2 arguments]

permet de donner au symbole <sym1> le type et la valeur de la fonction associée au symbole <sym2>.
SYNONYM peut être défini en Lisp de la manière suivante :

```
(DE SYNONYM (at1 at2)
    (LET ((defat2 (GETDEF at2)))
         (IF defat2
             (EVAL (MCONS (CAR defat2) at1 (CDDR defat2)))
             (SYSERROR ' SYNONYM "ce n'est pas une fonctio!
n" at2))))
```

```
ex :   (SYNONYM 'KONS 'CONS)    ->  KONS
       (KONS 'A 'B)             ->  (A . B)
```

(SYNONYMQ <sym1> <sym2>) **[FSUBR]**

cette fonction est la forme FSUBRée de la fonction précédente.

SYNONYMQ peut être défini en Lisp de la manière suivante :

```
(DF SYNONYMQ (l)
    (SYNONYM (CAR l) (CADR l)))
```

ex : (SYNONYMQ FOO BAR) est equivalent a (SYNONYM 'FOO 'BAR)

(TYPEFN <sym>) **[SUBR à 1 argument]**

retourne le type de la fonction associée au symbole <sym> ou NIL si le symbole ne possède pas de définition de fonction. Le type d'une fonction peut être l'un des atomes suivant : EXPR, FEXPR, MACRO, SUBR, NSUBR ou FSUBR.

(GETFN <sym>) **[SUBR à 1 argument]**

permet de retourner la lambda-expression (de type LAMBDA, FLAMBDA ou MLAMBDA) associée au symbole <sym>.

```
ex :  (DE BAR (N) (+ N N))  ->  BAR
      (TYPEFN 'BAR)         ->  EXPR
      (GETFN 'BAR)          ->  (LAMBDA (N) (+ N N))
      (DM FOO (x) (CAR x))  ->  FOO
      (TYPEFN 'FOO)         ->  MACRO
      (GETFN 'FOO)          ->  (MLAMBDA (x) (CAR x))
```

(GETDEF <sym>) **[SUBR à 1 argument]**

permet de retourner la définition de la fonction associée au symbole <sym> sous la forme de sa définition (i.e. sous la forme de l'appel d'une des fonctions DE/DF/DM/DS).

```
ex :  (DE BAR (N) (+ N N))  ->  BAR
      (TYPEFN 'BAR)         ->  EXPR
      (GETDEF 'BAR)         ->  (DE BAR (N) (+ N N))
```

(MAKUNBOUNDFN <sym>) **[SUBR à 1 argument]**

détruit la définition de fonction associée au symbole <sym>. MAKUNBOUNDFN retourne <sym> en valeur.

```
ex : (DE FOO (x) (+ x x))    ->  FOO
     (FOO 10)                ->  20
     (MAKUNBOUNDFN 'FOO)     ->  FOO
     (FOO 10)                ->
     ** EVAL : fonction indefinie : FOO
```

3.3.5 Définitions des fonctions dynamiques

(FLET <l> <s1> ... <sN>) [FSUBR]

permet de définir des fonctions dynamiquement (d'une manière analogue à la fonction LET pour les variables).

<l> est une liste de la forme :
(<sym> <l> <e1> ... <eN>)
<s1> ... <sN> est un corps de fonction.

FLET va lier, le temps de l'évaluation de <s1> ... <sN>, une nouvelle fonction à l'atome <sym>. Cette fonction, qui est toujours de type EXPR, possède une liste d'arguments <l> et un corps <e1> ... <eN>. La valeur retournée par FLET est la valeur de la dernière évaluation du corps du FLET i.e. <sN>. Au sortir du FLET, le symbole <sym> reprendra son ancienne définition (s'il en avait une).

[Attention : cette syntaxe risque de changer très prochainement]

```
ex : (FLET (CAR (x) (CDR x)) (CAR '(A B C)))  ->  (B C)
     (CAR '(A B C))                           ->  A
```

3.4 Les fonctions de contrôle

Toutes les fonctions de cette section vont permettre de rompre le déroulement séquentiel des évaluations. C'est la raison pour laquelle elles sont toutes de type FSUBR i.e. que les arguments ne seront pas évalués par EVAL mais par les fonctions elles-mêmes et cela sélectivement.

La fonction de contrôle la plus simple est la fonction conditionnelle IF. Cette fonction va être utilisée avec la fonction WHILE pour décrire en Lisp toutes les autres fonctions de contrôle sous forme de FEXPR. Rappelons qu'à l'appel d'une fonction de type FEXPR la liste des arguments non évalués (i.e. le CDR de l'appel lui-même) est liée à la 1ère variable de la liste des variables et le reste des variables de cette liste sont toutes liées à la valeur NIL.

3.4.1 Les fonctions de contrôle de base

(IF <s1> <s2> <s3> ... <sN>) [FSUBR]

si la valeur de l'évaluation de <s1> est différente de NIL, IF retourne la valeur de l'évaluation de l'expression <s2>, sinon IF évalue en séquence les différentes expressions <s3> ... <sN> et retourne la valeur de la dernière évaluation <sN>. IF permet de construire une structure de contrôle de type :

si <s1> alors <s2> sinon <s3> ... <sN>

```
ex : (IF T 1 2 3)     ->  1
     (IF NIL 1 2 3)   ->  3

     ; voici la fonction d'ACKERMANN decrite avec des IF

     (DE ACK (x y)
         (IF (= x 0)
             (1+ y)
             (ACK (1- x)
                  (IF (= y 0)
                      1
                      (ACK x (1- y))))))
```

Compatibilité : cette fonction n'existe pas en Maclisp ni en Interlisp.

(IFN <s1> <s2> <s3> ... <sN>) [FSUBR]

si la valeur de l'évaluation de <s1> est égale à NIL, IFN retourne la valeur de l'évaluation de l'expression <s2>, sinon IFN évalue en séquence les différentes expressions <s3> ... <sN> et retourne la valeur de la dernière évaluation <sN>. IFN permet de construire une structure de contrôle de type :

si non <s1> alors <s2> sinon <s3> ... <sN>

IFN est donc équivalent à (IF (NOT <s1>) <s2> <s3> ... <sN>).

IFN peut être défini en Lisp de la manière suivante :

```
(DF IFN (l)
    (IF (NOT (EVAL (CAR l)))
        (EVAL (CADR l))
        (EPROGN (CDDR l))))

ex : (IFN T 1 2 3)     ->  3
     (IFN NIL 1 2 3)   ->  1
```

Compatibilité : cette fonction n'existe ni en Maclisp ni en Interlisp.

(WHEN <s1> <s2> ... <sN>) [FSUBR]

si la valeur de l'évaluation de <s1> est différente de NIL, WHEN évalue en séquence les différentes expressions <s2> ... <sN> et retourne la valeur de la dernière évaluation <sN>. Si la valeur de <s1> est égale à NIL, WHEN retourne NIL. WHEN permet de construire une structure de contrôle de type :

si <s1> alors <s2> ... <sN> sinon NIL

WHEN peut être défini en Lisp de la manière suivante :

```
(DF WHEN (l)
    (IF (EVAL (CAR l))
        (EPROGN (CDR l))
        NIL))

ex : (WHEN T 1 2 3)    ->  3
     (WHEN NIL 1 2 3)  ->  NIL
```

Compatibilité : cette fonction n'existe que dans les post-Maclisp Lisps.

(UNLESS <s1> <s2> ... <sN>) [FSUBR]

si la valeur de l'évaluation de <s1> est égale à NIL, UNLESS évalue en séquence les différentes expressions <s2> ... <sN> et retourne la valeur de la dernière évaluation <sN>. Si la valeur de <s1> est différente de NIL, UNLESS retourne NIL. UNLESS permet de construire une structure de contrôle de type :

si <s1> alors NIL sinon <s2> ... <sN>

UNLESS peut être défini en Lisp de la manière suivante :

```
(DF UNLESS (l)
    (IF (EVAL (CAR l))
        NIL
        (EPROGN (CDR l))))

ex : (UNLESS T 1 2 3)    ->  NIL
     (UNLESS NIL 1 2 3)  ->  3
```

Compatibilité : cette fonction n'existe que dans les post-Maclisp Lisps.

(OR <s1> ... <sN>) [FSUBR]

évalue successivement les différentes expressions <s1> ... <sN> jusqu'à ce que l'une de ces évaluations ait une valeur différente de NIL. OR retourne cette valeur.

OR peut être défini en Lisp de la manière suivante :

```
(DF OR (l)
    (LET-NAMED OR-AUX ((l l))
        (IF (NULL (CDR l))
```

```
                    (E VAL (CAR 1))
                    (LET ((resul (E VAL (CAR 1))))
                         (IF resul
                             resul
                             (OR-AUX (CDR 1)))))))
```

```
ex : (OR NIL NIL 2 3)    ->  2
```

(AND <s1> ... <sN>) [FSUBR]

évalue successivement les différentes expressions <s1> ... <sN> jusqu'à ce que la valeur d'une évaluation soit égale à NIL, à ce moment AND retourne NIL sinon AND retourne la valeur de la dernière évaluation <sN>. Si aucune expression n'est fournie, cette fonction retourne T. AND permet de construire une structure de contrôle de type :

si <s1> alors si <s2> alors ... <sN>

AND peut être défini en Lisp de la manière suivante :

```
(DF AND (l)
    (IF (NULL l)
        T
        (LET-NAMED AND-AUX ((l l))
             (IF (NULL (CDR l))
                 (E VAL (CAR l))
                 (IF (E VAL (CAR l))
                     (AND-AUX (CDR l))
                     NIL)))))
```

```
ex : (AND)           ->  T
     (AND NIL)       ->  NIL
     (AND 1 2 3 4)   ->  4
     (AND 1 2 () 4)  ->  NIL
```

(COND <l1> ... <lN>) [FSUBR]

est la fonction conditionnelle la plus ancienne, la plus compatible et la plus générale de Lisp. Les différents arguments <l1> ... <lN> sont des listes appelées clauses qui ont la structure suivante :

(<ss> <s1> ... <sN>)

COND va sélecter une seule de ces clauses : celle dont la valeur de l'évaluation de son premier élément <ss> est différente de NIL. COND évalue alors en séquence les différentes expressions <s1> ... <sN> de la clause sélectée et retourne la valeur de la dernière évaluation <sN>. Si la clause sélectée n'a qu'un élément <ss>, COND retourne la valeur de l'évaluation de <ss> (i.e. la valeur qui a déclenché la sélection de cette clause). COND permet de construire des structures de contrôle de type :

si ... alors ... sinon si ... alors ... ...

Si aucune clause n'est sélectée, COND retourne NIL. Pour forcer la sélection de la

dernière clause, il est d'usage d'utiliser comme sélecteur l'atome T (dont la valeur est toujours différente de NIL).

COND peut être défini en Lisp de la manière suivante :

```
(DF COND (l)
    (LET-NAMED COND-AUX ((l l))
        (IF (NULL l)
            ()
            (LET ((select (E VAL (CAAR l))))
                (IF select
                    (IF (CDAR l)
                        (EPROGN (CDAR l))
                        select)
                    (COND-AUX (CDR l))))))))
```

```
; donc :                    est equivalent a :
      (COND                                (COND
        (p1 e11 e12 e13)                     (p1 (PRO!
GN e11 e12 e13))
        (p2 e21 e22)                         (p2 (PROGN e21 e22))
        (p3)                                 ((SETQ aux p3) aux)
        (p4 e41))                            (p4 e41))

ex : (COND (NIL 1 2) (T 3 4 5))   ->  5

    (COND ((ATOM X) 'ATOM)
          ((LISTP (CAR X))  'LIST)
          ((NULL (CDR X)) 'NULL)
          (T 'What? ?))
```

(SELECTQ <s> <l1> ... <lN>) [FSUBR]

comme pour la fonction COND, SELECTQ va sélecter une des clauses <l1> ... <lN>. Le sélecteur de ces clauses est la valeur de l'évaluation de <s>, la sélection s'effectue par comparaison du sélecteur avec :
- le CAR (non évalué) de la clause si celui-ci est un symbole (en utilisant le prédicat EQ).
- les différents éléments du sélecteur si celui-ci est une liste (en utilisant la fonction de recherche MEMBER). Cette dernière possibilité permet donc de sélecter des objets de n'importe quel type et ce en nombre quelconque.

Dès qu'une clause est sélectée, SELECTQ évalue en séquence le reste de la clause et retourne la valeur de la dernière évaluation.
Si aucune des clauses <l1> ... <lN> n'est sélectée, SELECTQ retourne NIL.
SELECTQ permet donc de construire des aiguillages sur valeurs constantes.
Comme pour la fonction COND, il est possible de forcer la sélection de la dernière clause en utilisant l'atome T en position sélecteur.

SELECTQ peut être défini en Lisp de la manière suivante :

```
(DF SELECTQ (L)
    (LET-NAMED SEL-AUX ((sel (E VAL (CAR L))) (cl (CDR L)))
```

```
        ; sel = le selecteur evalue, cl = les clauses non evaluees
        (COND
            ((NULL cl)
               ; plus de clauses
               ())
            ((EQ (CAAR cl) T)
               ; selection de la clause toujours vraie
               (EPROGN (CDAR cl)))
            ((AND (SYMBOLP (CAAR cl)) (EQ sel (CAAR  cl)))
               ; selection simple
               (EPROGN (CDAR cl)))
           ((AND (LISTP (CAAR cl)) (MEMBER sel (CAAR  cl)))
               ; selection generale
               (EPROGN (CDAR cl)))
           (T (SEL-AUX sel (CDR cl))))))

ex : (SELECTQ 'ROUGE
        (VERT 'ESPOIR)
        (ROUGE 'OK)
        (T 'NON))
     ->  OK


     (SELECTQ 'ROUGE
        ((BLEU VERT ROUGE) 'COULEUR)
        ((ROSE IRIS) 'FLEUR)
        (T 'SAIS-PAS))
     ->  COULEUR
```

Compatibilité : en Vlisp et en Interlisp, la dernière clause est toujours sélectée. Maclisp quant à lui utilise le prédicat MEMQ pour tester les sélecteurs multiples.

(WHILE <s> <s1> ... <sN>) [FSUBR]

tant que la valeur de l'évaluation de <s> est différente de NIL, WHILE va évaluer en séquence les différentes expressions <s1> ... <sN>. WHILE retourne toujours NIL en valeur (qui est la dernière évaluation de <s> qui fait sortir de la boucle WHILE). Cette fonction permet de construire des boucles conditionnelles d'une manière fort commode, ainsi que des boucles infinies en utilisant la forme : (WHILE T ...) car l'atome T possède toujours une valeur différente de NIL.

WHILE peut être défini en Lisp de la manière suivante :

```
(DF WHILE (l)
    (LET-NAMED WHILE-AUX ()
         (IF (NULL (EVAL (CAR l)))
             ()
             (EPROGN (CDR l))
             (WHILE-AUX))))

; WHILE est equivalent a la forme Lisp :
```

```
     ((LAMBDA-NAMED FOO () (IFN Is> NIL Isl> ... IsN> (FOO))) ())

; ou bien

     (LET-NAMED FOO () (IFN Is> NIL Isl> ... IsN> (FOO)))


ex : (SETQ S '(A B C D))                    ->  (A B C D)
     (WHILE S (PRIN (NEXTL S))) ABCD        ->  NIL
```

(UNTIL <s> <s1> ... <sN>) [FSUBR]

tant que la valeur de l'évaluation de <s> est égale à NIL, UNTIL va évaluer en séquence les différentes expressions <s1> ... <sN>. UNTIL retourne la valeur de la 1ère évaluation de <s> différente de NIL (celle qui fait sortir de la boucle UNTIL). Comme la fonction précédente, UNTIL permet de construire des boucles conditionnelles d'une manière fort commode, ainsi que des boucles infinies en utilisant la forme : (UNTIL () ...).

UNTIL peut être défini en Lisp de la manière suivante :

```
(DF UNTIL (l)
    (LET-NAMED UNTIL-AUX ()
         (OR (EVAL (CAR l))
             (PROGN (EPROGN (CDR l)) (UNTIL-AUX)))))

; UNTIL est equivalent a la forme Lisp :

     ((LAMBDA-NAMED FOO () (OR Is> (PROGN Isl> ... IsN> (FOO))))

; ou bien (a la valeur retournee pres) :

     (WHILE (NOT Is>) Isl> ... IsN>)

ex : (SETQ S '(A B C D))                        ->  (A B C D)
     (UNTIL (NULL S) (PRIN (NEXTL S))) ABCD     ->  T
```

(REPEAT <n> <s1> ... <sN>) [FSUBR]

évalue l'expression <n>. Cette valeur doit être un nombre <n>. REPEAT évalue alors <n> fois les différentes expressions <s1> ... <sN>. Si l'évaluation de <n> n'est pas un nombre strictement positif, la boucle n'est pas exécutée. Dans tous les cas REPEAT retourne le nombre <n>.

REPEAT peut être défini en Lisp de la manière suivante :

```
(DF REPEAT (l)
    (LET ((n (EVAL (CAR l))) (l (CDR l)))
         (WHILE (>= n 0) (EPROGN l) (DECR n)))
    n)
```

```
ex : (REPEAT 10 (PRIN '*)) * * * * * * * * * *   ->  10
     (REPEAT 0 (PRIN '*))  ->  0
```

Compatibilité : cette fonction n'existe pas en Maclisp et s'appelle RPTQ en Interlisp.

3.4.2 Les fonctions de contrôle de type PROG

Les fonctions de type PROG (lourd héritage de Lisp 1.5) permettent d'écrire des suites d'appels séquentiels à la manière ALGOL ou FORTRAN, de se brancher à des étiquettes (avec la fonction GO) et de retourner une valeur (avec la fonction RETURN). Bien évidemment ces 3 fonctions ne sont là que pour des raisons de compatibilité et de transport. Evitez de les utiliser, vos programmes en seront d'autant plus lisibles, plus facilement transportés et mieux compilés.

(PROG <l> <s1> ... <sN>) [FSUBR]

<l> est une liste de symboles faisant office de variables locales à l'intérieur du PROG (très longtemps il n'y a eu que cette fonction dans les systèmes Lisp pour déclarer des variables locales, ce qui explique une grande partie du succès du PROG). Toutes ces variables vont être liées à la valeur NIL à l'entrée du PROG. Cette liste peut être vide mais ne peut pas être omise. Une fois les variables locales liées, PROG va évaluer séquentiellement les expressions <s1> ... <sN>. Si dans cette liste se trouvent des symboles, ils sont considérés comme des étiquettes du PROG et ne sont pas évalués. La valeur d'un PROG s'il n'est pas interrompu par un appel de la fonction RETURN est la valeur retournée par la dernière évaluation, i.e. la valeur de <sN>.

(RETURN <s>) [SUBR à 1 argument]

si cette fonction est évaluée à l'intérieur d'un PROG, elle provoque l'abandon de celui-ci et retourne la valeur <s> comme valeur du PROG. Si cette fonction est utilisée à l'extérieur d'un PROG, une erreur est déclenchée.

(GO <sym>) [FSUBR]

<sym> est le nom d'une étiquette du PROG courant et permet de reprendre l'évaluation séquentielle à partir de celle-ci. Si <sym> n'est pas une étiquette du PROG courant, une erreur est déclenchée.

```
; Voici un exemple de l'horreur que l'on peut atteindre en utili!
sant le
; PROG, cet exemple est adapte du Manuel Spice Lisp de G. Steele :

(de horreur (w)
    (prog (x y z)
          (setq y (car w) z (cdr w))
    loop  (cond ((null y) (return x))
```

```
            ((null z) (go err)))
join  (setq x (cons (cons (car y) (car z)) x))
      (setq y (cdr y) z (cdr z))
      (go loop)
err   (error "Caramba  encore rate.")
      (setq z y)
      (go join)))
```

Compatibilité : Maclisp et certaines versions de Vlisp autorisent les variables d'étiquettes (fonction GOTO) et le branchement à des étiquettes de PROG englobant.

3.4.3 La fonction d'itération de type FOR

(FOR (<var> <init> <step> <until>) <s1> ... <sN>) **[FSUBR]**

permet aux nostalgiques (dont je suis) de réaliser la bonne vieille boucle Algol. <var> doit être un symbole et est la variable numérique de contrôle de la boucle, <init> est la valeur numérique initiale de cette variable en entrant dans la boucle <step> est l'incrément numérique à chaque passage dans la boucle et <until> est la valeur maximum (ou minimum dépendant du signe de <step>) que peut prendre la variable de contrôle. FOR retourne la valeur de la dernière évaluation du corps <s1> ... <sN>.

FOR peut être défini en Lisp de la manière suivante :

```
(DEFMACRO FOR ((var init step until) . body)
  (LET ((*for-result* ())
        (*for-step*   ,step)
        (*for-until*  ,until))
       (COND ((= *for-step* 0)
              (SYSERROR 'FOR "increment nul" *for-step*))
             ((> *for-step* 0)
              (WHILE ((IF (> *for-step* 0) '< '>) ,var ,until))
                     (SETQ *for-result* (PROGN ,@body))
                     (INCR ,var *for-step*))))
        *for-result*))))

ex :  (FOR (i 0 1 9) (PRIN i))  0123456789 -> 9
```

3.4.4 La fonction d'itération de type DO

(DO <lv> <lr> <s1> ... <sN>) [FSUBR]

cette fonction, empruntée à Maclisp, est une fonction très générale d'itération possédant une ou plusieurs variables de contrôle mais possède à notre avis les mêmes défauts que les fonctions de type PROG.

<lv> est une liste de triplets chacun d'eux décrivant une variable de contrôle de la manière suivante :

(<sym> <val> <inc>)

<sym> est le nom de la variable de contrôle

<val> est l'expression de sa valeur initiale (avant de renter dans la boucle) par défaut cette valeur est NIL.

<inc> est l'expression de sa valeur à chaque itération. Si cette expression n'est pas fournie, la variable ne changera pas de valeur pendant les itérations.

<lr>, le second argument d'un DO est une liste de la forme :

(<test> <e1> ... <eN>)

dans laquelle <test> est l'expression de l'arrêt de l'itération et <e1> ... <eN> la valeur à retourner en cas de fin d'itération. Cet argument est analogue à une clause de COND. Si la valeur de l'évaluation de <test> est égale à NIL, le corps du DO est exécuté dans le cas contraire les expressions <e1> ... <eN> sont évaluées et <eN> devient la valeur de sortie du DO.

<s1> ... <sN> est le corps du DO et se comporte exactement de la même manière que le corps du PROG (i.e. il peut comporter des étiquettes et réaliser des GO ou des RETURN).

L'évaluation d'un DO procède comme suit :

1) les différentes expressions <val1> ... <valN> des triplets sont évaluées et sont utilisées pour lier les variables de contrôle <sym1> ... <symN> à la manière d'un LET ou d'un PSETQ (i.e. en parallèle)

2) puis le test <test> est évalué. Si sa valeur est différente de NIL, les expressions <e1> ... <eN> sont évaluées et DO retourne la valeur de <eN>.

3) si le test a retourné NIL, le corps du DO <s1> ... <sN>
est exécuté à la manière d'un PROG.

4) arrivé à la fin du corps du DO, les différentes expressions <step1> ... <stepN> des triplets sont évaluées et sont utilisées (si elles existent) pour lier les variables de contrôle <sym1> ... <symN>. Puis le processus reprend en 2).

```
; Voici quelques exemples d'utilisation du DO sans corps :

(de length (l)
    (do ((x l (cdr x))
```

```
          (j 0 (1+ j)))
         ((atom l) j)))

(de reverse (l)
    (do ((x l (cdr x))
         (y '() (cons(car x) y)))
        ((atom x) y)))
```

3.4.5 Les fonctions de contrôle non locales

Les fonctions de contrôle qui vont suivre sont très puissantes et très rapides à la fois à la compilation et à l'interprétation. Leur usage est nécessaire pour tout ce qui est structure de contrôle un peu sophistiquée, et vivement recommandé. On appelle échappement un point de retour défini dynamiquement. Le_Lisp associe à ces échappements des symboles. Ces associations ne mettent en jeu ni la valeur de ces symboles, ni les fonctions associées à ces symboles.

Compatibilité : les échappements du système Le_Lisp sont complètement compatibles avec les CATCH, THROW, CATCH et THROW* de Maclisp. En revanche ils ne possèdent pas l'aspect 'fonctionnel' des échappements de Vlisp (fonction ESCAPE). Le_Lisp a préféré changer complètement le nom de ces fonctions vu la non standardisation des noms des fonctions d'échappement des post-Maclisp Lisps.*

(TAG <sym> <s1> ... <sN>) **[FSUBR]**

permet de définir un échappement de nom <sym>. Ce nom n'est pas évalué. Puis les différentes expressions <s1> ... <sN> sont évaluées. Si aucun appel de la fonction EXIT n'est réalisé durant ces évaluations, TAG retourne la valeur de la dernière évaluation (i.e. celle de <sN>). En revanche si au cours de ces évaluations un appel du type (EXIT <sym> <e1> ... <eN>) est rencontré, les évaluations à l'intérieur du TAG sont stoppées, les expressions <e1> ... <eN> sont évaluées et <eN> devient la valeur de retour de la fonction TAG.

```
ex : (DE PRESENT (l e)
         (TAG trouve
              (LET-NAMED AUX ((l))
                   (COND ((NULL l) ())
                         ((EQ l e) (EXIT trouve l))
                         ((CONSP l) (AUX (CAR l)) (AUX (CDR l)))
                         (T ()))))))

     (PRESENT '(1 (2 . 3) 4) 3)    ->  3
     (PRESENT '(1 (2 . 3) 4) 5)    ->  NIL
```

(EVTAG <s> <s1> ... <sN>) [FSUBR]

EVTAG est identique à la fonction précédente mais le nom de l'echappement est évalué. EVTAG permet de définir des échappements calculés.

(EXIT <sym> <s1> ... <sN>) [FSUBR]

permet de sortir de l'échappement de nom <sym> (i.e. retourner au dernier (TAG <sym> ...) ou (EVTAG <sym> ...) défini dynamiquement) après avoir évalué les différentes expressions <s1> ... <sN>. Si <sym> n'est pas le nom d'un échappement, EXIT déclenche une erreur dont le libellé est :

** <fnt> : échappement indéfini : <sym> ou bien
** <fnt> : undefined tag : <sym>

dans lequel <fnt> est la fonction appelée (EXIT ou EVEXIT) et <sym> est le nom de l'échappement.

(EVEXIT <s> <s1> ... <sN>) [FSUBR]

équivalent à EXIT mais le nom de l'échappemet est évalué.

(UNTILEXIT <sym> <e1> ... <eN>) [FSUBR]

permet de réaliser une structure de boucle contrôlée par un échappement. <sym> est le nom d'un échappement. Les expressions <e1> ... <eN> vont être répétées indéfiniment jusqu'à ce qu'un appel explicite de l'echappement <sym> (ou d'un échappement plus globale). UNTILEXIT retourne la valeur ramenée par cet échappement.

```
(UNTILEXIT Isym> Isl> ... IsN>)

; correspond donc a l'appel :

(TAG Isym> (WHILE T Isl> ... IsN>))
```

(CATCH-ALL-BUT <l> <s1> ... <sN>) [FSUBR]

<l> est une liste de noms d'échappements. Si durant l'evaluation des expressions <s1> ... <sN> un appel d'échappement est réalisé (au moyen d'un appel de EXIT ou EVEXIT) qui provoque une sortie du CATCH-ALL-BUT alors cette fonction teste si le nom de l'échappement en question fait partie de la liste de noms <l>. S'il en fait partie l'échappement est réalisé normalement, s'il n'en fait pas partie, l'erreur 'échappement indéfini' est déclenchée. Cette fonction permet donc de 'filter' les échappements.

(UNWIND-PROTECT <s1> <s2> ... <sN>) [FSUBR]

évalue les différentes expressions <s1> ... <sN> et retourne la valeur de la 1ère évaluation (i.e. celle de <s1>). Son comportement dans le cas le plus simple est donc équivalent à un PROG1. Toutefois si au cours de l'évaluation de <s1> un échappement se produit, les expressions <s2> ... <sN> seront quand même évaluées durant le processus de restauration du contexte de sortie de l'échappement. Cette fonction, indispensable pour l'écriture de sous-systèmes Lisp, permet entre autres de restaurer automatiquement des environnements complexes en cas de sorties extraordinaires ou d'erreurs.

3.5 Les prédicats de base

En Lisp la valeur Booléene *fausse* est assimilée à la valeur NIL et la valeur booléenne *vraie* est assimilée à toute valeur différente de NIL (ce qui laisse un très grand choix de représentation ...).
Certains prédicats devant retourner une valeur booléenne *vraie* utiliseront l'atome spécial T (initiale du TRUE anglais) qui par définition possède une valeur différente de NIL (la valeur de l'atome T est l'atome T lui-même).

Cette section ne décrit que les prédicats de base. Les tests de l'arithmétique entière et mixte sont décrits dans les sections suivantes.

(NULL <s>) [SUBR à 1 argument]

teste si <s> est égal à NIL. NULL retourne T si le test est vérifié et NIL dans le cas contraire.

NULL peut être défini en Lisp de la manière suivante :

```
(DE NULL (s)
   (EQ s NIL))

ex : (NULL NIL)   ->  T
     (NULL T)     ->  NIL
```

(NOT <s>) [SUBR à 1 argument]

effectue l'inversion de la valeur booléenne <s>. Du fait de la construction des valeurs booléennes en Lisp, cette fonction est identique à la fonction NULL mais il est souhaitable d'utiliser effectivement NULL pour tester si une liste est vide et NOT pour inverser une valeur logique, vos programmes gagneront en clarté.

(ATOM <s>) [SUBR à 1 argument]

teste si <s> est un atome, i.e. un symbole, un nombre ou une chaîne de caractères. ATOM retourne T si ce test est vérifié, NIL dans le cas contraire.

```
ex : (ATOM ())          -> T
     (ATOM 'ARGH)       -> T
     (ATOM "OUPS")      -> T
     (ATOM 42)          -> T
     (ATOM '(A B))      -> NIL
```

(CONSTANTP <s>) [SUBR à 1 argument]

teste si <s> est une constante, i.e. un objet qui évalué fournit le même objet. Sont des constantes tous les types de nombres, les chaînes de caractères et les symboles NIL et T. CONSTANTP retourne T si le test est vérifié et NIL dans le cas contraire.

```
ex : (CONSTANTP 123)      -> T
     (CONSTANTP "Barf")   -> T
     (CONSTANTP NIL)      -> T
     (CONSTANTP 'ARGH)    -> NIL
     (CONSTANTP '(A B))   -> NIL
```

(SYMBOLP <s>) [SUBR à 1 argument]

teste si <s> est un symbole. SYMBOLP retourne T si le test est vérifié et NIL dans le cas contraire.

```
ex : (SYMBOLP ())         -> T
     (SYMBOLP 'ARGH)      -> T
     (SYMBOLP "OUPS")     -> NIL
     (SYMBOLP 44)         -> NIL
     (SYMBOLP '(A B))     -> NIL
```

Compatibilité : cette fonction s'appelle LITATOM en Vlisp et en Interlisp.

(VARIABLEP <s>) [SUBR à 1 argument]

teste si <s> est un symbole non constant, i.e. différent de NIL ou de T. Retourne T si le test est vérifié et NIL dans le cas contraire.

```
ex : (VARIABLEP 123)      -> NIL
     (VARIABLEP "Barf")   -> NIL
     (VARIABLEP NIL)      -> NIL
     (VARIABLEP 'ARGH)    -> T
     (VARIABLEP '(A B))   -> NIL
```

(NUMBERP <s>) [SUBR à 1 argument]

teste si <s> est un nombre. NUMBERP retourne le nombre <s> si le test est vérifié et NIL dans le cas contraire. Les tests des différents types de nombres sont décrits dans la section sur les prédicats numériques.

```
ex : (NUMBERP ())        ->  NIL
     (NUMBERP 'ARGH)     ->  NIL
     (NUMBERP "OUPS")    ->  NIL
     (NUMBERP 44)        ->  44
     (NUMBERP '(A B))    ->  NIL
```

Compatibilité : cette fonction s'appelle NUMBP en Vlisp.

(STRINGP <s>) [SUBR à 1 argument]

teste si <s> est une chaîne de caractères. STRINGP retourne <s> si le test est vérifié et NIL dans le cas contraire.

```
ex : (STRINGP ())        ->  NIL
     (STRINGP 'ARGH)     ->  NIL
     (STRINGP "OUPS")    ->  "OUPS"
     (STRINGP 44)        ->  NIL
     (STRINGP '(A B))    ->  NIL
```

(CONSP <s>) [SUBR à 1 argument]

teste si <s> est une liste non vide. CONSP retourne <s> si le test est vérifié et NIL dans le cas contraire. CONSP est le prédicat inverse de ATOM.

```
ex : (CONSP ())      ->  NIL
     (CONSP 'ARGH)   ->  NIL
     (CONSP "OUPS")  ->  NIL
     (CONSP 44)      ->  NIL
     (CONSP '(A B))  ->  (A B)
```

(LISTP <s>) [SUBR à 1 argument]

teste si <s> est une liste qui peut être vide (i.e. égale au symbole NIL). LISTP retourne l'atome T si le test est vérifié et NIL dans le cas contraire. La représentation de ls liste vide sous la forme du symbole NIL pose le problème du status de ce symbole : NIL est à la fois un symbole et la représentation des listes vides. Le_Lisp, comme la plupart des post-maclisp Lisps, résout cette ambiguité en introduisant deux fonctions testant les listes.

```
ex : (LISTP NIL)     ->  T
     (LISTP 'ARGH)   ->  NIL
     (LISTP "OUPS")  ->  NIL
     (LISTP 44)      ->  NIL
     (LISTP '(A B))  ->  T
```

Compatibilité : la version Interlisp et Vlisp de cette fonction correspond à CONSP.

(EQ <s1> <s2>) [SUBR à 2 arguments]

sert à tester 2 symboles. EQ retourne T si les 2 symboles <s1> et <s2> sont égaux et NIL s'ils ne le sont pas. Dans le cas où les arguments ne seraient pas des symboles, la fonction EQ va tester l'égalité des représentations internes des arguments <s1> et <s2> (EQ teste si les deux arguments ont la même adresse physique).
La représentation interne des nombres et des chaînes de caractères variant en fonction des différentes implantations du système, il est recommandé d'utiliser la fonction = (qui est décrite avec les prédicats numériques) pour tester l'égalité des valeurs numériques et la fonction EQSTRING pour tester l'égalité des chaînes.

```
ex : (EQ 'A (CAR '(A)))        -> T
     (EQ "STRR" "STRR")        -> T ou NIL en fonction des implanta!
tions
     (EQ (1+ 119) 120)         -> T ou NIL en fonction des implanta!
tions
     (EQ '(A B) '(A B))        -> NIL
     (SETQ L '(x y))           -> (x y)
     (EQ L L)                  -> T
```

(NEQ <s1> <s2>) [SUBR à 2 arguments]

est équivalent à (NOT (EQ <s1> <s2>)).

NEQ peut être défini en Lisp de la manière suivante :

```
(DE NEQ (s1 s2)
    (IF (EQ s1 s2) NIL T))
```

(EQUAL <s1> <s2>) [SUBR à 2 arguments]

est la fonction de comparaison la plus générale et doit être utilisée dès que le type des objets à comparer n'est pas précisément connu. Si <s1> et <s2> sont des symboles, EQUAL est identique à la fonction EQ. Si <s1> et <s2> sont des nombres, EQUAL est identique à la fonction =, si <s1> et <s2> sont des chaînes de caractères, EQUAL est identique à la fonction EQSTRING, sinon si <s1> et <s2> sont des listes, EQUAL teste si elles possèdent la même structure (i.e. si les listes possèdent les mêmes éléments). Si le test est verifié, EQUAL retourne cette structure dans le cas contraire EQUAL retourne NIL.

EQUAL peut être défini en Lisp de la manière suivante :

```
(DE EQUAL (l1 l2)
    (COND ((SYMBOLP l1)
           (IF (SYMBOLP l2) (EQ l1 l2) ()))
          ((NUMBERP l1)
           (IF (NUMBERP l2) (= l1 l2) ()))
          ((STRINGP l1)
           (IF (STRINGP l2) (EQSTRING l1 l2) ()))
          ((ATOM l2) ())
          ((AND (EQUAL (CAR l1) (CAR l2))
                (EQUAL (CDR l1) (CDR l2))
                T))))
```

```
ex : (EQUAL "Foo bar" "Foo bar")            ->  T
     (EQUAL 1214 (1+ 1213))                 ->  1214
     (EQUAL '(A (B . C) D) '(A (B . C) D))  ->  (A (B . C) D)
     (EQUAL '(A B C) '(A B C D))            ->  NIL
```

(NEQUAL <s1> <s2>) [SUBR à 2 arguments]

est équivalent à (NOT (EQUAL <s1> <s2>)).

NEQUAL peut être défini en Lisp de la manière suivante :

```
(DE NEQUAL (s1 s2)
   (IF (EQUAL s1 s2) NIL T))
```

3.6 Les fonctions sur les listes

3.6.1 Les fonctions de recherche dans les listes

toutes ces fonctions acceptent comme argument <▷ une liste ou bien l'atome NIL. Dans tous les autres cas une erreur de type est déclenchée.

(CAR <▷) [SUBR à 1 argument]

si <▷ est une liste, retourne son premier élément. Si <▷ égal NIL, CAR retourne NIL. Le CAR d'un symbole autre que NIL, d'un nombre ou d'une chaîne de caractères est indéterminé et provoque de ce fait une erreur.

```
ex : (CAR '(A B C))     ->  A
     (SETQ X '(U P))    ->  (U P)
     (CAR X)            ->  U
     (CAR 'X)           ->  ** CAR : l'argument n'est pas une!
liste : X
```

(CDR <▷) [SUBR à 1 argument]

si <▷ est une liste, retourne cette liste sans son premier élément. Le CDR de NIL est NIL par définition. Le CDR d'un symbole autre que NIL, d'un nombre ou d'une chaîne de caractères est indéterminé et provoque également une erreur.

```
ex : (CDR '(A B C))     ->  (B C)
     (CDR '(X . Y))     ->  Y
     (CDR NIL)          ->  NIL
```

(C....R <▷) [SUBR à 1 argument]

les 28 combinaisons de CAR et de CDR imbriqués (jusqu'à 4 niveaux) sont disponibles.

(CADR <▷) est équivalent à (CAR (CDR <▷))
(CDAAR <▷) est équivalent à (CDR (CAR (CAR <▷)))
(CADAAR <▷) est équivalent à (CAR (CDR (CAR (CAR <▷))))

(MEMQ <sym> <▷) [SUBR à 2 arguments]

si le symbole <sym> est un élément de la liste <▷, retourne la partie de la liste <▷ commençant au symbole <sym>, sinon retourne NIL. Cette fonction utilise le prédicat EQ pour tester la présence du symbole <sym> dans la liste <▷. On utilise souvent cette fonction pour simplement tester la présence d'un élément de liste dans une liste donnée.

MEMQ peut être défini en Lisp de la manière suivante :

```
(DE MEMQ (at l)
    (IF (OR (ATOM l) (EQ at (CAR l)))
        l
        (MEMQ at (CDR l))))

ex : (MEMQ 'C '(A B C D E))   ->  (C D E)
     (MEMQ 'Z '(A B C D E))   ->  NIL
```

(MEMBER <s> <▷) [SUBR à 2 arguments]

si l'expression <s> est un élément de la liste <▷, retourne la partie de la liste <▷ commençant à l'élément <s>, sinon retourne NIL. Cette fonction utilise le prédicat EQUAL et permet donc de rechercher un élément de n'importe quel type dans une liste.

MEMBER peut être défini en Lisp de la manière suivante :

```
(DE MEMBER (s l)
    (IF (OR (ATOM l) (EQUAL s (CAR l)))
        l
        (MEMBER s (CDR l))))

ex : (MEMBER 'C '(A B C D E))        ->  (C D E)
     (MEMBER 'Z '(A B C D E))        ->  NIL
     (MEMBER '(A B) '(A (A B) C))    ->  ((A B) C)
```

(NTHCDR <n> <l>) [SUBR à 2 arguments]

retourne la partie de la liste <l> commençant à son <n>ième CDR. Si <l> n'est pas une liste ou si (LENGTH <l>) est plus petit que <n>, NTH retourne NIL. Si <n> ⩽0, NTHCDR retourne la liste <l> en entier.

NTHCDR peut être défini en Lisp de la manière suivante :

```
(DE NTHCDR (n l)
    (IF (<= n 0)
        l
        (NTHCDR (1- n) (CDR l))))
```

```
ex : (NTHCDR 3 '(A B C D E))  ->  (D E)
```

Compatibilité : Vlisp et Interlisp appellent cette fonction NTH et commencent à compter les éléments à partir de 0. Le nom NTH est en conflit avec la fonction suivante.

(NTH <n> <l>) [SUBR à 2 arguments]

retourne le <n>ième élément de la liste <l> le CAR de la liste étant l'élément de numéro 0. NTH est équivalent à :
(CAR (NTHCDR <n> <l>)).

NTH peut être défini en Lisp de la manière suivante :

```
(DE NTH (n l)
    (IF (<= n 0)
        (CAR l)
        (NTH (1- n) (CDR l))))
```

```
ex : (NTH 3 '(A B C D E F))     ->  D
     (NTH 10 '(A B C))          ->  NIL
```

Compatibilité : cette fonction s'appelle CNTH en Vlisp, de plus les éléments sont comptés à partir de 1.

(LAST <s>) [SUBR à 1 argument]

si <s> est une liste, retourne la liste composée de son dernier élément. si <s> n'est pas une liste, retourne <s> lui-même.

LAST peut être défini en Lisp de la manière suivante :

```
(DE LAST (s)
    (IF (OR (ATOM s) (ATOM (CDR s)))
        s
        (LAST (CDR s))))
```

```
ex : (LAST '(A B C D E))     ->  (E)
     (LAST '(A B C . D))     ->  (C . D)
     (LAST 120)              ->  120
```

(LENGTH <s>) **[SUBR à 1 argument]**

si <s> est une liste, retourne le nombre d'éléments de cette liste. Si <s> n'est pas une liste (donc si <s> est un atome), LENGTH retourne 0

LENGTH peut être défini en Lisp de la manière suivante :

```
(DE LENGTH (l)
    (IF (ATOM l)
        0
        (1+ (LENGTH (CDR l)))))

ex : (LENGTH '(A (B C) D E )  ->  4
     (LENGTH "Esar")          ->  0
```

3.6.2 Les fonctions de création de listes

Toutes les fonctions qui vont être décrites fabriquent de nouvelles listes. La gestion de la mémoire allouée aux listes est dynamique et automatique (voir la section sur le Garbage-collector).

(CONS <s1> <s2>) **[SUBR à 2 arguments]**

construit une liste dont le premier élément est <s1> et le reste la liste <s2>. Si <s2> est un atome, CONS produit la paire pointée

(<s1> . <s2>)

```
; En supposant que la variable FREE contient la liste des cellules
; allouables, CONS se decrit en Lisp :

     (DE CONS (x y)
         (RPLACA FREE x)
         (RPLACD (PROG1 FREE (SETQ FREE (CDR FREE))) y))

ex : (CONS 'A '(B C))        ->  (A B C)
     (CONS 1 'X)             ->  (1 . X)
     (SETQ 1 '(X Y Z))       ->  (X Y Z)
     (EQUAL (CONS (CAR 1) (CDR 1))
         1)                  ->  T

; Trouvee par D. Goossens,
; voyez-vous ce que construit cette fonction ?

     (DE FOO (1 x)
         (IF (ATOM 1)
```

```
                (CONS 1 x)
                (FOO (CAR 1) (IF (CDR 1) (FOO (CDR 1) x) x))))
```

exemple d'appel :

```
(FOO '(A (B . C) ((D))) NIL)
```

(XCONS <s1> <s2>) [SUBR à 2 arguments]

cette fonction est équivalente à :

```
(CONS s2 s1)
```

(NCONS <s>) [SUBR à 1 argument]

cette fonction correspond à :

```
(CONS Is> NIL)
```

(MCONS <s1> ... <sN>) [SUBR à N arguments]

réalise un CONS multiple. L'appel de :
(MCONS s1 s2 ...: sN-1 sN)
équivaut à l'appel
(CONS s1 (CONS s2 ... (CONS sN-1 sN) ...))

MCONS peut être défini en Lisp de la manière suivante :

```
; sous forme d'une FEXPR :

(DF MCONS (l)
    (CONS (EVAL (CAR l))
          (EVAL (IF (NULL (CDDR l))
                    (CADR l)
                    (CONS 'MCONS (CDR l))))))

; ou bien sous forme d'une MACRO

(DM MCONS (l)
    (LIST 'CONS (CADR l)
          (IF (NULL (CDDDR l))
              (CADDR l)
              (CONS 'MCONS (CDDR l)))))


ex : (MCONS 'A 'B)      ->  (A . B)
     (MCONS 'A 'B 'C)   ->  (A B . C)
```

Compatibilité : cette fonction s'appelle LIST en Maclisp et CONS en Interlisp, fonction qui est une NSUBR.*

(LIST <s1> ... <sN>) **[SUBR à N arguments]**

retourne la liste des valeurs des différentes expressions <s1> ... <sN>.
En terme de CONS l'appel (LIST <s1> <s2> ... <sN-1> <sN>) est équivalent à
(CONS <s1> (CONS <s2> ... (CONS <sN-1> (CONS <sN> NIL)) ...)).

LIST peut être défini en Lisp de la manière suivante :

```
; sous forme d'EXPR :

     (DE LIST l l)

; ou sous forme de FEXPR :

     (DF LIST (l)
         (LET ((r))
              (WHILE l (NEWL r (EVAL (NEXTL l))))
              r))

; ou sous forme de MACRO :

     (DM LIST (l)
         (IF (NULL (CDR l))
             NIL
             (DISPLACE l
                   (CONS 'CONS
                         (CONS (CADR l)
                               (IF (NULL (CDDR L))
                                   NIL
                                   (CONS (CONS 'LIST (CDDR l)))))))))

ex : (LIST 'A 'B 'C)   ->  (A B C)
     (LIST)            ->  NIL
```

(KWOTE <s>) **[SUBR à 1 argument]**

construit une liste de 2 arguments dont le 1er est l'atome QUOTE lui-même et le second la valeur de <s>.

KWOTE peut être défini en Lisp de la manière suivante :

```
    (DE KWOTE (s)
              (LIST 'QUOTE s))

ex : (KWOTE 'A)              -> 'A
     (KWOTE (CDR '(A . B))) ->  'B
```

(MAKE-LIST <n> <s>) **[SUBR à 2 arguments]**

retourne une liste de <n> éléments. Chacun de ces éléments contient la valeur <s>. Bien utile pour construire des listes de longueur donnée.

MAKE-LIST peut être défini en Lisp de la manière suivante :

```
(DE MAKE-LIST (n s)
    (IF (<= n 0)
        ()
        (CONS s (MAKE-LIST (1- n) s))))
```

```
ex : (MAKE-LIST 4 'A)   ->  (A A A A)
```

(APPEND <l1> ... <lN>) **[SUBR à N arguments]**

retourne la concaténation d'une copie du premier niveau de toutes les listes <l1> ... <lN-1> à la liste <lN>. S'il n'y a qu'un argument <l1>, retourne simplement une copie du premier niveau de <l1>.

APPEND peut être défini en Lisp de la manière suivante :

```
(DE APPEND l
    (APPEND2 (CAR l)
       (IF (NULL (CDDR l))
           (CADR l)
           (APPLY 'APPEND (CDR l)))))
```

```
; avec APPEND2 :
```

```
(DE APPEND2 (l1 l2)
    (IF (NULL l1)
        l2
        (CONS (CAR l1) (APPEND (CDR l1) l2))))
```

```
ex : (APPEND '(A B C))          ->  (A B C)
     (APPEND '(A B C) '(X Y))   ->  (A B C X Y)
     (APPEND () '(X Y))         ->  (X Y)
```

Compatibilité : Vlisp ne possède qu'une fonction APPEND à 2 arguments.

(REVERSE <s>) **[SUBR à 1 argument]**

retourne une copie inversée du premier niveau de la liste <s>.

REVERSE peut être défini en Lisp de la manière suivante :

```
(DE REVERSE (s)
    (LET-NAMED REV-AUX ((s s) (r))
         (IF (ATOM s)
             r
             (REVERSE (CDR s) (CONS (CAR s) r)))))
```

```
ex : (RE VERSE '(A (B C) D))          ->  (D (B C) A)

; On pretend parfois que cette fonction inverse aussi la liste l :

     (DE RE V (l)
         (IF (NULL (CDR l))
             l
             (CONS (CAR (RE V (CDR l)))
                   (RE V (CONS (CAR l)
                               (RE V (CDR (RE V (CDR l))))))))))
```

Compatibilité : Vlisp possède une fonction REVERSE à 2 arguments correspondant à : (NCONC (REVERSE s1) s2).

(COPY <l>) [SUBR à 1 argument]

fabrique une nouvelle copie de toute la liste <l>. Pour cette fonction la copie s'effectue à tous les niveaux de l'arborescence.

COPY peut être défini en Lisp de la manière suivante :

```
     (DE COPY (s)
         (IF (ATOM s)
             s
             (CONS (COPY (CAR s)) (COPY (CDR s)))))

ex : (COPY 'A)                  ->  A
     (COPY '(A (B (C (D))))  ->  (A (B (C (D))))

; cette fonction ne traite pas les listes circulaires ou partagees.
; Pour cela il faut utiliser la fonction :

     (DE CIRCOPY (l)
         (LET ((d))
              (LET-NAMED CIRC-AUX ((l l) (new))
                  (COND
                     ((ATOM l) l)
                     ((CDR (ASSQ l d)))
                     (T (SETQ new (CONS NIL NIL)
                              d (CONS (CONS l new) d))
                        (RPLACA new (CIRC-AUX (CAR l)))
                        (RPLACD new (CIRC-AUX (CDR l))))))))
```

(FIRSTN <n> <l>) [SUBR à 2 arguments]

retourne une copie des <n> premiers éléments de le liste <l>. Si la liste <l> possède moins de <n> éléments (i.e. si <n> > (LENGTH <l>)), FIRSTN retourne une copie de toute la liste <l>.

FIRSTN peut être défini en Lisp de la manière suivante :

```
(DE FIRSTN (n l)
    (COND ((NULL l) ())
          ((<= n 0) ())
          (t (CONS (CAR l) (FIRSTN (1- n) (CDR l))))))

ex : (FIRSTN 3 '(A B C D E F))    ->  (A B C)
     (FIRSTN 5 '(A B C D))        ->  (A B C D)
```

(LASTN <n> <l>) **[SUBR à 2 arguments]**

retourne une copie des <n> derniers éléments de la liste <l>. Si la liste contient moins d'eléments que demandé (i.e. si (LENGTH l) < n) LASTN retourne une copie de la liste <l> entière.

LASTN peut être défini en Lisp de la manière suivante :

```
(DE LASTN (n l)
    (RE VERSE (FIRSTN n (RE VERSE l))))

ex : (LASTN 2 '(A B C D E))   ->  (D E)
     (LASTN 10 '(A B C))      ->  (A B C)
```

(SUBST <s1> <s2> <l>) **[SUBR à 3 arguments]**

fabrique une nouvelle copie de toute la liste <l> en substituant à chaque occurrence de l'expression <s2>, l'expression <s1>. Cette fonction utilise le prédicat EQUAL pour réaliser le test et fabrique toujours une nouvelle expression (il faut utiliser la fonction NSUBST pour changer physiquement l'expression).

SUBST peut être défini en Lisp de la manière suivante :

```
(DE SUBST (new old l)
    (COND ((EQUAL l old) new)
          ((ATOM l) l)
          (T (CONS (SUBST new old (CAR l))
                   (SUBST new old (CDR l)))))))

ex : (SUBST '(X Y Z) 'A  '(A C (D A)))
     ->  ((X Y Z) C (D (X Y Z)))
```

(REMQ <sym> <l>) **[SUBR à 2 arguments]**

retourne une copie de <l> dans laquelle toutes les occurrences du symbole <sym> ont été enlevées.

REMQ peut être défini en Lisp de la manière suivante :

```
(DE REMQ (sym l)
    (COND ((NULL l) ())
          ((EQ sym (CAR l)) (REMQ sym (CDR l)))
          (T (CONS (CAR l) (REMQ sym (CDR l))))))
```

```
ex : (REMQ 'A '(A B A (C A B) D A S))   ->  (B (C A B) D S)
```

(REMOVE <s> <l>) **[SUBR à 2 arguments]**

cette fonction est identique à la fonction précédente mais réalise le test au moyen du prédicat EQUAL.

REMOVE peut être défini en Lisp de la manière suivante :

```
(DE REMO VE (s l)
    (COND ((NULL l) ())
          ((EQUAL s (CAR l)) (REMO VE s (CDR l)))
          (T (CONS (CAR l) (REMO VE s (CDR l))))))
```

```
ex : (REMO VE '(A) '(A B (A) (C A B) (A) S))   ->  (A B (C A B) S)
```

Compatibilité : Vlisp appelle DELETE (et la précédente DELQ) ce qui en Maclisp, Interlisp et Le_Lisp correspond à REMOVE et REMQ. De plus Maclisp et Le_Lisp appelent DELETE et DELQ les fonctions qui travaillent physiquement sur les listes (voir ces fonctions).

(OBLIST) **[SUBR à 0 argument]**

retourne la (très longue) liste de tous les symboles présents dans le système. A l'initialisation du système, cette liste contient le nom de toutes les fonctions et variables standards. Cette liste étant très longue il est conseillé d'utiliser la fonction MAPOBLIST pour accéder successivement à tous ces symboles. Il n'est pas possible de définir simplement cette fonction en Lisp car il faut accéder aux propriétés naturelles de type A-LINK de chacun des atomes, ce qui n'est réalisable qu'avec les fonctions LOC et MEMORY.

```
ex :  (OBLIST)  ->

      (RMARGIN LMARGIN OUTPOS PTYPE EXPLODE TERPRI
      PRINCH OBASE READ PEEKCH READCH TEREAD ASCII
      CASCII TYPECH IMPLODE E VAL APPLY EXIT
                        .....
                        .....
                        .....
      MEMORY CALL EXECUTE END)
```

3.6.3 Les fonctions de modification physique

Toutes les fonctions qui vont être décrites dans cette section doivent être utilisées conformément au mode d'emploi, pour éviter de placer le système dans un état de confusion dramatique car elles vont permettre de modifier physiquement les structures Lisp.
En particulier il faut prendre garde à la modification physiques des listes partagées.
Toutefois la possibilité d'une véritable chirurgie sur les représentations internes des listes confère à Lisp la puissance des langages machines.

(RPLACA <l> <s>) **[SUBR à 2 arguments]**

remplace le CAR de la liste <l> par <s>. Retourne la liste ainsi modifiée <l> en valeur. Si l'argument <l> n'est pas une liste, une erreur de type se produit.

```
ex : (RPLACA '(A B C) '(X Y))  ->  ((X Y) B C)
     (RPLACA 'X 'FOO)          ->
     ** RPLACA : l'argument n'est pas une liste : X
```

(RPLACD <l> <s>) **[SUBR à 2 arguments]**

remplace le CDR de la liste <l> par <s>. Retourne la nouvelle liste <l> en valeur. Si l'argument <l> n'est pas une liste, une erreur de type se produit.

```
ex : (RPLACD '(A B C) '(X Y Z))  ->  (A X Y Z)
     (RPLACD NIL 'T)             ->
     ** RPLACD : l'argument n'est pas une liste : NIL
```

(RPLAC <l> <s1> <s2>) **[SUBR à 3 arguments]**

replace le CAR de la liste <l> par <s1> et le CDR de la liste <l> par <s2>. Si l'argument <l> n'est pas une liste, une erreur de type apparait.

RPLAC peut être défini en Lisp de la manière suivante :

```
(DE RPLAC (l s1 s2)
    (IF (ATOM l)
        (SYSERROR 'RPLAC "l'argument n'est pas un CONS" l)
        (RPLACA l s1)
        (RPLACD l s2)))

ex : (SETQ l1 '(A B) l2 l1)     ->  (A B)
     (RPLAC l1 1 '(2))          ->  (1 2)
     l2                         ->  (1 2)
```

Compatibilité : cette fonction s'appelle RPLNODE en Interlisp et n'existe ni en Vlisp ni en Maclisp.

(DISPLACE <l> <ln>) **[SUBR à 2 arguments]**

remplace le CAR de la liste <l> par le CAR de <ln> et le CDR de <l> par le CDR de <ln>. Cette fonction est souvent utilisée dans des MACRO pour modifier physiquement l'appel de MACRO lui-même.

DISPLACE peut être défini en Lisp de la manière suivante :

```
(DE DISPLACE (l ln)
    (RPLACA l (CAR ln))
    (RPLACD l (CDR ln))
    l)
```

```
ex : (SETQ L1 '(A B C))        ->  (A B C)
     (SETQ L2 L1)              ->  (A B C)
     (DISPLACE L1 '(X Y))      ->  (X Y)
     L2                        ->  (X Y)
```

Compatibilité : cette fonction s'appelle RPLACB en Vlisp et RPLNODE2 en Interlisp

(NCONC <l1> ... <lN>) **[SUBR à N arguments]**

concatène physiquement toutes les listes <li> (i.e. place dans le CDR du dernier élément de <l1-1> un pointeur sur la liste <li>). NCONC retourne la nouvelle liste <l1> en valeur. Si deux listes <li> sont les mêmes pointeurs physiques, NCONC permet de construire des listes circulaires par forcage dans le dernier CDR de la liste d'un pointeur sur le 1er élément de cette même liste.

NCONC (à 2 arguments) peut être défini en Lisp de la manière suivante :

```
(DE NCONC2 (l1 l2)
    (LET ((l1 l1))
         (WHILE (CONSP (CDR l1)) (NEXTL l1))
         (RPLACD l1 l2))
    l1)
```

```
ex : (SETQ X1 '(A B C) X2 X1)  ->  (A B C)
     (NCONC X1 '(D E F))       ->  (A B C D E F)
     X2                        ->  (A B C D E F)
     (NCONC X1 X1)             ->  (A B C D E F A B C D E F A B ...
```

(NREVERSE <l>) **[SUBR à 1 argument]**

renverse physiquement et rapidement la liste <l>. Cette fonction doit être manipulée avec précaution car elle modifie physiquement toute la liste ce qui peut provoquer des catastrophes si cette liste était partagée.

NREVERSE peut être défini en Lisp de la manière suivante :

```
(DE NRE VERSE (l)
    (IF (CONSP l)
        (LET-NAMED NR ((l l) (r))
```

```
            (IF (NULL (CDR l))
                (RPLACD l r)
                (NR (CDR l) (RPLACD l r))))
        l))
```

Compatibilité : en Vlisp cette fonction correspond à la fonction FREVERSE à 1 argument et en Interlisp s'appelle DREVERSE.

(NRECONC <l> <s>) [SUBR à 2 arguments]

renverse physiquement (et rapidement) la liste <l> et ajoute au moyen d'un NCONC physique la liste <s>. Cette fontion correspond donc à :

(NCONC (NREVERSE <l>) <s>)

Cette fonction doit etre manipulée avec précaution car elle modifie physiquement des structures Le_Lisp (en particulier les modifications peuvent etre boulversantes en cas de structures partagées), mais elle est évidement beaucoup plus rapide que la fonction traditionnelle REVERSE et REVAPPEND.

NRECONC peut être défini en Lisp de la manière suivante :

```
(DE NRECONC (l s)
    (IF (CONSP l)
        (NCONC (NRE VERSE l) s)
       s))

ex :  (SETQ L1 '(A B C D E)) -> (A B C D E)
      (SETQ L2 (CDR L1))     -> (B C D E)
      (SETQ L3 (LAST L1))    -> (E)
      (NRECONC L1 '(X Y))    -> (E D C B A X Y)
      L1                     -> (A X Y)
      L2                     -> (B A X Y)
      L3                     -> (E D C B A X Y)
```

Compatibilité : cette fonction correspond en Vlisp à la fonction FREVERSE à 2 arguments.

(NSUBST <s1> <s2> <l>) [SUBR à 3 arguments]

modifie physiquement la liste <l> en substituant à chaque occurrence de l'expression <s2>, l'expression <s1>. Cette fonction utilise le prédicat EQUAL pour réaliser le test. Pour obtenir une copie de la liste substituée il faut utiliser la fonction SUBST.

NSUBST peut être défini en Lisp de la manière suivante :

```
(DE NSUBST (new old l)
    (COND ((EQUAL l old) new)
          ((ATOM l) l)
          (T (RPLAC l
                    (NSUBST new old (CAR l))
                    (NSUBST new old (CDR l)))))))
```

```
ex : (SETQ l '(A C (D A)))    ->  (A C (D A))
     (NSUBST '(X Y Z) 'A l)   ->  ((X Y Z) C (D (X Y Z)))
     l                        ->  ((X Y Z) C (D (X Y Z)))
```

(DELQ <sym> <l>) **[SUBR à 2 arguments]**

enlève physiquement toutes les occurrences du symbole <sym> au premier niveau de la liste <l>. Cette fonction utilise le prédicat EQ. Pour réaliser une copie de ce type de liste, il faut utiliser la fonction REMQ.

DELQ peut être défini en Lisp de la manière suivante :

```
(DE DELQ (sym l)
    (COND ((ATOM l) l)
          ((EQ sym (CAR l)) (DELQ sym (CDR l)))
          (T (RPLACD l (DELQ sym (CDR l))))))

ex : (SETQ l '(A B C B B D))  ->  (A B C B B D)
     (DELQ 'B l)              ->  (A C D)
```

(DELETE <s> <l>) **[SUBR à 2 arguments]**

enlève physiquement toutes les occurrences de l'expression <s> au premier niveau de la liste <l>. Cette fonction utilise le prédicat EQUAL. Pour fabriquer une copie de ce type de liste, il faut utiliser la fonction REMOVE.

DELETE peut être défini en Lisp de la manière suivante :

```
(DE DELETE (s l)
    (COND ((ATOM l) l)
          ((EQUAL s (CAR l)) (DELETE s (CDR l)))
          (T (RPLACD l (DELETE s (CDR l))))))

ex : (SETQ l '(A (B) C B (B) D))  ->  (A (B) C B (B) D)
     (DELETE '(B) l)              ->  (A C B D)
```

3.6.4 Les fonctions sur les A-listes

En Lisp, les A-listes (les listes d'association) sont des tables (au sens de SNOBOL 4) qui possèdent la structure suivante :

```
((cle1 . val1) (cle2 . val2)   ...   (cleN . valN))
```

Chaque élément d'une A-liste est un couple constitué d'une clé (le CAR de l'élément de la table) et d'une valeur (le CDR de l'élément). L'accès à une valeur est réalisé au moyen de sa clé.

Pour toutes les fonctions qui vont être décrites, l'argument <al> doit être une A-liste, et les clés doivent être des symboles (pour les fonctions d'accès aux A-listes utilisant le prédicat EQ) ou des S-expressions quelconques (pour les fonctions utilisant le prédicat EQUAL).

(ASSQ <sym> <al>) **[SUBR à 2 arguments]**

retourne l'élément de la A-liste <al> dont la clé (le CAR) est égal au symbole <sym>, sinon ASSQ retourne NIL.

ASSQ peut être défini en Lisp de la manière suivante :

```
(DE ASSQ (at al)
    (COND ((NULL al) NIL)
          ((EQ (CAAR al) at) (CAR al))
          (T (ASSQ at (CDR al)))))
```

```
ex : (ASSQ 'B '((A) (B 1) (C D E)))  ->  (B 1)
```

(CASSQ <sym> <al>) **[SUBR à 2 arguments]**

est identique à ASSQ mais retourne la valeur seule (le CDR) de l'élément dont on précise la clé.

(CASSQ at al) est donc équivalent à (CDR (ASSQ at al))

ATTENTION : on ne peut pas distinguer la valeur NIL associée à un élément de A-liste avec l'absence de cet élément.

```
ex : (CASSQ 'C '((A) (B 1) (C D E)))  ->  (D E)
```

(RASSQ <sym> <al>) **[SUBR à 2 arguments]**

retourne l'élément de la A-liste <al> dont la valeur (le CDR) est égal au symbole <sym>, sinon RASSQ retourne NIL.

RASSQ peut être défini en Lisp de la manière suivante :

```
(DE RASSQ (sym al)
    (COND ((NULL al) NIL)
          ((EQ (CDAR al) at) (CAR al))
          (T (RASSQ sym (CDR al)))))
```

(ASSOC <s> <al>) [SUBR à 2 arguments]

Cette fonction est identique à la fonction ASSQ mais utilise le prédicat EQUAL pour tester les clés qui peuvent donc être de n'importe quel type.

ASSOC peut être défini en Lisp de la manière suivante :

```
(DE ASSOC (s al)
   (COND
      ((NULL al) NIL)
      ((EQUAL s (CAAR al)) (CAR al))
      (T (ASSOC s (CDR al)))))
```

(CASSOC <s> <al>) [SUBR à 2 arguments]

Cette fonction est identique à la fonction CASSQ mais utlise le prédicat EQUAL pour tester les clés qui peuvent donc être de n'importe quel type.

(CASSOC s l) est donc équivalent à (CDR (ASSOC s l))

(RASSOC <s> <al>) [SUBR à 2 arguments]

cette fonction est équivalente à la fonction RASSQ mais utilise le prédicat EQUAL pour tester les valeurs.

RASSOC peut être défini en Lisp de la manière suivante :

```
(DE RASSOC (s al)
    (COND ((NULL al) NIL)
          ((EQUAL s (CDAR al)) (CAR al))
          (T (RASSOC a (CDR al)))))
```

(ACONS <s1> <s2> <l>) [SUBR à 3 arguments]

rajoute un élément à la A-liste <l>. L'élément est composé de la clé <s1> et a pour valeur <s2>. ACONS retourne la nouvelle A-liste ainsi créée.

ACONS peut être défini en Lisp de la manière suivante :

```
(DE ACONS (s1 s2 l)
    (CONS (CONS s1 s2) l))
```

```
ex : (ACON!
S 'A 10 '((B . 11) (Z . 40)))  ->  ((A . 10) (B . 11) (Z . 40))
```

(PAIRLIS <l1> <l2> <al>) [SUBR à 3 arguments]

<l1> doit être une liste de clés
<l2> doit être une liste de valeurs
PAIRLIS retourne une nouvelle A-liste formée à partir de la liste des clés <l1> et de la liste des valeurs associées <l2>. Si le troisième argument <al> est fourni, il est ajouté à la fin de la A-liste créée.

PAIRLIS peut être défini en Lisp de la manière suivante :

```
(DE PAIRLIS (l1 l2 al)
   (IF (NULL l1)
       al
       (CONS (CONS (CAR l1) (CAR l2))
          (PAIRLIS (CDR l1) (CDR l2) al))))
```

```
ex : (PAIRLIS '(YA JESTEM WIELKY)  '(LE MERLE CHANTE) ())
        ->  ((YA . LE) (JESTEM . MERLE) (WIELKY . CHANTE))
     (PAIRLIS '(X Y Z) '(A (B)) '((A . X) (B . Y)))
       ->  ((X . A) (Y B) (Z) (A . X) (B . Y))
```

(SUBLIS <al> <s>) [SUBR à 2 arguments]

retourne une copie de l'expression <s> dans laquelle toutes les occurrences des clés de la A-liste <al> ont été remplacées par leurs valeurs associées correspondantes. Cette fonction effectue une copie entière de l'expression <s> et utilise la fonction ASSQ.

SUBLIS peut être défini en Lisp de la manière suivante :

```
(DE SUBLIS (al s)
    (IF (ATOM s)
        (LET ((x (ASSQ s al))) (IF x (CDR x) s))
        (CONS (SUBLIS al (CAR s)) (SUBLIS al (CDR s))))))
```

```
ex :  (SUBLIS '((A . Z) (B 2 3)) '(A (B A C) D B . B)))
         -> (Z ((2 3) Z C) D (2 3) 2 3)
```

3.7 Les fonctions sur symboles

3.7.1 Fonctions d'accès aux valeurs des symboles

L'accès aux valeurs des symboles est normalement réalisé par EVAL à l'évaluation d'un symbole. EVAL de plus teste si une valeur a été affectée à ce symbole : dans le cas d'un symbole indéfini il provoque une erreur. Il existe de plus deux fonctions particulières d'accès aux symboles.

(BOUNDP <sym>) **[SUBR à 1 argument]**

l'argument <sym> doit être un symbole. BOUNDP teste si ce symbole possède une valeur et retourne T si ce test est vérifié et NIL dans le cas contraire. Cette fonction est utilisée permet d'éviter les erreurs : variable indéfinie. Du fait de la représentation des valeurs indéfinies il n'est pas possible de décrire cette fonction en Lisp.

```
ex : (BOUNDP T)          ->  T
     (BOUNDP NIL)        ->  T
     (BOUNDP 'foofoo)    ->  NIL
                  ; si foofoo n'a pas de valeur
```

(SYMEVAL <sym>) **[SUBR à 1 argument]**

retourne la valeur du symbole <sym>. Cette fonction est donc équivalente à l'appel de EVAL mais est beaucoup plus efficace en particulier pour le compilateur. SYMEVAL a le même comportement que EVAL et provoque donc une erreur si le symbole est indéfini.

```
ex : (SETQ FOO 'BAR)     ->  BAR
     (SYME VAL 'FOO)      ->  FOO
     (SYME VAL 'NIEMA)    ->
     ** SYME VAL : variable indefinie : NIEMA
           ; si NIEMA ne possede pas de valeur
```

3.7.2 Fonctions d'accès aux valeurs des symboles

(MAKUNBOUND <sym>) **[SUBR à 1 argument]**

change la valeur du symbole <sym> de telle manière que tout nouvel accès à sa valeur déclenche l'erreur : variable indéfinie.

(SET <sym> <s>) **[SUBR à 2 arguments]**

change la valeur du symbole <sym> par <s>. SET retourne en valeur <s>.

```
ex : (SETQ X '(A B C))    ->   (A B C)
     (SET 'X '(X Y))      ->   (X Y)
     X                    ->   (X Y)
```

(SETQ <sym1> <s1> ... <symN> <sN>) **[FSUBR]**

<sym1> ... <symN> sont des symboles qui ne sont pas évalués ;
<s1> ... <sN> sont des expressions quelconques qui seront évaluées. SETQ est la fonction d'affectation la plus utilisée : chaque atome <symi> est initialisé avec l'expression correspondante <si>. Ces affectations ainsi que les évaluations des expressions <ei> sont réalisées séquentiellement. SETQ retourne <sN> en valeur.

```
ex : (SETQ L1 '(A B C))     ->  (A B C)
     (SETQ L2 L1)           ->  (A B C)
     (SETQ L3 L2 L4 'FOO)   ->  FOO
     L3                     ->  (A B C)
```

(SETQQ <sym1> <e1> ... <symN> <eN>) **[FSUBR]**

identique à la fonction SETQ mais les valeurs affectées ne sont pas évaluées.

```
ex : (SETQQ A 10 B (X Y Z) C B)    ->  B
     B                             ->  (X Y Z)
     C                             ->  B
```

(PSETQ <sym1> <s1> ... <symN> <sN>) **[FSUBR]**

identique à la fonction SETQ mais les affectations ont lieu en parallèle. Très utile pour réaliser des permutations de variables.

PSETQ peut être défini en Lisp de la manière suivante :

```
(DF PSETQ (l)
    (LET ((lvar) (lval))
         (WHILE l (NEWL lvar (NEXTL l))
                  (NEWL lval (E VAL (NEXTL l))))
         (WHILE lvar (SET (NEXTL lvar) (NEXTL lval)))))
```

(DESET <l1> <l2>) [SUBR à 2 arguments]

réalise le 'destructuring SET' du système *NIL* [White 79]. <l1> est un arbre de variables, <l2> est un arbre de valeurs. DESET va affecter aux variables de l'arbre des variables les valeurs correspondantes de l'arbre de valeurs. DESET retourne toujours T en valeur.

DESET peut être défini en Lisp de la manière suivante :

```
(DE DESET (l1 l2)
    (COND ((NULL l1)
           (OR (NULL l2)
               (SYSERROR 'DESET "trop de valeurs" l2)))
          ((VARIABLEP l1) (SET l1 l2) T)
          ((AND (CONSP l1) (LISTP l2))
           (DESET (CAR l1) (CAR l2))
           (DESET (CDR l1) (CDR l2)))
          (T (SYSERROR 'DESET "liaison impossible"
                       (LIST l1 l2)))))

ex : (DESET '(A (B . C)) '((1 2) (3 4)))    -> T
     A                                      -> (1 2)
     B                                      -> 3
     C                                      -> (4)
```

(DESETQ <l1> <l2>) [FSUBR]

est équivalent à la fonction précédente mais le 1er argument n'est pas évalué (à la SETQ).

```
(DESETQ Il1> I12>)
                    ; est donc equivalent a
(ESET (QUOTE Il1>) I12>)
```

(NEXTL <sym1> <sym2>) [FSUBR]

<sym1> (qui n'est pas évalué) doit être un symbole dont la valeur doit être une liste. NEXTL retourne le CAR de cette liste en valeur et donne comme nouvelle valeur de <sym1> le CDR de son ancienne valeur. Si le deuxième argument <sym2> est donné, ce doit être un symbole, non évalué, qui recevra la valeur retournée par NEXTL.

Cette fonction est très utile pour *avancer dans une liste* qui est la valeur d'un atome.

```
; Cette fonction (a 1 argument) correspond en Lisp a :

(PROG1 (CAR Isym>) (SETQ Isym> (CDR Isym>)))

; a 2 arguments elle correspond a :

(PROG1 (SETQ Isym2> (CAR Isym1>))
       (SETQ Isym1> (CDR Isym1>)))
```

```
ex : (SETQ A '(X Y Z))    ->  (X Y Z)
     (NEXTL A)            ->  X
     A                    ->  (Y Z)
```

Compatibilité : cette fonction s'appelle POP en Maclisp.

(NEWL <sym> <s>) [FSUBR]

<sym> (qui n'est pas évalué) doit être un symbole dont la valeur est une liste. NEWL place en tête de cette liste la valeur de <s> et retourne en valeur la nouvelle liste formée. L'utilisation conjuguée des fonctions NEXTL et NEWL permet de construire très naturellement des piles-listes.

```
; cette fonction correspond en Lisp a :

    (SETQ Isym> (CONS Is> Isym>)).

ex : (SETQ A '(X Y Z))      ->  (X Y Z)
     (NEWL A 'W)            ->  (W X Y Z)
     A                      ->  (W X Y Z)
```

Compatibilité : cette fonction s'appelle PUSH en Maclisp mais possède ses arguments inversés.

(INCR <sym> <n>) [FSUBR]

<sym> doit être le nom d'un symbole qui possède une valeur numérique. INCR va incrémenter cette valeur de la valeur de l'expression <n> si celle-ci est donnée ou de 1 dans le cas contraire.
(INCR <sym> <n>) est équivalent à :
    (SETQ <sym> (+ <sym> <n>))
et (INCR <sym>) est équivalent à :
    (SETQ <sym> (1+ <sym>))

INCR peut être défini en Lisp de la manière suivante :

```
(DF INCR (l)
    (SET (CAR l)
         (IF (CDR l)
             (+  (SYMEVAL (CAR l)) (EVAL (CADR l)))
             (1+ (SYMEVAL (CAR l))))))
```

(DECR <sym> <n>) [FSUBR]

<sym> doit être le nom d'un symbole qui possède une valeur numérique. DECR va décrémenter cette valeur de la valeur de l'expression <n> si celle-ci est donnée ou de 1 dans le cas contraire.
(DECR <sym> <n>) est équivalent à :
    (SETQ <sym> (- <sym> <n>))
et (DECR <sym>) est équivalent à :
    (SETQ <sym> (1- <sym>))

DECR peut être défini en Lisp de la manière suivante :

```
(DF DECR (l)
    (SET (CAR l)
         (IF (CDR l)
             (-  (SYMEVAL (CAR l)) (EVAL (CADR l)))
             (1- (SYMEVAL (CAR l))))))
```

3.7.3 Les fonctions sur les P-Listes

En Lisp, les P-LIST (listes de propriétés) sont des listes constituées d'indicateurs et de valeurs qui ont la structure suivante :

```
(indic1 val1 indic2 val2 ... indicN valN)
```

A chaque indicateur indici est associée une valeur vali qui est l'élément suivant de la P-LIST. Les recherches sur les P-LIST s'effectuent donc deux éléments par deux éléments.

Chaque symbole possède une P-LIST qui lui est propre.

Il existe, en Lisp, 8 fonctions de manipulation de ces P-LIST.
Les arguments de ces fonctions sont :
<pl> - un symbole dont on veut utiliser la P-LIST .
Si <pl> est un nombre, une liste ou bien l'atome NIL, une erreur de type est déclenchée.
<ind> un indicateur. Celui-ci doit être un symbole, la recherche des indicateurs utilisant le prédicat EQ.
<pval> peut être n'importe quelle expression.

(PLIST <pl>) [SUBR à 1 argument]

retourne la P-LIST associée au symbole <pl>. Si le symbole <pl> ne possède pas de P-LIST, PLIST retourne NIL.

PLIST ne peut pas être défini simplement en Lisp.

(SETPLIST <pl> <l>) [SUBR à 2 arguments]

force la P-LIST du symbole <pl> avec la liste <l>. Cette fonction permet donc d'initialiser une P-LIST. SETPLIST retourne la nouvelle P-LIST (i.e. la valeur de <l>) en valeur. Du fait de la représentation des P-LIST en Lisp, SETPLIST ne peut pas être simplement défini en Lisp.

```
ex :   (SETPLIST 'rose '(nom commun genre feminin))
            ->  (nom commun genre feminin)
       (PLIST 'rose)  ->  (nom commun genre feminin)
```

(GET <pl> <ind>) [SUBR à 2 arguments]

retourne la valeur associée à l'indicateur <ind> dans la P-LIST du symbole <pl>. Si l'indicateur n'existe pas, GET retourne NIL.

ATTENTION : on ne peut pas discerner la valeur égale à NIL d'un indicateur avec l'absence de cet indicateur.

GET peut être défini en Lisp de la manière suivante :

```
(DE GET (pl ind)
    (LET-NAMED GET1 ((pl (PLIST pl)))
         (COND ((NULL pl) NIL)
               ((EQ (CAR pl) ind) (CADR pl))
               (T (GET1 (CDDR pl))))))
```

```
ex : (PLIST 'rose)          ->  (nom commum genre feminin)
     (GET 'rose 'genre)     ->  feminin
     (GET 'rose 'famille)   ->  NIL
```

(GETL <pl> <l>) [SUBR à 3 arguments]

<l> doit être une liste d'indicateurs. GETL recherche si l'un de ces indicateurs existe sur la P-liste du symbole <pl>. Si cette recherche réussi, GETL retourne la partie de la P-liste de <pl> commençant à cet indicateur.

GETL peut être défini en Lisp de la manière suivante :

```
(DE GETL (pl l)
    ; pl = une P-LIST
    ; l  = une liste d'indicateurs
    (LET-NAMED GETL-AUX ((pl (PLIST pl)))
        (COND
            ((NULL pl) pl)
            ((MEMQ (CAR pl) l) pl)
            (T (GETL-AUX (CDDR pl))))))
```

(ADDPROP <pl> <pval> <ind>) [SUBR à 3 arguments]

rajoute en tête de la P-LIST <pl> l'indicateur <ind> et sa valeur associée <pval>. ADDPROP retourne <pl> en valeur.

ADDPROP peut être défini en Lisp de la manière suivante :

```
(DE ADDPROP (pl pval ind)
    (SETPLIST pl (CONS ind (CONS pval (PLIST pl))))
    pl)
```

```
ex : (PLIST 'PLT)              ->  (I1 A I2 B)
     (ADDPROP 'PLT 'C 'I1)     ->  'PLT
     (PLIST 'PLT)              ->  (I1 C I1 A I2 B)
```

(PUTPROP <pl> <pval> <ind>) [SUBR à 3 arguments]

si l'indicateur <ind> existe déjà sur la P-LIST <pl>, sa valeur associée prend la nouvelle valeur <pval>, sinon l'indicateur <ind> et sa valeur associée <pval> sont ajoutés en tête de <pl> (d'une manière identique à la fonction ADDPROP). PUTPROP retourne <pl> en valeur.

PUTPROP peut être défini en Lisp de la manière suivante :

```
(DE PUTPROP (pl pval ind)
    (LET-NAMED PUT-AUX ((p (PLIST pl)))
         (COND ((NULL p) (ADDPROP pl pval ind))
               ((EQ (CAR p) ind) (RPLACA (CDR p) pval))
               (T (PUT-AUX (CDDR p)))))
   pl)

ex : (PLIST 'PLT)              ->  (I1 A I2 B)
     (PUTPROP 'PLT 'C 'I1)     ->  PLT
     (PLIST 'PLT)              ->  (I1 C I2 B)
     (PUTPROP 'PLT 0 'I9)      ->  PLT
     (PLIST 'PLT)              ->  (I9 0 I1 C I2 B)
```

(DEFPROP <pl> <pval> <ind>) [FSUBR]

est la version FSUBRée (i.e. qui n'évalue pas ses arguments) de la fonction précédente. Utile quand tous les arguments de PUTPROP sont des constantes.

(REMPROP <pl> <ind>) [SUBR à 2 arguments]

enlève de la P-LIST <pl> l'indicateur <ind> s'il existe ainsi que sa valeur associée. REMPROP retourne <pl> en valeur.
L'utilisation conjuguée des fonctions REMPROP et ADDPROP permet d'utiliser les P-LIST comme des piles de propriété-valeurs.

REMPROP peut être défini en Lisp de la manière suivante :

```
(DE REMPROP (pl ind)
    (LET-NAMED REM-AUX ((pl1 pl) (pl2 (PLIST pl)))
         (COND ((NULL pl2) ())
               ((EQ (CAR pl2) ind) (RPLACD pl1 (CDDR pl2)))
               (T (REM-AUX pl2 (CDDR pl2))))))

ex : (PLIST 'PLT)              ->  (I1 A I2 B)
     (REMPROP 'PLT 'I2)        ->  PLT
     (PLIST 'PLT)              ->  (I1 A)
```

3.8 Les fonctions sur les P-Names

«««< Toutes les fonctions de cette section risquent de changer dans les nouvelles versions du système Le_Lisp 68K »»»>

Rappelons que :
- le P-NAME d'un symbole est son nom externe
- le P-NAME d'un nombre est sa représentation externe dans la base de conversion de sortie courante.
- le P-NAME d'une chaîne de caractères est cette suite de caractères

Toutes ces fonctions peuvent être décrites au moyen des 2 fonctions de base, IMPLODE et EXPLODE qui permettent de passer de la représentation interne d'un P-NAME à sa représentation externe et vice-versa.

(EXPLODE <s>) [SUBR à 1 argument]

retourne la liste de tous les codes ASCII de la représentation externe de l'expression <s>, qui peut être de type quelconque. EXPLODE retourne la liste de codes qui serait imprimée si on demandait l'impression de <s> au moyen de la fonction PRIN (du reste il n'y a pas de fonction EXPLODE à proprement parler dans l'interprète mais un changement de direction du flux de sortie produit par la fonction PRIN).

```
ex : (EXPLODE -120)              ->  (45 49 50 48)
     (EXPLODE '(CAR '(A B)))
         ->  (40 67 65 82 32 39 40 65 32 66 41 41)
```

(IMPLODE <ln>) [SUBR à 1 argument]

suppose que <ln> est une liste de codes ASCII. IMPLODE retourne l'objet Lisp qui possède comme représentation externe la liste de caractères <l>. IMPLODE est l'inverse de EXPLODE et permet de fabriquer de nouveaux objets Lisp au moyen de leur représentation externe DE LA MEME MANIERE que si ces caractères étaient lus par la fonction READ (il n'y a pas de fonction spéciale IMPLODE dans l'interprète mais simplement un changement d'origine du flux d'entrée de la fonction READ).

```
ex : (IMPLODE '(45 50 51 55))     ->  -120
     (IMPLODE (EXPLODE '(A B)))   ->  (A B)
```

(PLENGTH <sym>) [SUBR à 1 argument]

retourne en valeur le nombre de caractères du P-NAME du symbole <sym>. Si l'argument <sym> n'est pas un symbole, PLENGTH retourne 0.

PLENGTH peut être défini en Lisp de la manière suivante :

```
(DE PLENGTH (at)
    (IF (SYMBOLP at)
```

```
            (LENGTH (EXPLODE at))
            0))

ex :  (PLENGTH 'FOOBAR)  ->  6
      (PLENGTH ())       ->  3  ; taille de l'atome NIL
      (PLENGTH 120)      ->  0
```

(CHRPOS <cn> <sym>) [SUBR à 2 arguments]

retourne en valeur la position du caractère de code ASCII <cn> dans le P-NAME du symbole <sym>. Si le code <cn> n'existe pas dans le P-NAME de l'atome CHRPOS retourne NIL. La position (le rang) du premier caractère est 0 (et non 1).

CHRPOS peut eêtre de'fini en Lisp de la manie`re suivante :

```
(DE CHRPOS (c at)
    (LET-NAMED CHRP ((l (EXPLODE at)) (n 0))
         (COND ((NULL l) NIL)
               ((EQ (CAR l) c) n)
               (T (CHRP (CDR l) (1+ n))))))

ex : (CHRPOS 'Y "OTYoty")            ->  3
     (CHRPOS 'N "OTY")               ->  NIL
     (CHRPOS 'D "0123456789ABCDEF")  ->  14
```

(CHRNTH <n> <sym>) [SUBR à 2 arguments]

retourne le <n>ième caractère du P-NAME du symbole <sym> et s'il n'existe pas retourne la valeur NIL.

CHRNTH peut eêtre de'fini en Lisp de la manie`re suivante :

```
(DE CHRNTH (n at)
    (IF (SYMBOLP at)
        (NTH n (EXPLODE at))
        ()))

ex : (CHRNTH 0 'FOO)                 ->  F
     (CHRNTH 10 "0123456789ABCDEF")  ->  A
```

(ALPHALESSP <sym1> <sym2>) [SUBR à 2 arguments]

retourne T si le P-NAME du symbole <sym1> est inférieur ou égal (lexicographiquement) au P-NAME du symbole <sym2>,sinon retourne NIL. Cette fonction est utilisée pour réaliser des tris alphabétiques.

ALPHALESSP peut eêtre de'fini en Lisp de la manie`re suivante :

```
(DE ALPHALESSP (A1 A2)
    (LET-NAMED ALPHALESSP1 ((L1 (EXPLODE A1))(L2 (EXPLODE A2)))
         (COND ((NULL L1) T)
```

```
                ((NULL L2) NIL)
                ((= (CASCII (CAR L1)) (CASCII (CAR L2)))
                    (ALPHALESSP1 (CDR L1) (CDR L2)))
                ((< (CASCII (CAR L1)) (CASCII (CAR L2))))))))

ex : (ALPHALESSP 'A 'A)        ->  T
     (ALPHALESSP 'B 'A)        ->  NIL
     (ALPHALESSP 'A 'B)        ->  T
     (ALPHALESSP 'ZZZ 'ZZZZ)   ->  T
```

Comment réaliser le tri d'une liste par échange physique :

```
(DE SORTL  (l)
    (LET ((s (APPEND l)))
         (SETQ s (APPEND l))
         (MAP (LAMBDA (sl1)
                (IF (CDR l)
                   (MAP (LAMBDA (sl2)
                          (IF (ALPHALESSP (CAR sl1) (CAR sl2))
                             ()
                             (RPLACA sl2
                                (PROG1 (CAR sl1)
                                   (RPLACA sl1 (CAR sl2))))))
                        (CDR sl1))))
              S)
         S)
```

Voici une autre manière de trier une liste d'atomes au moyen d'un tri-fusion récursif

```
   (DE TRI-FUSION (l)
       ; tri la liste l
       (IF (NULL (CDR l))
           l
           (FUSION (TRI-FUSION (MOITIE1 l)) (TRI-FUSION (MOI!
TIE2 l)))))

   (DE FUSION (l1 l2)
       ; fusionne les 2 listes triees l1 et l2
       (COND
           ((NULL l1) l2)
           ((NULL l2) l1)
           ((ALPHALESSP (CAR l1) (CAR l2))
               (CONS (CAR l1) (FUSION (CDR l1) l2)))
           (T  (CONS (CAR l2) (FUSION l1 (CDR l2))))))

   (DE MOITIE1 (l)
       ; retourne la 1ere moitie de l
       (FIRSTN (// (LENGTH l) 2) l))

   (DE MOITIE2 (l)
       ; retourne la 2eme moitie de l
       (NTHCDR (// (LENGTH l) 2) l))
```

Cet exemple n'est pas optimum quant au nombre de CONS effectués. L'utilisation des fonctions de modification physique permet la suppression de tous les CONS des fonctions FUSION et MOITIE1 voici une version rapide du tri récursif :

```
(DE QUICKSORTL (L)
    (IF (NULL (CDR L))
        ()
        (LETSEQ ((L1 L)
                 (L  (NTHCDR (// (LENGTH L) 2) L))
                 (L2 (PROG1 (CDR L) (RPLACD L ())))
                (FFUSION (QUICKSORTL L1) (QUICKSORTL L2)))))


(DE FFUSION (L1 L2)
    ; fusionne physiquement les 2 listes triees
    (IF (ALPHALESSP (CAR L2) (CAR L1)) (PSETQ L1 L2 L2 L1))
    (LET-NAMED FFUSION1 ((R L1) (L1 (CDR L1)) (L2 L2))
         (COND ((NULL L1) (RPLACD R L2))
               ((NULL L2) (RPLACD R L1))
               ((ALPHALESSP (CAR L1) (CAR L2))
                  (RPLACD R L1)
                  (FFUSION1 L1 (CDR L1) L2))
               (T (RPLACD R L2)
                  (FFUSION1 L2 L1 (CDR L2)))))
    L1)
```

(GENSYM <n>) **[SUBR à 1 argument]**

retourne à chaque appel un nouveau symbole de type Gxxx dans lequel xxx est un nombre incrémenté de 1 à chaque appel (on appelle ce nombre le compteur de GENSYM); xxx vaut 100 au départ de l'interprète. Si l'argument <n> est fourni il devient le nouveau compteur de GENSYM.

```
; si GENSYM-COUNTER est le nom de la variable globale qui
; contient le compteur de GENSYM, il doit etre initialise :

   (SETQ GENSYM-COUNTER 100 GENSYM-PREF #/G)

; GENSYM peut etre defini en Lisp de la maniere suivante :

   (DE GENSYM (n)
       (WHEN (NUMBERP n) (SETQ GENSYM-COUNTER n))
       (WHEN (SYMBOLP n) (SETQ GENSYM-PREF n))
       (INCR GENSYM-COUNTER)
       (IMPLODE (APPEND GENSYM-PREF (EXPLODE GENSYM-COUNTER))))

ex : (GENSYM)                    ->  G101
     (GENSYM)                    ->  G102
     (GENSYM)                    ->  G103
```

Une utilisation intéressante de la fonction GENSYM est la possibilité de réaliser une protection contre les collisions homonymiques de certaines variables critiques contenues

dans différents fichiers. Autrement dit, il faut pouvoir s'assurer que certaines variables d'un fichier seront uniques en ce sens qu'aucun autre atome ne pourra avoir le même P-NAME dans d'autres fichiers.

Pour ce faire on peut définir un macro-caractère (ici le ~) qui sera placé devant toutes les occurrences de ces variables critiques. Pour réaliser cette protection de variables, durant la lecture d'un fichier, il faut l'organiser de la manière suivante :

```
; en tete du fichier faire

(SETQ prefix (EXPLODE (GENSYM)))

(DMC k~| () (IMPLODE (APPEND prefix (EXPLODE (READ)))))

    .....

(DE FOO (~l s )

    ; les variables l et s sont maintenant protegees
    ; car elles sont traduites : G1011  G101s ...
    .....

; et en fin de fichier provoquer l'irremediable

(GENSYM)
```

3.9 Les fonctions sur les caractères

Ces fonctions vont manipuler les codes internes des caractères <cn>.

(ASCII <cn>) [SUBR à 1 argument]

retourne le caractère de code ASCII <cn> (modulo 256).

```
ex : (ASCII 67)          ->  C
     (1+ (ASCII 49))     ->  2
```

(CASCII <ch>) [SUBR à 1 argument]

retourne le code ASCII du caractère <ch>. Un caractère étant un symbole ou un nombre dont le P-LEN est égal à 1.

```
ex : (CASCII 'C)  ->  67
     (CASCII 1)   ->  49
```

(UPPERCASE <cn>) [SUBR à 1 argument]

si <cn> représente le code interne d'un caractère miniscule, UPPERCASE retourne le code de ce caractère majuscule. Dans le cas contraire cette fonction retourne <cn> inchangé.

(LOWERCASE <cn>) [SUBR à 1 argument]

si <cn> représente le code interne d'un caractère majuscule, LOWERCASE retourne le code de ce caractère minuscule. Dans le cas contraire cette fonction retourne <cn> inchangé.

(DIGITP <cn>) [SUBR à 1 argument]

teste si <cn> est le code interne d'un chiffre décimal. DIGITP retourne le code si le test est vérifié et NIL dans le cas contraire.

3.10 Les fonctions sur les chaînes de caractères

«««««« à finir »»»»»»

(STRING <s>) [SUBR à 1 argument]

(SUBSTRING <str> <n1> <n2>) [SUBR à 3 arguments]

(CATENATE <str1> ... <strN>) [SUBR à N arguments]

(INDEX <str1> <str2> <n>) [SUBR à 3 arguments]

(EQSTRING <str1> <str2>) [SUBR à 2 arguments]

3.11 Les fonctions sur les nombres

Le_Lisp possède plusieurs types de nombres :

- les nombres entiers (sur 16 bits)
- les nombres flottants (sur 32 bits)
- les nombres entiers à précision infinie

Sauf pour les fonctions de test de type de nombres et de conversion explicite de type, il y a conversion automatique des types.

Si au cours d'un calcul en précision fixe (entier ou flottant) un débordement se produit, le système teste l'état de la variable système STATUS-OVERFLOW. Si ce symbole est égal à NIL (ce qui est l'option par défaut) le calcul se poursuit mais le résultat sera bien évidemment faux. Si le symbole STATUS-OVERFLOW possède une valeur différente de NIL, une erreur se produit dont le libellé est :

** <fnt> : débordement numérique. ou bien
** <fnt> : numeric overflow.

dans lequel le nom de la fonction qui a provoqué l'erreur <fnt> est imprimé.

3.11.1 Les tests de type

En plus de la fonction NUMBERP qui teste si son argument est un nombre, il existe 3 fonctions qui testent le type des nombres.

(FIXP <s>) **[SUBR à 1 argument]**

retourne <s> si <s> est un nombre entier de précision fixe et NIL dans le cas contraire.

```
ex :  (FIXP 120)       ->  120
      (FIXP 10.256)    ->  NIL
      (FIXP 4599999)   ->  NIL
```

(FLOATP <s>) **[SUBR à 1 argument]**

retourne <s> si <s> est un nombre flottant de précision fixe et NIL dans le cas contraire.

```
ex :  (FLOATP 120)       ->  NIL
      (FLOATP 10.256)    ->  10.256
      (FLOATP 4599999)   ->  NIL
```

(BIGP <s>) **[SUBR à 1 argument]**

retourne <s> si <s> est un nombre entier de précision variable et NIL dans le cas contraire.

```
ex :  (BIGP 120)       ->  NIL
      (BIGP 10.256)    ->  NIL
      (BIGP 4599999)   ->  4599999
```

3.11.2 Les conversions numériques

Bien que la plupart des fonctions numériques convertissent leurs arguments, il est possible de réaliser des conversions explicitement au moyen des 2 fonctions :

(FIX <n>) [SUBR à 1 argument]

si l'argument <n> est un nombre entier, FIX le retourne. Si <n> est un nombre flottant <n> est converti en entier si c'est possible sinon une erreur de débordement numérique apparait de même si <n> n'est pas un nombre.

(FLOAT <n>) [SUBR à 1 argument]

si l'argument <n> est un nombre flottant, FLOAT le retourne. Si <n> est un nombre entier, il est converti en flottant. Si <s> n'est pas un nombre, une erreur de débordement numérique apparait.

3.11.3 L'arithmétique mixte

Les arguments de ces fonctions peuvent être des nombres de n'importe quel type : entiers ou flottants. Si l'un des arguments est flottant, le résultat de ces fonctions est un nombre flottant ; si tous les arguments sont de type entier, le résultat de ces fonctions est un nombre entier. Il y a donc conversion automatique des arguments avec priorité aux nombres flottants.

(1+ <n>) [SUBR à 1 argument]

retourne la valeur : In> E 1 .

```
ex :  (1+ 6)      ->  7
      (1+ -3)     ->  -2
```

(1- <n>) [SUBR à 1 argument]

retourne la valeur : <n> - 1

```
ex : (1- 7)       ->  6
     (1- -3)      ->  -4
```

(+ <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur de la somme : <n1> + <n2> + ... + <nN> Si aucun argument n'est fourni, + retourne l'élément neutre de l'addition 0.

```
ex : (+ 5 6 7)      ->  18
     (+ 5 6)        ->  11
     (+ 8)          ->  8
     (+)            ->  0
```

(- <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur de la différence : <n1> - <n2> - ... - <nN> Si aucun argument n'est fourni, - retourne l'élément neutre de la soustraction, 0. Si cette fonction ne possède qu'un argument elle réalise une négation arithmétique.

```
ex : (- 8 3 2)      ->  3
     (- 6 2)        ->  4
     (- 12)         ->  -12
     (-)            ->  0
```

(* <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur du produit : <n1> * <n2> * ... * <nN> Si aucun argument n'est fourni, * retourne l'élément neutre de la multiplication, 1.

```
ex : (* 5 6 7)      ->  210
     (* 10 4)       ->  40
     (* 100)        ->  100
     (*)            ->  1
```

(/ <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du quotient de : <n1> / <n2>. Si l'argument <n2> est égal à 0, une erreur se produit. Dans certaines versions de Le_Lisp, la caractère / est un caractère spécial qui permet de quoter des caractères en entrée (voir les fonctions d'entrée). Dans ce cas il faut soit quoter ce caractère lui-même au moyen d'un double slash //, soit utiliser la notation |/|.

```
ex : (/ 20 5)       ->  4
     (/ 10 (1- 1))  ->  ** erreur division par 0

     ; et si ! est un caractere special

     (// 100 5)     ->  20     ou bien
     (|/| 100 5)    ->  20
```

(<n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du reste de la division de <n1> par <n2>.

```
ex : ( 11 3)    ->  2
     ( 12 4)    ->  0
```

(ABS <n>) [SUBR à 1 argument]

retourne la valeur absolue de l'argument <n>.

```
ex : (ABS 10)   ->  10
     (ABS -10)  ->  10
     (ABS ())   ->  0

     (* x (ABS x))  ; calcule le carre de x en conservant son signe,
                    ; dit G. Baudet.
```

3.11.4 L'arithmétique entière

Les fonctions qui vont être décrites utilisent des opérandes <n> supposés de type entier. Ces fonctions n'effectuent aucun contrôle de validité de type. Si les arguments ne sont pas des nombres entiers, ces fonctions livrent en général de bien étranges résultats.

L'utilisation de ces fonctions n'est pas recommandée pour des travaux ordinaires. En effet elles ne sont utilisées que par des programmes systèmes (comme le compilateur) pour des raisons d'efficacité.

(ADD1 <n>) [SUBR à 1 argument]

retourne la valeur : In> E 1 .

```
ex :  (ADD1 6)   ->  7
      (ADD1 -4)  ->  -3
```

(SUB1 <n>) [SUBR à 1 argument]

retourne la valeur : <n> - 1

```
ex : (SUB1 7)   ->  6
     (SUB1 -9)  ->  -10
```

(ADD <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la somme : <n1> + <n2>

```
ex : (ADD 5 6)      ->  11
     (ADD -5 -6)    ->  -11
```

(SUB <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la différence : <n1> - <n2>

```
ex : (SUB 6 2)      ->  4
     (SUB 12 -7)    ->  19
```

(MUL <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du produit : <n1> * <n2>

```
ex : (MUL 10 4)       ->  40
     (MUL 2 -3)       ->  -6
     (MUL -100 -100)  ->  10000
```

(DIV <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du quotient de : <n1> / <n2>.
DIV effectue une division entière sur des nombres entiers. Si l'argument <n2> est égal à 0, une erreur se produit.

```
ex : (DIV 23 5)     ->  4
     (DIV 40 -4)    ->  -10
```

(REM <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du reste de la division entière de <n1> par <n2>.

```
ex : (REM 11 3)  ->  2
     (REM 14 22) ->  14
```

(SCALE <n1> <n2> <n3>) [SUBR à 3 arguments]

retourne la valeur du calcul <n1> * <n2> / <n3> (SCALE <n1> <n2> <n3>) est donc équivalent à (DIV (MUL <n1> <n2>) <n3>) La différence réside dans le fait que le résultat intermédiaire est rangé en double précision (dans ce cas 32 bits).

```
ex :  (SCALE 1000 1000 1000)   ->  1000
```

3.11.5 Les tests de l'arithmétique mixte

Ces fonctions possèdent toutes 2 arguments <n1> et <n2> qui peuvent être des nombres de n'importe quel type (entier ou flottant). Si l'un des arguments est flottant, ces fonctions réalisent des comparaisons flottantes ; si tous les arguments sont de type entier, les fonctions réalisent des comparaisons entières. Il y a donc conversion automatique des arguments avec priorité aux nombres flottants.

ATTENTION : d'une manière générale, les fonctions qui vont être décrites sont à préférer aux fonctions de la section suivante car elles existent dans tous les systèmes (que le système possède ou non des nombres flottants) et sont plus générales.

Toutes ces fonctions retournent le 1er argument si le test est vérifié et NIL dans le cas contraire.

(ZEROP <n>) **[SUBR à 1 argument]**

retourne 0 si l'argument <n> est égal à zéro et NIL dans le cas contraire.

(PLUSP <n>) **[SUBR à 1 argument]**

si <n> est plus grand ou égal à 0 alors PLUSP retourne <n> sinon PLUSP retourne NIL.

```
ex : (PLUSP 0)   ->  0
     (PLUSP -1)  ->  NIL
```

(MINUSP <n>) **[SUBR à 1 argument]**

si <n> est plus petit strictement que 0, MINUSP retourne <n> sinon MINUSP retourne NIL.

```
ex : (MINUSP 0)   ->  NIL
     (MINUSP -1)  ->  -1
```

(EVENP <n>) **[SUBR à 1 argument]**

si l'argument <n> est paire, alors EVENP retourne <n> sinon EVENP retourne NIL.

(ODDP <n>) **[SUBR à 1 argument]**

si l'argument <n> est impaire alors ODDP retourne <n> sinon ODDP retourne NIL.

(= <n1> <n2>) [SUBR à 2 arguments]

si <n1> = <n2> alors = retourne <n1>, sinon = retourne NIL.

```
ex : (= 10 10)      ->  10
     (= -3 3)       ->  NIL
```

(<> <n1> <n2>) [SUBR à 2 arguments]

si <n1> # <n2> alors <> retourne <n1>, sinon <> retourne NIL.

```
ex : (<> 11 10)     ->  11
     (<> -3 -3)     ->  NIL
```

(>= <n1> <n2>) [SUBR à 2 arguments]

si <n1> >= <n2> alors >= retourne <n1> sinon >= retourne NIL.

```
ex : (>= 3 7)       ->  NIL
     (>= 7 7)       ->  7
```

(> <n1> <n2>) [SUBR à 2 arguments]

si <n1> > <n2> alors > retourne <n1> sinon > retourne NIL.

```
ex : (> 5 5)        ->  NIL
     (> 7 4)        ->  7
```

(<= <n1> <n2>) [SUBR à 2 arguments]

si <n1> <= <n2> alors <= retourne <n1> sinon <= retourne NIL.

```
ex : (<= 5 5)       ->  5
     (<= 4 6)       ->  4
     (<= 8)         ->  NIL
```

(< <n1> <n2>) [SUBR à 2 arguments]

si <n1> < <n2> alors < retourne <n1> sinon < retourne NIL.

```
ex : (< 5 5)        ->  NIL
     (< 4 5)        ->  4
     (< 0)          ->  NIL
```

3.11.6 Les tests de l'arithmétique entière

Ces fonctions possèdent toutes 2 arguments <n1> et <n2> qui doivent être des nombres de type entier. Elles n'effectuent aucun test de validité de type et ne provoquent donc jamais d'erreur. Toutefois si les arguments de ces fonctions ne sont pas des nombres entiers, leurs résultats ne sont pas significatifs.

Toutes ces fonctions retournent le 1er argument si le test est vérifié et NIL dans le cas contraire.

L'utilisation de ces fonctions n'est pas recommandée pour des travaux ordinaires. En effet elles ne sont utilisées que par des programmes systèmes (comme le compilateur) pour des raisons d'efficience.

(EQN <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> = <n2> alors EQN retourne <n1>, sinon EQN retourne NIL.

```
ex : (EQN 10 10)    ->  10
     (EQN -3)       ->  NIL
```

(NEQN <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> est différent <n2> alors NEQN retourne <n1>, sinon NEQN retourne NIL.

```
ex : (NEQN 11 10)   ->  11
     (NEQN -3 -3)   ->  NIL
```

(GE <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> >= <n2> alors GE retourne <n1> sinon GE retourne NIL.

```
ex : (GE 3 7)       ->  NIL
     (GE 7 7)       ->  7
```

(GT <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> > <n2> alors GT retourne <n1> sinon GT retourne NIL.

```
ex : (GT 5 5)     ->  NIL
     (GT 7 4)     ->  7
```

(LE <n1> <n2>) [SUBR à 2 arguments]

si <n1> <= <n2> alors LE retourne <n1> sinon LE retourne NIL.

```
ex : (LE 5 5)      ->  5
     (LE 4 6)      ->  4
     (LE 8)        ->  NIL
```

(LT <n1> <n2>) [SUBR à 2 arguments]

si <n1> < <n2> alors LT retourne <n1> sinon LT retourne NIL.

```
ex : (LT 5 5)      ->  NIL
     (LT 4 5)      ->  4
     (LT 0)        ->  NIL
```

3.11.7 Les fonctions logiques

Pour toutes les fonctions qui vont être décrites, les arguments <n1> et <n2> doivent être de type entier. Ces fonctions ne travaillent que sur des mots de 16 bits, ne réalisent aucune conversion de type et ne provoquent jamais d'erreur.

(LOGNOT <n>) [SUBR à 1 argument]

retourne la valeur du complément logique de <n>.

(LOGNOT <n>) est équivalent à (LOGXOR <n> #$FFFF)

```
ex : (LOGNOT 0)    ->  -1       ; en hexadecimal #$FFFF
     (LOGNOT -2)   ->  1
```

(LOGAND <n1> <n2>) [SUBR à 2 arguments]

effectue l'opération de ET logique entre
<n1> et <n2>.

```
ex : (LOGAND #$36 #$25)    ->  #$24

     ; pour savoir si In> est une puissance de 2 evaluez :

     (= In> (LOGAND In> (MINUS In>)))
```

(LOGOR <n1> <n2>) [SUBR à 2 arguments]

effectue l'opération de OU logique entre <n1> et <n2>.

```
ex : (LOGOR #$15 #$17)   ->  #$17
```

(LOGXOR <n1> <n2>) [SUBR à 2 arguments]

effectue l'opération de OU exclusif logique entre <n1> et <n2>.

```
ex : (LOGXOR 5 3)   ->  6
```

(LOGSHIFT <n1> <n2>) [SUBR à 2 arguments]

```
(DE LOGSHIFT (n nb)
   ; decale logiquement la valeur n de nb positions.
   ; Si nb est 00, le decalage s'effectue a gauche,
   ; sinon si nb I0, le decalage s'effectue a droite.
   (IF (= nb 0)
       n
       (IF (< nb 0)
           (LOGSHIFT (// n 2) (1+ nb))
           (LOGSHIFT (* n 2) (1- nb)))))
```

3.11.8 Les fonctions sur champ de bits

Toutes ces fonctions travaillent sur des mots de 16 bits et permettent de manipuler des champs arbitraires à l'intérieur de ces mots. Un champ de bit est décrit au moyen de 2 nombres, <np> le numéro du 1er bit du champ (compté à partir de 0, le bit 0 étant le bit de plus faible poids) et <nl> la longeur de ce champ.

(2* <n>) [SUBR à 1 argument]

retourne la valeur 2 à la puissance <n>, i.e. le bit de rang <n> d'un mot.

2** peut être défini en Lisp de la manière suivante :

```
(DE 2** (n)
    (LOGSHIFT 1 n))
```

(LOAD-BYTE <n> <np> <nl>) [SUBR à 3 arguments]

extrait du mot <n> le champ spécifié par <np> et <nl>.

LOAD-BYTE peut etre de'fini en Lisp de la manie`re suivante :

```
(DE LOAD-BYTE (n p l)
    (LOGSHIFT (MASK-FIELD n p l) (- p)))
```

(MASK-FIELD <n> <np> <nl>) [SUBR à 3 arguments]

retourne le mot résultant de l'isolation du champ de bit <np> <nl>.

MASK-FILED peut etre de'fini en Lisp de la manie`re suivante :

```
(DE MASK-FIELD (n p l)
    (LOGAND (LOGSHIFT (1- (2** l)) p) n))
```

(DEPOSIT-BYTE <n1> <np> <nl> <n2>) [SUBR à 4 args]

retourne le mot <n1> dans lequel le champ <np> <nl> a été remplacé par la valeur <n2>.

DEPOSIT-BYTE peut etre de'fini en Lisp de la manie`re suivante :

```
(DE DEPOSIT-BYTE (n1 p l n2)
    (LET ((n    (MASK-FIELD (LOGSHIFT n2 p) p l))
          (numr (LOGAND n1 (1- (2** p))))
          (numl (LOGAND n1 (LOGNOT (1- (2** (+ p l))))))
        (+ n numl numr))))))
```

(DEPOSIT-FIELD <n1> <np> <nl> <n2>) [SUBR à 4 args]

retourne le mot <n1> dans lequel le champ <np> <nl> a été remplacé par le champ correspondant du mot <n2>.

DEPOSIT-BYTE peut etre de'fini en Lisp de la manie`re suivante :

```
(DE DEPOSIT-FIELD (n1 p l n2)
    (LET ((n    (MASK-FIELD n2 p l))
          (numr (LOGAND n1 (1- (2** p))))
          (numl (LOGAND n1 (LOGNOT (1- (2** (+ l p)))))))
        (+ n numl numr))))))
```

(LOAD-BYTE-TEST <n> <np> <nl>) [SUBR à 3 args]

teste si le champ <np> <nl> est égal à zéro. Si le test est vérifié, LOAD-BYTE-TEST retourne 0 sinon il retourne NIL.

LOAD-BYTE-TEST peut etre de'fini en Lisp de la manie`re suivante :

```
(DE LOAD-BYTE-TEST (n p l)
    (<> (LOAD-BYTE n p l) 0))
```

3.12 Les fonctions sur les tableaux

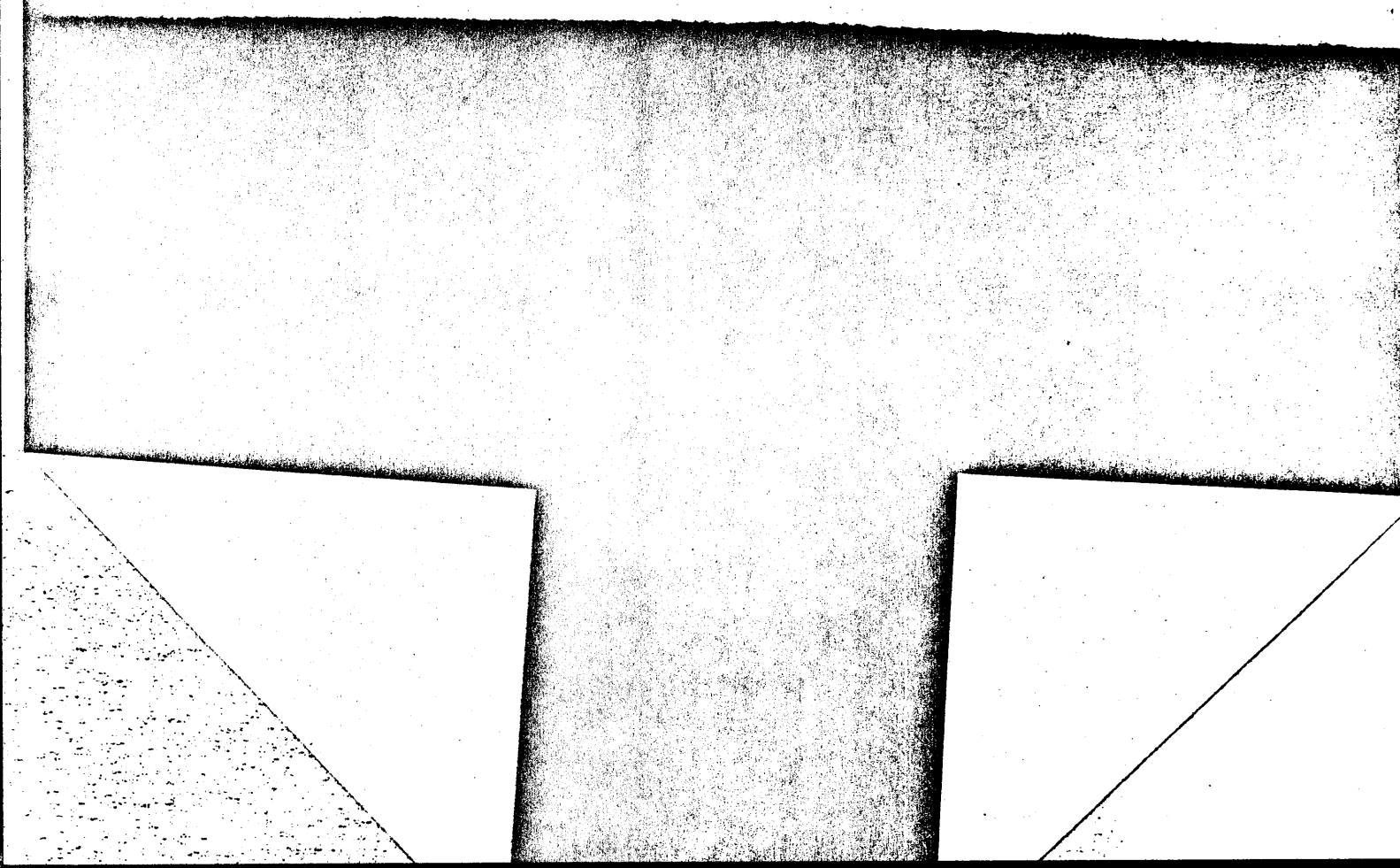
«««««« à finir »»»»»»

(ARRAY <sym> <typearay> <dim1> ... <dimN>) [FSUBR]

(AREF <sym> <dim1> ... <dimN>) [FSUBR]

(ASET <s> <sym> <dim1> ... <dimN>) [FSUBR]

(MAPARRAY <fn> <sym>) [SUBR à 2 arguments]



CHAPITRE 4

LES ENTRéES/SORTIES

Les entrées/sorties s'effectuent au niveau du caractère, de la ligne ou de la S-expression Lisp, sur des flux séquentiels.

Toutefois de nombreuses fonctions ont été rajoutées pour pouvoir utiliser des périphériques spéciaux tels :

- des terminaux graphiques
- des synthétiseurs de musique
- des écrans couleur

Il existe deux façons de représenter des caractères :

- sous forme d'atome mono-caractère (nombre ou symbole) <ch>
- sous forme de nombre représentant les codes ASCII de ces caractères <cn>

Le_Lisp utilise l'une ou l'autre de ces représentations. Les fonctions qui utilisent la première <ch> sont en général suffixée par CH et celles utilisant la représentation sous forme de code ASCII par CN. Cette deuxième représentation est toujours utilisée en cas de liste de caractères <lcn>.

Compatibilité : c'est en général au niveau des Entrées\Sorties que se manifeste le plus grand nombre d'incompatibilités entre les différents dialectes de Lisp. Nous ne saurons trop conseiller de toujours encapsuler les Entrées\Sorties dans des fonctions utilisateurs redéfinissables.

4.1 Les fonctions d'entrée de base

Toutes les lectures sont réalisées dans le flux d'entrée qui est associé soit au terminal, soit à un fichier sélecté par la fonction INPUT.

Voici les fonctions d'entrée de base.

(READ) [SUBR à 0 argument]

lit la S-expression suivante de type quelconque (atome ou liste) du flux d'entrée et la retourne en valeur. READ est la principale fonction de lecture et permet de lire des objets Lisp. Son fonctionnement détaillé est donné dans la section suivante.

Si une erreur de syntaxe Lisp est détectée, une erreur fatale se produit et le libellé suivant est imprimé :

** <fnt> : erreur de syntaxe : <n> ou bien
** <fnt> : syntax error : <n>

dans lequel le nom de la fonction qui a provoqué l'erreur est imprimé ainsi que le type de cette erreur. Voici la liste des erreurs :

1 : la liste du IMPLODE est trop petite
2 : chaîne de caractères trop longue
3 : P-NAME de symbole trop long
4 : unité syntaxique débutant par) ou .
5 : une) ne ferme pas une (
7 : mauvaise construction . Is>)
9 : la liste du IMPLODE ne contient pas que des caractères
10 : la liste du REREAD ne contient pas que des caractères

Le plus souvent cette erreur provient d'une mauvaise utilisation des paires pointées ou d'un P-NAME trop long (de plus de 62 caractères). dû à l'oubli du caractère | ou "".

(READLINE) [SUBR à 0 argument]

lit la ligne suivante du flux d'entrée et retourne la liste des caractères de cette ligne. Cette liste ne contient pas le caractère fin de ligne (en l'occurrence le caractère Return) ni les caractères line-feed. READLINE est utilisée pour réaliser à peu de frais les interfaces entre programme et utilisateur.

ATENTION : cette fonction n'effectue pas de conversion automatique des caractères minuscules en caractères majuscules.

READLINE peut etre de'fini en Lisp de la manie` re suivante :

```
(DE READLINE ()
    (LET ((l) (c))
```

```
            (WHILE (NEQ (SETQ c (READCN)) #CR)
                   (NEWL l c))
            (NREVERSE  l)))

ex :  ? (MAPCAR 'ASCII (READLINE)) ; appelle la lecture d'une ligne
      Le gai rossignol             ; ligne terminee par Return
      = (L e !  g a i !  r o s s i g n o l ) ; liste des caracte!
res lus

      ; pour fabriquer une suite de mots a partir d'une ligne
      ; qui ne contient pas de caracteres speciaux ( ) . ■ $

      ? (IMPLODE (APPEND #"(" (READLINE) #")")))
      Le merle moqueur             ; ligne terminee par Return
      = (LE MERLE MOQUEUR)         ; liste des atomes lus
```

(READCH) [SUBR à 0 argument]

lit et retourne en valeur le caractère suivant du flux d'entrée. Ce caractère est retourné sous la forme d'un atome (symbole pour les lettres et nombre pour les chiffres).
Cette fonction n'effectue pas de conversion automatique des caractères minuscules en caractères majuscules.

(READCN) [SUBR à 0 argument]

lit et retourne en valeur le caractère suivant du flux d'entrée. Ce caractère est retourné sous la forme d'un code interne (i.e. d'un nombre). Comme pour la fonction précédente, il n'y a pas de conversion automatique des caractères minuscules en caractères majuscules.

(PEEKCH) [SUBR à 0 argument]

retourne en valeur le caractère du flux d'entrée d'une manière identique à la fonction READCH, toutefois, ce caractère n'est pas véritablement lu mais seulement *consulté*. Il en résulte que des appels successifs de la fonctions PEEKCH retournent toujours le même résultat. PEEKCH est utilisée pour tester le caractère suivant du flux d'entrée AVANT de le lire véritablement.

(PEEKCN) [SUBR à 0 argument]

est identique à la fonction précédente, mais retourne le code ASCII du caractère suivant du flux d'entrée qui n'est pas lu mais simplement consulté.

(TEREAD) [SUBR à 0 argument]

passe à la ligne suivante du flux d'entrée en ignorant le reste éventuel de la ligne courante. Cette fonction permet d'effacer le contenu du tampon d'entrée. TEREAD retourne NIL en valeur.

(REREAD <lcn>) [SUBR à 1 argument]

remet la liste des caractères <lcn> en tête du flux d'entrée. Ces caractères seront relus au prochain appel d'une des fonctions de lecture vues précédemment (READ, READLINE, READCH ...). Cette fonction particulièrement puissante permet de construire dynamiquement une liste de caractères et de fabriquer sa représentation interne.

PEEKCN peut eîre de'fini en Lisp de la manie` re suivante :

```
(DE PEEKCN ()
    (LET ((cn (READCN)))
         (REREAD (LIST cn))
         cn))

(PROGN (REREAD (LIST #/' #/a)) (READ))  ->  (QUOTE A)
```

4.2 Contrôle des fonctions d'entrée

4.2.1 Utilisation du terminal en entrée

A chaque demande de ligne sur le terminal, le système imprime le caractère ? puis un certain nombre d'espaces en fonction de la profondeur de la lecture (i.e. du nombre de parenthèses ouvrantes non encore refermées). Ce dispositif connu sous le nom de PRETTY-READ permet de contrôler immédiatemment le niveau d'imbrication des parenthèses.

De plus un éditeur de ligne minimum est activé qui interprète les caractères suivants :

RUBOUT ou DELETE ou BACKSPACE : détruit le dernier caractère entré.

^U ou ^X : détruit toute la ligne entrée

Il existe dans le système Le_Lisp un véritable éditeur très puissant et extensible, Emacs, inclu dans le système d'une façon standard qui permet d'éditer des fichiers disques ou des fonctions précédemment chargées. Cet éditeur, qui permet d'éditer n'importe quel type de fichier, connait également la syntaxe et la sémantique de Lisp et apporte de ce fait une très grande aide pour l'ecriture et la mise-au-point des fonctions Lisp. La description de cet éditeur et de la manière d'écrire des extensions est donnée dans le troisième volume de ce manuel.

4.2.2 La lecture standard

La lecture des S-expressions Lisp par la fonction READ s'effectue en format LIBRE i.e. que chaque élément syntaxique peut être encadré d'un ou plusieurs espaces.

Durant la lecture des symboles seuls les 62 premiers caractères sont pris en compte. Les caractères de contrôle (i.e. les caractères entrés en pressant simultanément la touche CTRL et le caractère) sont imprimés à la DEC (i.e. ^ suivi du caractère). S'il faut insérer des caractères spéciaux dans un symbole, il faut soit les faire précéder du caractère spécial quote-caractère / (le slash par défaut) soit encadrer tous les caractères du symbole du caractère délimiteur de symboles (la barre de valeur absolue | par défaut).

STATUS-READ-CASE [Variable]

cette variable permet de contrôler la conversion automatique des caractères minuscules en leur équivalent majuscule. Si la valeur de cette variable est NIL la conversion est automatique, si elle est différente de NIL aucune conversion n'est réalisée.

```
ex : CaR                        correspond a l'atome   CAR
     LONGTRESLONGATOME             "       "           LONGTRESLONGATOME
     RE/(3/)                      "       "          RE(3)
     kFoo Bar|                     "       "          Foo Bar

     ; mais si on lit de la maniere suivante :

     (LET ((STATUS-READ-CASE T))
          (READ))

     CdR                        correspond a l'atome      CdR
     kAch|                         "       "              Ach
```

Les chaînes de caractères s'écrivent encadrées du caractère délimiteur de chaîne (par défaut le guillemet "). Si ce caractère doit être introduit dans une chaîne, il doit être doublé. N'importe quel caractère peut faire partie d'une chaîne en particulier les caractères de mouvement de papier, Return, Line-feed ... Une chaîne de peut dépasser 256 caractères actuellement. Il n'y a jamais de transcodage minuscule majuscule durant la lecture des chaînes.

Les nombres entiers sont représentés par une suite de chiffres décimaux qui peut être précédée du signe + ou du signe -.

Il existe d'autres manières de rentrer des nombres en particulier grâce aux SHARP-MACRO qui permettent de décrire des nombres dans n'importe quelle base, ou de rentrer des codes internes de caractères ASCII.

```
ex :  123          correspond au nombre     123
      +27             "        "           27
      -45             "        "           -45
      +               n'est pas un nombre
```

Il est possible d'insérer des commentaires, qui sont totalement ignorés au cours de la lecture. Un commentaire est une suite de caractères quelconques précédée d'un caractère de type début de commentaire et terminée par le caractère de type fin de commentaires. Par défaut il n'y a qu'un caractère de type début de commentaires, le caractère point-virgule (;) et qu'un seul caracrère fin de commentaires le caractère Return. Par défaut tout commentaire s'arrêtera donc en fin de ligne.

Compatibilité : les commentaires à la Vlisp (i.e. terminés par le caractère début de commentaires) ne sont pas autorisés.

```
ex : (DE FOO (N              ; le nombre
             L)              ; la liste
        (IF (= N 0)          ; plus rien a faire
            ...
```

La représentation des listes est classique : une liste se représente par une parenthèse ouvrante "(" suivie des éléments de la liste séparés par des espaces et suivis d'une parenthèse fermante ")". Il est possible également d'utiliser la notation pointée généralisée :

```
        ( S-expr    .    S-expr )

   ex :   (A . (B . (C . D))) correspond a (A B C . D)
          ((A) . (B))              "      "   ((A) B)
```

Au début d'une expression lue par la fonction READ, il peut y avoir un nombre quelconque de parenthèses fermantes qui sont ignorées. Ceci permet de refermer à coup sûr les S-expressions avec une giclée de parenthèses fermantes sans avoir à les dénombrer.

```
(DE FOO (N) (ADD1 N))))))    ; est lu sans erreur
```

4.2.3 Type des caractères

L'analyseur lexical Le_Lisp (i.e. la fonction READ) utilise une *table de lecture* pour effectuer commodément son analyse. Cette table associe un type codé à chacun des caractères.

Cette table de lecture est totalement accessible à l'utilisateur qui peut ainsi la changer pour pouvoir lire facilement de nouveaux dialectes de LISP aux syntaxes étranges.

Les types de caractères disponibles sont les suivants :

0 : type NULL. Tous les caractères de ce type sont complètement ignorés à la lecture (Ex: le caractère Line/Feed, le caractère *Avance-bande* ou NULL ...).

1 : type BCOM. Ce type de caractère sert à indiquer le début d'un commentaire qui sera terminé à l'occurrence d'un caractère du type suivant. Par défaut il n'existe qu'un seul caractère de ce type, le caractère point virgule ; . éilùùùù

2 : type ECOM. Ce type de caractère sert à indiquer la fin d'un commentaire. Par défaut
il n'existe qu'un seul caractère de ce type, le caractère Return

3 : type QUOTEC. Ce type de caractère est utilisé pour *quoter* n'importe quel autre caractère. *Quoter* un caractère consiste à lui donner implicitement le type 12 (le type des caractères normaux). Par défaut il n'existe qu'un seul caractère de ce type, le caractère "slash" / .

4 : type LPAR. Ce type de caractère (la parenthèse ouvrante (par défaut) sert de caactère de début de liste.

5 : type RPAR. Ce type de caractère (la parenthèse fermante) par défaut) sert de caractère de fin de liste.

8 : type DOT. Ce type de caractère (le point . par défaut) sert à écrire les paires pointées.

9 : type SEP. Définit un caractère séparateur standard (Ex: l'espace, la tabulation ...).

10 : type MACH. Ce type indique que le caractère est un macro-caractère. Une fonction est associée à l'atome dont le P-NAME est ce caractère. Cette fonction est invoquée automatiquement à la lecture de ce caractère dans le flux d'entrée (voir la section suivante).

11 : type CSTRING. Ce type de caractère fait office de caractère délimiteur de chaîne de caractères. Par défaut il n'existe qu'un seul caractère de ce type, le caractère guillemets " .

12 : type REGULAR. Définit un caractère normal pouvant être utilisé pour construire un P-NAME.

13 : type CSYMB. Ce type de caractère indique le délimiteur de symboles spéciaux. Par défaut il n'y a qu'un seul cacatère de ce type, le caractère valeur absolue.

14 : MONOSYMB. Ce type indique que le caratère doit être lu comme un symbole mono-caractère et qu'il n'a pas besoin de délimiteur de symbole.

(TYPECH <ch> <n>) [SUBR à 1 ou 2 arguments]

permet de connaître (si <n> n'est pas fourni) ou de modifier (si <n> est fourni) le type du caractère <ch>. Cette fonction retourne le type courant du caractère <ch> après modification éventuelle.

```
ex :  ; apres la redefinition syntaxique des
      ; caracteres I et 0,

         (TYPECH '< 4)   ->  4
         (TYPECH '> 5)   ->  5

      ; l'entree :

         ICONS '(A> '<B)>

      ; est lue :

         (CONS '(A) '(B))
```

(TYPECN <cn> <n>) [SUBR à 1 ou 2 arguments]

est identique à la fonction précédente, mais le caractère est spécifié sous la forme de son code ASCII.

Voici la table des caractères standards :

```
0     0       0                40     64     @    12
      1  ■  A  12               41     65     A    12
      2  ■  B  12               42     66     B    12
      3  ■  C  12               43     67     C    12
      4  ■  D  12               44     68     D    12
      5  ■  E  12               45     69     E    12
      6  ■  F  12               46     70     F    12
      7  bell 12                47     71     G    12
      8   bs  9                48     72     H    12
      9  tab  9                49     73     I    12
      10  lf  9                4A     74     J    12
      11  vt  9                4B     75     K    12
      12  ff  9                4C     76     L    12
      13  cr  2                4D     77     M    12
      14 ■  N  12               4E     78     N    12
      15 ■  O  12               4F     79     O    12
0     16 ■  P  12               50     80     P    12
1     17 ■  Q  12               51     81     Q    12
```

12	18	■	R 12	52	82	R	12
3	19	■	S 12	53	83	S	12
4	20	■	T 12	54	84	T	12
5	21	■	U 12	55	85	U	12
6	22	■	V 12	56	86	V	12
7	23	■	W 12	57	87	W	12
8	24	■	X 12	58	88	X	12
9	25	■	Y 12	59	89	Y	12
A	26	■	Z 12	5A	90	Z	12
B	27		12	5B	91	■	12
C	28		12	5C	92		12
D	29		12	5D	93	$	12
E	30		12	5E	94	k	12
F	31		12	5F	95	_	12
0	32		12	60	96		12
1	33		12	61	97	a	12
2	34	"	11	62	98	b	12
3	35		12	63	99	c	12
4	36	$	12	64	100	d	12
5	K37	%	12	65	101	e	12
6	38	&	12	66	102	f	12
7	39	'	10	67	103	g	12
8	40	(	4	68	104	h	12
9	41	)	5	69	105	i	12
A	42	*	12	6A	106	j	12
B	43	+	12	6B	107	k	12
C	44	,	12	6C	108	l	12
D	45	-	12	6D	109	m	12
E	46	.	8	6E	110	n	12
F	47	!	3	6F	111	o	12
0	48	0	12	70	112	p	12
1	49	1	12	71	113	q	12
2	50	2	12	72	114	r	12
3	51	3	12	73	115	s	12
4	52	4	12	74	116	t	12
5	53	5	12	75	117	u	12
6	54	6	12	76	118	v	12
7	55	7	12	77	119	w	12
8	56	8	12	78	120	x	12
9	57	9	12	79	121	y	12
A	58	:	12	7A	122	z	12
B	59	;	1	7B	123	K	12
C	60	I	12	7C	124	k	13
D	61	=	12	7D	125	L	12
E	62	0	12	7E	126	_	12
F	63	?	12	7F	127	del	0

4.2.4 Les macro-caractères

Un macro-caractère est un caractère quelconque auquel a été associé une fonction qui est lancée automatiquement à la lecture de ce caractère dans le flux d'entrée. La valeur retournée par cette fonction remplace le macro-caractère lu. Tout caractère peut être utilisé comme macro-caractère.

(DMC <ch> <l> <s1> ... <sN>) **[FSUBR]**

l'argument <ch> doit être un symbole mono-caractère. DMC associe à ce caractère une fonction qui possède une liste de variables locales <l> (cette liste est obligatoire à cette position même s'il n'y a pas de variables locales) et un corps de fonction <s1> ... <sN>. DMC possède la même syntaxe que les autres fonctions de définition (DE, DF et DM) et retourne <ch> en valeur.

DMC peut eˆtre de'fini en Lisp de la manie` re suivante :

```
(DF DMC (L)
    (E VAL (CONS 'DE L))        ; fabrique la fonction associee
    (TYPECH (CAR L) 10))        ; force le nouveau type de ce caractere
    (CAR L))                    ; et retourne en valeur le caractere.
```

Pour détruire une définition de macro-caractère, il faut changer le type du caractère et détruire la fonction qui lui était associée par exemple au moyen de la fonction suivante :

```
(DE REMACH (ch)                 ; [pour  REMove MAcro CHaracter]
    (TYPECH ch 12)              ; le caractere redevient normal
    ch))
```

ATTENTION : la fonction associée à un macro-caractère étant de la même nature que les fonctions définies par l'utilisateur (de type DE) il n'est pas possible, pour un atome mono-caractère, de posséder tout à la fois une définition de type DE et une définition de type DMC.

Il existe 4 macro-caractères standards :

- la macro quote '
- la macro load ^L
- la macro sharp #
- la macro backquote `

4.2.4.1 le macro caractère quote

le quote (apostrophe) ' est le plus connu et le plus utilisé des macro-caractères. Placé devant une S-expression quelconque, il retourne la liste (QUOTE S-expression).

```
ex :  '(A B)   est equivalent a     (QUOTE (A B))
      ''A              "            (QUOTE (QUOTE A))
```

4.2.4.2 le macro caractère ⁉L

placé devant un symbole, fabrique l'appel : (load <symbole>.LL) Il permet donc de charger un fichier Lisp interactivement d'une manière extrêmement concise. ^L imprime sur le flux de sortie courant le nom du fichier en cours de chargement.

4.2.4.3 le macro caractère ᵉ

ce macro caractère permet de convertir des nombres ou d'appeler l'interprète en cours d'une lecture normale. Son fonctionnement est très général et peut s'expliquer comme suit : à l'occurrence d'un caractère # dans le flux d'entrée, le lecteur Lisp lit le caractère suivant d'une manière analogue à la fonction READCH. Ce caractère lu, appelé sélecteur de sharp-macro, doit posséder sur sa P-list une fonction sous l'indicateur SHARP-MACRO. Cette fonction est lançée sans argument et c'est la valeur retournée par cet appel qui remplace le macro-caractère #. Il est donc possible de définir de nouvelles sharp-macros.

(SHARP-DMC <ch> <lvar> <s1> ... <sN>) [FSUBR]

définit une nouvelle macro-sharp. <ch> est le nouveau caractère sélecteur, <lvar> est la liste des variables de la fonctions associée à cette macro et <s1> ... <sN. est le corps de cette fonction. SHARP-DMC possède donc la même syntaxe que la fonction DMC.

SHARP-DMC peut etre de'fini en Lisp de la manie` re suivante :

```
(DF SHARP-DMC (1)
    (PUTPROP (CAR 1) (CONS LAMBDA (CDR 1)) 'SHARP-MACRO)))
```

Il existe un certain nombre de sélecteurs standards :

/ retourne le code interne (ASCII) du caractère suivant lu sur le flux d'entrée. C'est le meilleur moyen connu à ce jour pour entrer des codes ASCII dans un programme. #/A remplace avantageusement 41 pour décrire le code interne du caractère A.

retourne la valeur associée au symbole suivant lu sur le flux d'entrée. La valeur associée à ce type de symbole se trouve sur la P-list de ce symbole sous l'indicateur SHARP-VALUE. Cette macro est souvent utilisée pour définir des codes internes de caractères non imprimables.

. évalue l'expression suivante du flux d'entrée. Cette valeur est retournée par la sharp-macro.

+ évalue l'expression suivante du flux d'entrée.Si cette valeur est vraie, la sharp-macro retourne la valeur de l'expression suivante du flux d'entrée sinon elle ne fait que la lire et retourne NIL. Très utile pour des lectures conditionnelles.

- le contraire de la macro précedente. Si la valeur de l'expression suivante du flux d'entrée est fausse (égale à NIL) l'expression qui suit encore est évaluée sinon elle est simplement lue sans être évaluée.

$ lit le nombre suivant en base seize.

% lit le nombre suivant en base deux.

Voici une description Lisp du fonctionnement de la SHARP-MACRO :

```
(DMC k#| ()
     (LET ((ch (READCH)))
          (LET ((f (GETPROP ch 'SHARP-MACRO)))
               (IF f
                   (APPLY f ())
                   (SYSERROR '|#| "selecteur inconnu" ch)))))

(SHARP-DMC k.| ()
    ; retourne la valeur d'une evaluation durant un READ
    (E VAL (READ)))

(SHARP-DMC k/| ()
    ; retourne le code ascii du caractere
    (READCN))

(SHARP-DMC k| ()
    ; retourne une valeur
    (LET ((l (READ)))
         (OR (GETPROP l 'SHARP- VALUE)
             (SYSERROR '|#| " : pas de valeur pour" l)))))

(MAPC  (LAMBDA (x y) (PUTPROP x y  'SHARP- VALUE))
       '(BELL  BS TAB LF CR EOF ESC SP RUBOUT)
       '(7     8  9   10 13 26  27  32 127)))))
```

```
(SHARP-DMC k^  | ()
    ; retourne la valeur d'un caractere control
    (LOGAND 31 (READCN))))

(SHARP-DMC k"| ()
    ; retourne la liste quotee des codes ASCII
    (LET ((l) (cn))
         (UNTILEXIT FINL
             (IF (= (SETQ cn (READCN)) #/")
                 (EXIT FINL (LIST QUOTE (NRE VERSE l)))
                 (NEWL l cn)))))))

(SHARP-DMC k+| ()
    ; lecture conditionnelle
    (IF (E VAL (READ)) (E VAL (READ)) (READ)))

(SHARP-DMC k-| ()
    ; lecture conditionnelle
    (IF (E VAL (READ)) (READ) (E VAL (READ))))
```

4.2.4.4 la macro backquote

cette macro sert à fabriquer des programmes qui construisent des listes. Les programmes synthétisés par ce macro-caractère utilisent les fonctions CONS, MCONS, LIST, APPEND et QUOTE.

```
? ? ? ? ? ? ?    A FINIR   ? ? ? ? ? ? ?

ex :  (A B C)        est equivalent a        '(A B C)
      (A ,B C)              ""               (LIST 'A B 'C)
      (,X . Y)              ""               (CONS X 'Y)
      (A B . ,C)            ""               (MCONS 'A 'B C)
      (X ,@Y ,@Z  V)         ""
                    (CONS 'X (APPEND Y (APPEND Z '( V))))
```

4.3 Les fonctions de sortie de base

Toutes éditent dans un tampon de sortie totalement accessible à l'utilisateur qui est vidé dans le flux de sortie associé soit à un terminal soit à un fichier sélecté au moyen de la fonction OUTPUT.
Si au cours d'une édition le tampon de sortie devient plein, il est automatiquement imprimé, en utilisant la fonction (TERPRI 1), et l'édition se poursuit dans un nouveau tampon vide.
On peut, à tout moment, suspendre une impression sur le terminal (holding) en entrant sur le terminal le caractère spécial ^W (propre à VERSAdos).

(PRIN <s1> ... <sN>) **[SUBR à N arguments]**

édite dans le tampon de sortie la valeur des différentes S-expressions <s1> ... <sN> sans imprimer le tampon. PRIN retourne en valeur la valeur de l'évaluation de <sN>.

(FLUSH) **[SUBR à 0 argument]**

imprime tous les caractères du tampon de sortie. Cette fonction n'insere aucun caractère propre dans le tampon de sortie.

```
ex : ? (DE QUAM (msg)
     ?     (PRIN msg)       ; edite le msg dans le tampon
     ?     (FLUSH)          ; puis imprime le contenu du tampon.
     ?     (READ))          ; enfin lecture de la reponse.
     = QUAM

     ? (QUAM "Nb de satellites de  VEGA")
       Nb de satellites de  VEGA ? xxx
                                  ; xxx est la reponse entree
     = xxx
```

(TERPRI <n>) **[SUBR à 0 ou 1 argument]**

imprime tous les caractères du tampon de sortie, se positionne en début de ligne en imprimant le code Return , saute <n> lignes en imprimant <n> fois le code Line-feed , puis édite un certain nombre d'espaces pour se positionner à la marge gauche (voir la fonction LMARGIN). Si l'argument <n> n'est pas un nombre ou n'est pas fourni, TERPRI agit comme si <n> était égal à 1 provoquant une impression avec interlignage simple. L'appel (TERPRI) est donc équivalent à (TERPRI 1).

(PRINT <s1> ... <sN>) **[SUBR à N arguments]**

édite dans le tampon de sortie les différentes S-expressions
<s1> ... <sN> qui sont évaluées par la fonction PRINT puis imprime ce tampon (en faisant un appel implicite à la fonction (TERPRI 1)). PRINT retourne en valeur la valeur de l'évaluation de <sN>.

PRINT peut être de'fini en Lisp de la manie` re suivante :

```
(DF PRINT (l)
    (PROGl (E VAL (CONS 'PRIN l)) (TERPRI l)))

ex : ? (PRINT (l+ 9) (CDR '(A B C)))    ; forme a evaluer
     10(B C)                            ; execution
   = (B C)                              ; valeur retournee
```

Compatibilité : en Maclisp, le retour chariot est réalisé AVANT l'impression, et Vlisp insère un espace entre chaque objet imprimé.

(PRINCH <ch> <n>) **[SUBR à 1 ou 2 arguments]**

édite <n> fois le caractère <ch>. Si <n> n'est pas un nombre ou est omis, le caractère <ch> n'est édité qu'une fois. PRINCH retourne le symbole ou le nombre <ch> en valeur.

```
ex : (PRINCH '| k 10)  ; edite  10 caracteres espace dans le tampon

     (DE PYR (n)
         (IF (<= n 0)
             ()
             (PRINCH '| k n)
             (PRINCH '|*| (1+ (* (- 4 n) 2)))
             (TERPRI)
             (PYR (1- n))))

     (PYR 4)

     produira

        *
       ***
      *****
     *******
```

(PRINCN <cn> <n>) **[SUBR à 1 ou 2 arguments]**

comme la fonction précédente, édite <n> fois le caractère <cn>. Si <n> est omis, le caractère <cn> n'est édité qu'une fois. PRINCN retourne le code ASCII <cn>.

```
     (LET ((cn #/A))
          (REPEAT 10
                  (PRINCN cn)
                  (SETQ cn (1+ cn))))
; produira

ABCDEFGHIJ
```

(SPACES <n>) [SUBR à 1 argument]

édite <n> fois le caractère espace (#SP). Retourne <n> en valeur.

4.4 Contrôle des fonctions de sortie

Les fonctions de sortie éditent les représentations externes des objets Le_Lisp dans un tampon de sortie accessible au moyen de fonctions spécialisées.

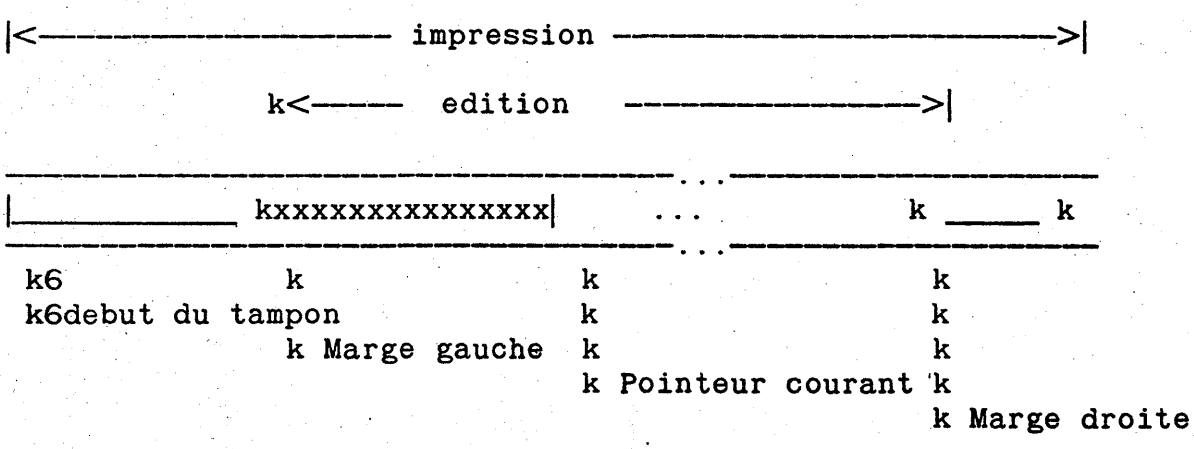
La plupart de ces fonctions vont permettre de contrôler un paramètre des fonctions de sortie et possèdent 0 ou 1 argument. Un appel sans argument est une demande de la valeur du paramètre (GET), tandis qu'un appel avec 1 argument provoque un changement du paramètre (SET). Le_Lisp utilise des fonctions (et non des variables) pour contrôler les paramètres car elles réalisent des tests de validité. En cas de changement temporaire d'un de ces paramètres il est commode de s'assurer de la restitution du paramétre modifié au moyen de la fonction UNWIND-PROTECT.

```
; si <fnt> est une fonction de controle de ce type, on realise un
; changement temporaire et protege de la maniere suivante :

   (let ((dummy (<fnt>)))
        (<fnt> -nouvelle valeur-)
        (unwind-protect
                -avec la nouvelle valeur de Ifnt>-
                (<fnt> dummy)))
```

4.4.1 Le tampon de sortie

Le tampon de sortie est organisé de la manière suivante :



Les éditions ne s'effectuent dans le tampon de sortie qu'entre la marge gauche et la marge droite alors que les impressions vont porter sur tout le tampon. L'accès aux valeurs de la marge gauche, du pointeur courant et de la marge droite est réalisé au moyen de fonctions spéciales ; il existe de plus une fonction OUTBUF permettant de manipuler directement n'importe quel caractère de ce tampon de sortie. Les fonctions qui vont suivre testent la validité de leur argument. Dans le cas où on sort du tampon de sortie une erreur apparait dont le libellé est :

** <fnt> : débordement du tampon de sortie : <n> ou bien
** <fnt> : output buffer overflow : <n>

dans lequel le nom de la fonction qui a provoqué l'erreur <fnt> est imprimé ainsi que l'argument défectueux <n>.

(LMARGIN <n>) [SUBR à 0 ou 1 argument]

permet de spécifier la marge à gauche <n> de l'impression. Par défaut à l'initialisation du système on a (LMARGIN 0). Si <n> n'est pas fourni, la marge n'est pas modifiée. LMARGIN retourne la valeur de la marge gauche courante et est principalement utilisée par le Pretty-print (Voir le 3ème volume du manuel) pour gérer automatiquement les renfoncements gauches à l'impression des structures de contrôle.

(RMARGIN <n>) [SUBR à 0 ou 1 argument]

permet de spécifier la marge à droite <n> de l'impression. Par défaut à l'initialisation du système on a (RMARGIN 72). Si <n> n'est pas fourni, la marge n'est pas modifiée. RMARGIN retourne la valeur de la marge droite courante et est utilisée principalement pour régler la taille des lignes en fonction du terminal de sortie utilisé (télétype, écran, imprimante ...).

(OUTPOS <n>) **[SUBR à 0 ou 1 argument]**

permet de spécifier, si <n> est fourni et est un nombre, la nouvelle valeur du pointeur courant sur le tampon de sortie. Cet index pointe toujours sur la 1ère position libre dans le tampon. OUTPOS retourne en valeur la position courante de ce pointeur.

(OUTBUF <n> <cn>) **[SUBR à 1 ou 2 arguments]**

considère le tampon de sortie comme un vecteur et permet donc d'avoir accès et/ou de modifier le <n>ième caractère du tampon de sortie avec le code interne <cn>, si l'argument <cn> est donné. OUTBUF retourne toujours la valeur du <n>ième caractère du tampon de sortie.

4.4.2 Limitations d'impression

Pour pouvoir limiter les impressions des structures très longues et rendre possible celles des listes circulaires deux nouvelles fonctions ont été ajoutées.

(PRINTLENGTH <n>) **[SUBR à 0 ou 1 argument]**

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) le nombre maximum d'éléments de liste qui est imprimé lors d'un PRINT ou d'un PRIN. Par défaut la valeur de ce nombre est de 2000. Si une liste contient un nombre d'éléments plus élevé, le reste des éléments ne sera pas imprimé et des points de suspension ... apparaitront devant la parenthèse fermante. Cette fonction permet de terminer l'impression des listes circulaires sur les CDR. PRINTLENGTH retourne en valeur la longeur maximum courante.

```
ex : (PRINTLENGTH 6)        -> 6
     '(1 2 3 4 5 6 7 8 9) -> (1 2 3 4 5 6...)
```

(PRINTLEVEL <n>) **[SUBR à 0 ou 1 argument]**

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) la profondeur maximum d'impression (i.e. le nombre maximum de parenthèses ouvrantes non fermées). En cas de dépassement, la liste qui provoque le débordement n'est pas imprimée et le caractère & est imprimé à sa place. Cette fonction permet d'imprimer des listes circulaires sur les CAR. PRINTLEVEL retourne en valeur la profondeur maximum courante.

```
ex : (PRINTLE VEL 3) -> 3

     '(DE FOO (L)
          (IF (NULL (CDR L)) L
              (FOO (CDR L))))
     ->  (DE FOO (L) (IF (NULL &) L (FOO &)))
```

```
; impression des listes circulaires ;

?  (PRINTLE VEL 10)
=  10
?  (PRINTLENGTH 50)
=  50
?  (SETQ L '(X Y Z))
=  (X Y Z)
?  (RPLACD (CDDR L) L)
=  (Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X
=   Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X...)
?  (RPLACA L L)
=  ((((((((((& Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z &
=  Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z &
=  Y...)...)...)...)...)...)...)...)...)...)
?
```

(PRINTLINE <n>) **[SUBR à 0 ou 1 argument]**

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) le nombre maximum de lignes (i.e. d'appels à la fonction TERPRI) qui est imprimé lors d'un PRINT ou un PRIN. Par défaut la valeur de ce nombre est 5000. Si une expression nécessite plus de ligne, la dernière ligne apparaîtra avec des points de suspension. PRINTLINE retourne en valeur le nombre maximum de lignes autorisé pour un PRINT.

4.4.3 Listes circulaires ou partagées

L'utilisation des fonctions de limitation d'impression du paragraphe précédent, pour autant qu'elles stoppent l'impression et permettent d'en avoir une vue abbrégée, ne permettent pas, en cas de structures circulaires ou partagées de connaître les cellules effectivement partagées.

Pour cela il faut utiliser les fonctions suivantes, écrites en Lisp qui utilisent la propriété de la fonction EQ (ici de la fonction MEMQ) suivante : (EQ l1 l2) est vraie si les 2 listes l1 et l2 sont les mêmes physiquement (ont la même adresse physique).

```
; Fonctions d'impressions circulaires simples :

(DE CPRINT (l)
    ; Fonction d'impression de l
    (CPRIN l)
    (TERPRI)
    T)

(DE CPRIN (l)
    ; Fonction d'edition de l
    (LET ((vus))
```

```
        ; vus contient la liste des objets deja visites
        (CPRIN1 l)))

(DE CPRIN1 (l)
    ; Fonction auxilaire
    (IF (ATOM l)
        ; Les atomes peuvent etre partages
        (PRIN l)
        ; En cas de liste
        (PRINCH '|(|)
        (WHILE (CONSP l)
               ; iteration sur les CDR
               (IF (MEMQ l vus)
                   ; c'est un objet deja visite
                   (SETQ l '|...|)
                   ; c'est un objet nouveau
                   (NEWL vus l)
                   ; recurse sur les CAR
                   (CPRIN1 (NEXTL l))
                   ; separation inter element d'une liste
                   (IF l (PRINCH '| k))))
      (IF (NULL l)
          ; La liste se termine bien
          ()
          ; C'est une paire pointee
          (PRIN '|. k l))
      ; Fermante finale
      (PRINCH '|)|))))))

; ainsi si     (SETQ LL '(A B C D))

; l'appel      (CPRINT (NCONC LL LL))

; produira     (A B C D . ...)
```

4.4.4 Edition standard

Les différents objets Lisp sont édités dans le tampon de sortie en respectant l'unique régle suivante : la représentation externe d'un objet atomique ne peut etre éditée à cheval sur deux lignes; ceci pour les symboles et les nombres.
L'impression de la fonction (QUOTE <s>) s'effectue '<s> (donc en restituant le macro-caractère standard d'entrée) pour des raisons de lisibilité.

deux variables STATUS- permettent de régler certaines modalités d'impression et la fonction PTYPE permet de manipuler aisément le mode d'impression de chaque symbole (le P-type).

STATUS-PRINT [Variable]

contient l'indicateur d'impression des P-NAME spéciaux et des chaînes de caractères. Si STATUS-PRINT = T, les symboles spéciaux sont encadrés du caractère | à l'impression, de même les chaînes de caractères sont également encadrées du caractère ".

STATUS-PRINT-CASE [Variable]

contient l'indicateur de casse en sortie. Si STATUS-PRINT-CASE = NIL, les symboles normaux sont imprimés en majuscule. Si STATUS-PRINT-CASE = T, ces symboles sont imprimés en minuscule.

(PTYPE <sym> <n>) [SUBR à 1 ou 2 arguments]

permet de modifier (si l'argument <n> est fourni) ou de consulter (si l'argument <n> n'est pas fourni) la valeur du P-TYPE du symbole <sym>. Ce P-TYPE est principalement utilisé par le PRETTY-PRINT pour y stocker le format à utiliser pour éditer cet atome symbole en tant que fonction. PTYPE retourne la valeur courante du P-TYPE du symbole <sym>.

4.4.5 Interruption fin de ligne

quand le système décide de vider le tampon de sortie, implicitement si un atome ne rentre pas dans le tampon, ou explicitement par appel de la fonction TERPRI, il déclenche une interruption programmée de nom EOL. En temps normal, cette fonction vide le tampon de sortie dans le flux de sortie courant et efface tout le tampon mais il est parfois souhaitable de prendre le contrôle de cette fonction pour copier le tampon sur plusieurs flux, le recopier dans la mémoire, transformer certains caractères du tampon ... Toutes ces actions vont être réalisables en redéfinissant la fonction EOL.

(EOL) [SUBR à 0 argument]

vide le tampon de sortie dans le flux de sortie courant. Cette fonction est appelée automatiquement par le système et n'existe en tant que fonction que pour être redéfinie (interruptions programmée).

(STREAM-OUTPUT <s1> ... <sN>) [FSUBR]

permet de récupérer sous forme de liste de listes de caractères toutes les impressions produitent durant le temps de l'évaluation des expressions <s1> ... <sN>. Chaque tampon vidé produit une liste de caractères et STREAM-OUTPUT retourne une liste de ces lignes. Les tampons ne sont pas copiés dans le flux de sortie. Cette fonction est extrêmement utile en particulier pour l'éditeur Emacs qui peut ainsi récupérer tout ce qui peut être imprimé en vue de son édition.

STREAM-OUTPUT peut être défini en Lisp de la manière suivante :

```
(SYNONYM 'STD-EOL 'EOL)

(DE NEW-EOL ()
    ; realise le nouveau EOL
    (LET ((soi 0) (sor))
         ; recupere la liste des caracteres du tampon
         (REPEAT (OUTPOS)
                 (NEWL sor (OUTBUF soi))
                 (OUTBUF soi #SP)
                 (INCR soi))
         ; reconstruit la marge gauche
         (OUTPOS 0)
         (WHILE (< (OUTPOS) (LMARGIN))
                (OUTBUF (OUTPOS) #SP)
                (OUTPOS (1+ (OUTPOS))))
         ; fabrique la liste des listes de caracteres
         (NEWL solo (NRE VERSE sor))))

(DF STREAM-OUTPUT (sol solo)
    ; solo sera la liste des listes.
    (SYNONYM 'eol 'new-eol)
    (UNWIND-PROTECT
          (EPROGN sol)
```

```
                (SYNONYM 'eol 'std-eol))
            (NRE VERSE solo)))))))
```

4.5 Les fonction sur les flux d'entrée/sortie

Dans l'état minimum du système, Le_Lisp peut gérer simultanément
- un terminal d'entrée/sortie
- un fichier d'entrée
- un fichier de sortie
- un fichier d'entrée auxiliaire.

Une spécification de fichier <file> est en Lisp un symbole qui utilise la même syntaxe que le système hôte (ici VERSAdos).

```
sous  VERSAdos       cccccccc.ffffffff.ee
```

cccccccc est le nom d'un catalogue (au maximum 8 caractères), ffffffff est le nom du fichier (au maximum 8 caractères), ee est le nom de l'extension (au maximum 2 caractères).

Il est recommandé d'utiliser les extensions suivantes :

```
sous  VERSAdos
                .LL       pour les fichiers source Le_Lisp
                .SA       pour les entrees/sorties non Lisp
```

(FILENAMEP <sym>) [SUBR à 1 argument]

<sym> doit être un symbole représentant un nom de fichier. Après avoir converti tous les caractères minuscules de <sym> en leur équivalent majuscules, FILENAMEP teste si ce symbole est un nom de fichier correct pour le système hôte. Il retourne <sym>, augmenté des valeurs par défaut des différents champs, si la syntaxe de ce nom de fichier est correcte et NIL dans le cas contraire. VERSAdos étant très susceptible sur ces petites choses, nous vous conseillons très fortement de systématiquement tester tout nom de fichier avant de les utiliser.

```
ex :  (FILENAMEP 'foo)          ->  FOO.LL
      (FILENAMEP 'exll.top.)    ->  EXLL.TOP.LL
      (FILENAMEP 'f.lisp)       ->  NIL

          ; voici le bon moyen d'ouvrir un fichier

      (INPUT (OR (FILENAMEP file) (SYSERROR 'FILENAMEP "quezako"!
file)))
```

4.5.1 La sélection des flux

Un canal <chan> est un numéro associé à tout flux ouvert. Il est utilisé pour connecter un flux ouvert au flux courant d'entrée ou de sortie.

Toutes les fonctions de cette section vont utiliser le système de gestion de fichiers du système hôte (ici VERSAdos) qui peut retourner un code erreur en cas d'anomalie. Si c'est le cas, une erreur Lisp se déclenche dont le libellé est :

** <fnt> : _ erreur_ d'entrée_ sortie : <n> ou bien
** <fnt> : _ I/O error : <n>

dans lequel <fnt> est le nom de la fonction Lisp qui était appelée et <n> le code erreur retourné par le système de gestion de fichier. Les erreurs les plus fréquentes de VERSAdos sont :

4	le volume n'est pas monté
7	le fichier est protégé
10	plus de place sur le disque
11	le fichier est occupé
12	plus de place en mémoire centrale
24	le fichier n'existe pas

et les erreurs détectées par Le_Lisp sont :

-1	plus de canaux disponibles
-2	numéro de canal incorrect
-3	canal non ouvert

(OPENF IN/OUT/APPEND <file> <eofv>) [FSUBR]

permet d'ouvrir en fichier et retourne le numéro de canal <chan> affecté à ce fichier. Le 1er argument n'est jamais évalué et indique le type du fichier :

IN pour un fichier en entrée
OUT pour un nouveau fichier de sortie
APPEND pour un ancien fichier en sortie

le second argument <file>, évalué, est le nom du fichier. Si le fichier est un fichier d'entrée (de type IN), le troisième argument indique la forme à évaluer en cas de fin de fichier (c'est la valeur retournée par cette évaluation qui est retournée à la fonction de lecture).

```
ex : (OPENF IN "FOO.LL" '(PROGN (PRINT "EOF sur FOO") EOF VAL))
```

Ouvre le fichier de nom FOO.LL en lecture, en cas de fin de fichier la forme (PROGN (PRINT "EOF sur FOO") EOFVAL) est évaluée, elle imprime le message "EOF sur FOO" et retourne comme valeur de la lecture la valeur du symbole EOFVAL.

(INCHAN) [SUBR à 0 argument]

retourne le numéro du canal associé au flux d'entrée.

(OUTCHAN) [SUBR à 0 argument]

retourne le numéro du canal associé au flux de sortie.

(INPUT <flux>) [SUBR à 1 argument]

cette fonction permet de sélecter le flux d'entrée (la fonction précédente ne faisant que l'ouvrir).
<flux>. Ce flux peut être :
NIL, il s'agit alors du terminal.
<chan>, il s'agit alors d'un canal précédemment ouvert avec la fonction OPENF

Si la sélection est possible, INPUT retourne <chan> ou T, mais si la sélection ne l'est pas INPUT retourne NIL.

ex : voici une fonction de chargement d'un fichier

```
(DE LOAD (file)
    (LET ((oldinput (INCHAN)))
         (INPUT (OPENF IN file '(EXIT EOF (INPUT oldinput))))
         (UNTILEXIT EOF (ERRSET (E VAL (READ)) T))))
```

EOFVAL [Variable]

(OUTPUT <flux>) [SUBR à 1 argument]

cette fonction permet de sélecter le flux de sortie
<flux>. Ce flux peut être :
NIL, il s'agit alors du terminal.
<file>, il s'agit alors d'un fichier sur disque.

Si durant l'ouverture du flux, une erreur se produit, le fichier en question est automatiquement fermé, le terminal est ouvert à sa place et le message suivant apparait :

** OUTPUT :_ erreur_ d'entrée/sortie : <n> ou bien
** OUTPUT :_ I/O error : <n>

dans lequel <n> est le numéro d'erreur retourné par VERSAdos. De même s'il n'est pas possible d'ouvrir le fichier <file> en sortie (il n'y a plus de place, le périphérique n'est pas disponible ou pour toute autre raison ...) cette fonction retourne NIL.
Enfin il est possible de déclencher avec cette fonction (ou avec les écritures qui vont suivre) des erreurs du système de gestion de disque lui-même.

ex : de manipulation de fichiers

```
; fonction COPYFILE qui copie entierement le fichier
; d'entree <filin> dans le fichier de sortie <filout>
```

```
  (DE COPYFILE (<filin> Ifilout>)
      (OR (INPUT Ifilin>) (SYSERROR 'COPYFILE "Il n'y a pas" Ifi!
lin>)))
      (OR (OUTPUT Ifilout>)
          (PROGN (INPUT) (SYSERROR "Probleme avec : " Ifilout>))))
      (LET ((nbsexp 0) (lu))
           (UNTIL (EQ (SETQ lu (READ)) STATUS-EOF VAL)
                  (INCR nbsexp)
                  (PRINT (READ))))
      (INPUT)
      (OUTPUT)
      (PRINT "Copie de : " Ifilin> " sur " Ifilout>
          "  " nbsexp " S-expressions ecrites.")
      'OK))

? (COPYFILE 'JC.FILE.LL 'OLD.FILE.LL)
  Cpie de : JC.FILE.LL sur OLD.FILE.LL  7 S-expressions ecrites.
= OK
```

4.5.2 Les fonctions sur les fichiers

(CLOSE <chan>) **[SUBR à 1 argument]**

ferme le canal de numéro <chan>. CLOSE retourne toujours T.

(RENAMEFILE <ofile> <nfile>) **[SUBR à 2 arguments]**

permet de changer le nom du fichier <ofile> par <nfile>. RENAMEFILE retourne toujours T.

(DELETEFILE <file>) **[SUBR à 1 argument]**

permet de détruire le fichier <file>. DELETEFILE retourne toujours T en valeur.

4.5.3 La fonction LOAD et le mode AUTOLOAD

(LOAD <file>) **[FSUBR]**

permet de charger (en évaluant toutes les expressions qui s'y trouvent) le fichier de nom <file>. LOAD retourne T en valeur.

(LIBRARY <file>) **[SUBR à 1 argument]**

est identique à la fonction précédente mais le nom du fichier <file> est évalué. LIBRARY retourne toujours T en valeur.

(AUTOLOAD <file> <sym1> ... <symN>) **[FSUBR]** Comme il est fastidieux de charger à la main plusieurs fichiers de fonctions à chaque exécution Le_Lisp permet l'utilisation de fonctions autoloads (qui se chargent toutes seules dynamiquement à leur premier appel).

AUTOLOAD peut etre de'fini en Lisp de la manie`re suivante :

```
(DF AUTOLOAD (l)
    ; (AUTOLOAD fichier at1 ... atN)
    (MAPC (LAMBDA (at)
              (E VAL (LIST 'DM at '(l)
                           (LIST 'MAKUNBOUNDFN (LIST QUOTE at))
                           (LIST 'LIBRARY (CAR l))
                           'l)))
      (CDR l)))
```

Ainsi l'appel :

```
(AUTOLOAD "PRETTY.LL" PRETTY)
```

donne la définition suivante à la fonction PRETTY :

```
(DM PRETTY (l)
    (MAKUNBOUNDFN 'PRETTY)
    (LIBRARY "PRETTY.LL")
    l)
```

ce qui fait qu'à la première évaluation de (PRETTY ...) la MACRO est appelée. Cette MACRO se détruit elle-même (pour éviter de boucler au cas ou la fonction ne serait pas définie) puis évalue (LIBRARY "PRETTY.LL") qui charge en silence le fichier "PRETTY.LL".
La valeur retournée par la MACRO étant l'appel originel (PRETTY ...) lui-même, EVAL reévalue cette forme dans laquelle la fonction PRETTY est maintenant définie (cette nouvelle définition se trouvait dans le fichier). Ouf.

4.6 Les fonctions spéciales sur terminaux

Un ensemble de fonctions permettent de travailler directement sur le terminal (TTY) pour utiliser au mieux toutes ses possibilités. Ces fonctions n'utilisent pas le système d'entrées-sorties classique et peuvent donc être utilisées à tout moment.

Ces fonctions sont utilisables sur n'importe quel terminal (en particulier sur tous les terminaux papier et à écran).

? ? ? Emacs ? ? ? Buffer TTY ? ? ? ?

(TYI) [SUBR à 0 argument]

retourne un nombre représentant le code interne du caractère suivant lu sur la TTY, sans aucune conversion et sans utiliser l'éditeur de ligne du moniteur (en particulier les caractères RUB-OUT, et les différents control-caractères ne sont plus effectifs).

(TYS) [SUBR à 0 argument]

si un caractère est prêt à être lu sur le terminal, retourne la valeur que retournerait la fonction TYI, sinon si aucun caractère n'est prêt TYS retourne NIL. Cette fonction permet de tester si un caractère a été frappé sur le terminal. ATTENTION : si un caractère est effectivement retourné il est réellement lu Problème : cette fonction ne fonctionne toujours pas sur l'EXORmacs (merci VERSAdos).

Donc (UNTIL (TYS)) est bien équivalent à (TYI).

(TYO <cn>) [SUBR à 1 argument]

imprime sur le terminal le caractère de code interne <cn>. Il n'y pas de conversion de caractère au moment de la sortie (par ex: les caractères contrôles sont envoyés directement sans passer par la forme ^x). Il sera donc possible d'envoyer au moyen de cette fonction des caractères de contrôle (comme le déplacement du curseur) propre à chaque terminal.

(TYOL <lcn>) [SUBR à 1 argument]

imprime sur le terminal la liste des caractères <lcn>. Cette fonction s'utilise de la même manière que la fonction précédente.

TYOL peut etre de'fini en Lisp de la manie`re suivante :

```
(de tyol (lcn)
    (mapc 'tyo lcn))

(tyol #"Foo Bar")

; enverra sur le terminal la liste des codes internes correspon!
dant
;aux caracteres ; Foo Bar
```

(TYFLUSH) [SUBR à 0 argument]

4.7 Les systèmes COLORIX

COLORIX est un système de visualisation couleur réalisé par Louis Audoire (1). Il existe 2 systèmes COLORIX :
- le système COLORIX 80
- le système COLORIX 82 (dit COLORY)

4.7.1 Le système COLORIX 80

Le système COLORIX 80 est une version améliorée du système COLORIX 75. L'écran de COLORIX 80 est composé de 360 colonnes de 256 lignes chacunes. Chaque point de cet écran peut recevoir une couleur codée sur 12 bits (4 bits pour chaque couleur primaire). Il y a donc 4096 (2^12) couleurs possibles. Dans toutes ces fonctions les arguments <r>, <b>, <v> représentent les 3 couleurs primaires (qui sont calculées modulo 16), l'argument <c> représente une couleur complète codée sur 12 bits, les arguments <x>, <y> représentent des coordonnées (prises modulo 512).

(COLORIX <n1> <n2>) [SUBR à 2 arguments]

charge le registre interne de COLORIX de numéro <n1> avec la valeur <n2>.

```
ex : (COLORIX 6 0)          ; efface l'ecran visible
     (COLORIX 6 #$F00))     ; rempli l'ecran de rouge
     (COLORIX 10 0)         ; passe en mode pleine page
```

(COLOR <r> <v> <b>) [SUBR à 3 arguments]

(COLORHSV <h> <s> <v>) [SUBR à 3 arguments]

(COLPIX <x> <y> <c>) [SUBR à 3 arguments]

(COLBOX <x> <y> <w> <h> <c>) [SUBR à N arguments]

(COLSHAPE <x> <y> <w> <h> <c>) [SUBR à N arguments]

(COLVECTOR <x1> <y1> <x2> <y2> <c>) [SUBR à N arguments]

(COLTEXT <sym> <x> <y> <c> <z>) [SUBR à N arguments]

(COLFONT <e> <x> <y> <c> <f> <zx> <zy>)

(COLMEM <x> <y>) [FSUBR]

retourne la couleur actuelle (sous la forme d'un nombre de 12 bits) du point de coordonnées <x> et <y>. Cette dernière fonction permet de lire la mémoire de rafraîchissement de COLORIX 80. Pas vrai Loulou ?

4.7.2 Le système COLORY

(COLYCLEAR) [SUBR à 0 argument]

(COLYTC <i> <n>) [SUBR à 2 arguments]

(COLYFLUSH <x> <y> <w> <h> <m>) [SUBR à N arguments]

(COLYADD <x> <y> <w> <h> <m>) [SUBR à N arguments]

(COLYREM <x> <y> <w> <h> <m>) [SUBR à N arguments]

(COLYZOOM <n>) [SUBR à 1 argument]

(COLYPAN <x> <y>) [SUBR à 2 arguments]

(COLYFLIP <n>) [SUBR à 1 argument]

4.8 La tablette graphique

- précision 1/10 mm
- 2794 x 2794
- org ortho-normé
- retourne le code d'erreur (0 = OK)

(OPENPEN) **[SUBR à 0 argument]**

(INITPEN <n>) **[SUBR à 1 argumnent]**

sélecte le mode de fonctionnement de la tablette. Si <n> = #/O, mode poursuite à 105 échantillonnage par seconde. Si <n> = #/S, arrêt du mode échantillonnage.

(WAITPEN) **[SUBR à 0 argument]**

attend l'arrivée d'un échantillonnage.

(GETXPEN) **[SUBR à 0 argument]**

(GETYPEN) **[SUBR à 0 argument]**

(GETFPEN) **[SUBR à 0 argument]**

CHAPITRE 5

LES FONCTIONS SYSTèME

5.1 Appel et sortie de l'interprète

(LELISP <n> <file>) **[SUBR à 2 arguments]**

permet d'appeler récursivement (mais hélas sans retour possible avec le système VERSAdos) le système Le_Lisp 68K. Le 1er argument est la taille de la mémoire à utiliser et le second est le nom du fichier qui doit être chargé avant de rentrer dans la boucle interactive du système.

(LELISP-SIZE) **[SUBR à 0 argument]**

retourne la taille de la mémoire du système courant. Cette taille est celle fournie au dernier appel, via VERSAdos (au moyen de la commande =lelisp)
ou Le_Lisp lui-même (au moyen de la fonction précédente).

(END) **[SUBR à 0 argument]**

arrête l'évaluation en cours, ferme tous les fichiers ouverts, sort de l'interprète, ne passe pas par la case départ et rend le contrôle au moniteur. END est utilisée pour sortir définitivement de l'interprète Le_Lisp et pour revenir au moniteur standard. END est la seule fonction standard qui ne retourne pas de valeur. QUIT pour des raisons de compatibilité avec Maclisp est un synonyme de END.

5.2 Le TOP-LEVEL

La boucle principale (ou TOP-LEVEL) de l'interprète consiste à évaluer indéfiniment la forme :

(WHILE T (TOPLEVEL))

C'est donc la fonction TOPLEVEL qui détermine le mode de fonctionnement de l'interprète. Cette fonction (de type SUBR) est bien évidemment redéfinissable par l'utilisateur qui désire se construire son propre système. Il existe de plus un STATUS qui permet de contrôler le fonctionnement de la fonction TOPLEVEL standard et deux variables globales.

(TOPLEVEL) [SUBR à 0 argument]

Cette fonction va, d'une manière standard :
- lire une S-expression dans le flux d'entrée courant
- évaluer cette S-expression
- imprimer le résultat de cette évaluation dans le flux de sortie courant

STATUS-TOPLEVEL [Variable globale]

cette variable contient l'indicateur standard du toplevel. Si cet indicateur est faux, la fonction TOPLEVEL n'imprime pas les résultats des évaluations, dans le cas contraire TOPLEVEL imprime la valeur retournée par chacune des évaluations.

\+ **[Variable globale]**

cette variable contient la forme qui est en cours d'évaluation par la fonction TOPLEVEL.

\- **[Variable globale]**

cette variable contient la dernière forme qui a été évaluée par la fonction TOPLEVEL.

* **[Variable globale]**

cette variable contient la valeur de la dernière forme évaluée par la fonction TOPLEVEL.

A l'initialisation du système la fonction TOPLEVEL est équivalente à :

```
(DE TOPLE VEL ()
    (SETQ  - +
           + (READ)
           * (E VAL +))
    (WHEN STATUS-TOPLE VEL (PRINT "= " *)))
```

ATTENTION : une redéfinition de cette fonction, qui n'évalue pas des formes lues, rend tout le système Le_Lisp inutilisable.

exemple de définition à éviter :

```
(DE TOPLEVEL () (READ) (PRINT 'MARRE))
```

5.3 Le fichier initial

A l'initialisation des systèmes Le_Lisp utilisant des fichiers, un fichier initial est lu avant de passer en mode conversationnel sur le terminal. Ce fichier se nomme :
- STARTUP.LL dans le système VERSAdos.

Ce fichier contiendra les définitions de EXPR/FEXPR/MACRO que l'utilisateur désire posséder à chaque appel de Le_Lisp.

5.4 Les fichiers image-mémoire

(SAVE-CORE <file> <s>) **[SUBR à 2 arguments]**

(RESTORE-CORE <file>) **[SUBR à 1 argument]**

5.5 Le GARBAGE-COLLECTOR

La zone de l'interprète qui contient les objets Lisp est allouée dynamiquement. Quand cette zone est saturée une machinerie connue sous le nom de *Garbage-collector* est automatiquement appelée pour récupérer les objets inutilisés.
Si cet essai s'avère infructueux, une erreur fatale se produit dont le libellé est :

** GC : zone liste pleine. ou bien
** GC : no room for list.

si la zone contenant les doublets de listes est pleine ou bien :

** GC : zone symbole pleine. ou bien
** GC : no room for symbols. si c'est le cas de la zone allouée aux symboles.

(GC <i>) [SUBR à 1 argument]

permet d'appeler le Garbage-collecting explicitement. Si l'indicateur <i> est différent de NIL, le message suivant est édité dans le flux de sortie courant à la fin de la récupération :

NB GC : Innn> SYMBOLES=<xxxx> LISTES=<yyyy>

dans lequel :
- <nnn> est le nombre de GC effectués depuis le début de la session
- <xxxx> est le nombre d'octets libres dans le zone atome
- <yyyy> est le nombre de doublets libres dans la zone liste.

Dans tous les cas GC retourne le nombre de doublets libres dans la zone liste.

(GCINFO) [SUBR à 0 argument]

permet de retourner une liste contenant les informations concernant la dernière récupération de mémoire. Cette liste a la forme :

```
(<xxxx> Iyyyy>)
```

<xxxx> est le nombre d'octets libres dans la zone atome et <yyyy> est le nombre de doublets libérés dans la zone liste.

(GCALARM <n>) [SUBR à 1 argument]

est une interruption programmée lançée automatiquement par le système après chaque récupération. L'argument <n> est le nombre de doublets libérés. Cette fonction permet d'éviter l'erreur fatale vue précédemment en provoquant une erreur normale pendant qu'il est encore temps.

cette redéfinition de GCALARM permet en cas de diminution trop importante du nombre de doublets de listes de provoquer une erreur douce :

```
(DE GCALARM (n)
    (WHEN (< n 500)
          (PRINT "J'arrete les frais.")
          (SYSERROR 'GCALARM "nb de doublets restants" n)))
```

Cette autre redéfinition permet en cas de diminution trop importante du nombre de doublets d'agrandir dynamiquement l'espace alloué aux objets Lisp. On suppose que le fichier TMPLL.STARTUP.LL contient juste l'appel suivant :

```
(RESTORE-CORE 'TMPLL.GCALARM.)
```

```
(DE GCALARM (n)
    (WHEN (< n 500)
          (TYOL #"Plus de memoire. L'augmente je?")
          (IFN (MEMQ (TYI) #"OoYyTt")
               (SYSERROR 'GCALARM "plus de place" n)
               (SA VE-CORE 'TMPLL.GCALARM. '(GC T))
               (LELISP (1+ (LELISP-SIZE)) 'TMPLL.STARTUP.LL))))
```

5.6 Erreurs de l'interprète

A l'apparition d'une erreur durant une évaluation, un message décrivant l'erreur est édité dans le flux de sortie courant, ainsi qu'une trace des 4 derniers blocs de contrôle présents dans la pile (voir la fonction CSTACK).
Ce message contient toujours le nom de la fonction ayant provoquée l'erreur, un message explicatif (en français ou en anglais) et l'argument défectueux.
Pour réaliser ce travail, le système appelle toujours la fonction SYSERROR (qui est bien évidemment redéfinissable).

Le_Lisp permet également de provoquer explicitement une erreur, et de tester si une évaluation produit une erreur.

(SYSERROR <sym> <s1> <s2>) **[SUBR à 3 arguments]**

est appelée systématiquement par le système en cas d'erreur ou explicitement dans un programme. <sym> est le nom de la fonction qui a provoqué l'erreur. <s1> est le message décrivant l'erreur. <s2> est l'argument (ou la liste d'arguments) défectueux. D'une manière standard, SYSERROR imprime sur le flux de sortie courant le message :

```
** Isym> : Is1> : Is2>
```

et imprime les 4 derniers bloc de contrôle fabriqués dans la pile, puis retourne au top-level de Lisp après avoir délié toutes les variables. Il est également possible d'entrer dans une boucle d'interaction avec le système avant de retourner au top-level pour inspecter l'état de certaines variables locales ou de retester certaines évaluations. Ces nombreuses possibilités sont décrites dans le troisième volume du manuel.

(ERRSET <e> <i>) **[FSUBR]**

<i> qui est évalué avant <e> est un indicateur. ERRSET va permettre d'évaluer sans risque l'expression <e>. Si au cours de cette évaluation une erreur se déclenche, le système teste l'indicateur <i>. S'il est vrai (<i> différent de NIL), le même message que celui produit par la fonction SYSERROR est imprimé, dans le cas contraire rien n'est imprimé. Dans ces deux cas d'erreur, ERRSET retourne NIL. Si l'évaluation de <e> ne provoque aucune erreur, ERRSET retourne une liste dont le CAR est la valeur de l'expression (ceci pour distinguer une valeur retournée égale à NIL d'une erreur dans l'évaluation).

(ERR <s1> ... <sN>) **[FSUBR]**

évalue les différentes expressions <s1> ... <sN>. La valeur de <sN> devient la valeur retournée du dernier ERRSET (ou du TOPLEVEL si aucun ERRSET n'était défini dynamiquement). Attention cette valeur DOIT être un atome si ERR retourne une erreur, ou une liste si ERR retourne une valeur sans erreur.

(ERRSETP) **[SUBR à 0 argument]**

retourne l'état de l'indicateur du dernier ERRSET défini dynamiquement. ERRSETP est utile si on envisage de redéfinir les fonctions d'erreurs.

(ERROR <s1> <s2>) **[SUBR à 2 arguments]**

est inclus dans le système pour des raisons de compatibilité avec Maclisp. ERROR est équivalent à un appel de la fonction SYSERROR avec le 1er argument égal à l'atome ERROR.

ERROR peut etre de'fini en Lisp de la manie` re suivante :

```
(DE ERROR (s1 s2)
    (SYSERRROR 'ERROR s1 s2))
```

5.7 Accès à l'interprète

Le_Lisp 68K possède trois fonctions de contrôle interne de l'interprète. Ces fonctions sont utilisées pour s'assurer du bon fonctionnement de l'interprète et réaliser les outils de tests décrits dans le troisième volume du manuel en particulier, les traces, les exécutions incrémentales et les contrôles dynamiques.

(TRACEVAL <e>) [SUBR à 1 argument]

évalue l'expression <e> en mode trace. Ce mode spécial de l'interprète permet de suivre tous les appels récursifs internes à la fonction EVAL. En effet à chaque appel interne de EVAL, la fonction suivante STEPEVAL est appelée avec la forme à évaluer comme argument. TRACEVAL retourne la valeur de <e> évalué en mode trace.

(STEPEVAL <e>) [SUBR à 1 argument]

est appelée à chaque appel interne à la fonction EVAL. L'argument est la forme qui devait être évaluée. D'une manière standard STEPEVAL imprime dans le flux de sortie courant :

--> la forme qui devait être évaluée
I-- le résultat de l'évaluation de cette forme.

STEPEVAL peut être de'fini en Lisp de la manie` re suivante :

```
(DE STEPE VAL (e)
    (PRINT "-->" e)
    (PRINT "<--" (E VALTRACE e)))
```

(CSTACK <n>) [SUBR à 1 argument]

A chaque entrée dans une fonction, un échappement, une fermeture ... l'interprète fabrique un bloc de contrôle dans la pile. CSTACK retourne les <n> derniers blocs de contrôle de la pile Lisp. Chacun de ces blocs est transformé en une liste dont le CAR est le type du bloc, et le CDR les arguments du bloc. Voici la liste des blocs actuellement fabriqués par le système. Cette liste pourra s'allonger dans le futur au fur et à mesure du développement du système Le_Lisp.

0	LAMBDA
1	LABEL
2	TAG/EVTAG
3	TRACEVAL
4	CATCH-ALL-BUT
5	UNWIND
6	CLOSURE
7	PROG/DO

5.8 Accès à la mémoire et au CPU

Certaines fonctions permettent d'accéder directement à la mémoire pour pouvoir définir de nouvelles fonctions SUBR, pour tester l'interprète ... Ces fonctions servent à construire le chargeur LLM3 (ce chargeur/assembleur, décrit dans le second volume du manuel, permet de charger du code LLM3 directement écrit à la main ou engendré par le compilateur Le_Lisp).

Ces fonctions utilisent des adresses mémoire quelconques. Les nombres entiers Le_Lisp étant codés sur 16 bits ne permettent pas de représenter une adresse sur 32 bits. Une adresse mémoire <adr> sera représentée en Lisp de 2 manières possibles :
- un nombre sur 16 bits (pour les 64 1ers k de la mémoire)
- une liste de 2 nombres de la forme : (<high> . <low>)

dans laquelle le premier nombre <high> contient les 16 bits de poids forts de l'adresse et le second <low> les 16 bits de poids faibles.

```
l'adresse #$1F3C00   doit s'ecrire    (#$1F . #$3C00)
```

(LOC <s>) **[SUBR à 1 argument]**

permet de connaître l'adresse absolue de l'objet Lisp <s>. L'adresse d'un objet Lisp est le pointeur sur la valeur de cet objet. Cette fonction permet de localiser en mémoire les objets Le_Lisp qui sont crées dynamiquement.

```
ex : (LOC '(A B C))      ->  (15564 . 288)
     (LOC '(A B C))      ->  (13210 . 1200)
```

(VAG <adr>) **[SUBR à 1 argument]**

est la fonction inverse de LOC. VAG retourne l'objet Lisp dont l'adresse absolue est fournie en argument. Donc (VAG (LOC <s>)) = <s>. Attention : VAG ne teste pas si <adr> est l'adresse d'un véritable objet Lisp, un appel erroné de cette fonction peut provoquer une erreur du système hôte.

(MEMORYB <adr> <n>) **[SUBR à 1 ou 2 arguments]**

permet de consulter ou de modifier (si le 2ème argument numérique <n> est fourni) n'importe quel octet de la mémoire dont l'adresse <adr> est fournie en premier argument. Cette fonction ne fait AUCUN contrôle de validité d'adresse et doit donc être utilisée avec beaucoup de précaution. MEMORYB retourne en valeur un nombre, sur 8 bits, représentant la valeur de l'octet après modification eventuelle.

(MEMORY <adr> <n>) [SUBR à 1 ou 2 arguments]

permet de consulter ou de modifier (si le 2ème argument numérique <n> est fourni) n'importe quel mot (de 16 bits) de la mémoire dont l'adresse <adr> est fournie en premier argument. Cette fonction ne fait AUCUN contrôle de validité d'adresse et doit donc être utilisée avec beaucoup de précaution. En particulier les adresses du 68000 étant des adresses d'octets, il est obligatoire de donner une adresse paire à cette fonction. MEMORY retourne en valeur un nombre, sur 16 bits, représentant la valeur du mot après modification eventuelle.

(CALL <adr> <a1> <a2> <a3> <a4>) [SUBR à N arguments]

le premier argument <adr> doit être l'adresse d'un sous-programme en mémoire. CALL va lançer ce sous-programme après avoir charger les quatre accumulateurs A1, A2, A3 et A4 avec les valeurs respectives <a1>, <a2>, <a3> et <a4>. Cette fonction doit être utilisée avec précaution car elle ne teste pas la validité du code qui est lançé mais permet de réaliser l'interface machine-68000 <-> Le_Lisp utilisée dans le compilateur ou pour des traitements particuliers.
CALL retourne en valeur la valeur actuelle de l'accumulateur A1 après exécution du code débutant en <adr>.

Attention : le code invoqué au moyen de CALL doit impérativement se terminer par l'instruction LLM3, RETURN (voir le 2ème volume du manuel), et charger le registre A1 avec une valeur Lisp qui est retournée par la fonction CALL.

(CALLN <adr> <l>) [SUBR à 2 arguments]

est identique à la fonction précédente mais empile la liste des valeurs contenues dans <l> avant d'appeler le sous-programme rangé à l'adresse <adr>. Ce format est celui des NSUBR et de certaines EXPR compilées. Les mêmes précautions, que pour la fonction précédente doivent être prises.