MAI 86
2e Edition

Le_Lisp
Version 15.2

LE MANUEL DE RÉFÉRENCE

Jérôme CHAILLOUX

~~LA BIBLIOTHÈQUE INITIALE~~

Matthieu DEVIN
Francis DUPONT
Jean-Marie HULLOT
~~Bertrand SERLET~~ Bernard SERPETTE
Jean VUILLEMIN

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

L E _ L I S P

d e l ' I N R I A

V e r s i o n 1 5 . 2

L e M a n u e l

d e r é f é r e n c e

Mai 1986

Jérôme CHAILLOUX

Matthieu DEVIN
Francis DUPONT
Jean-Marie HULLOT
Bernard SERPETTE
Jean VUILLEMIN

AVANT-PROPOS

Le_Lisp [Chailloux&al 84] est un système Lisp, développé à l'INRIA, qui 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 Lisp* tels que le Lisp de la machine Lisp du M.I.T. [Weinreb et Moon 81], NIL [White 79, Burke & Carette 82] et Franz Lisp [Foderaro & Sklower 81]) à qui il doit ses performances en tant que langage d'écriture de systèmes sûrs et performants.

Le problème de la compatibilité avec les autres systèmes Lisp fut une préoccupation constante tout au long de la conception du nouveau dialecte Le_Lisp. L'aboutissement récent du premier effort du COMMON LISP Américain, [COMMON 84, Steele 84] a permis la clarification des différences. Toutefois devant le problème majeur posé par la taille du système résultant, le peu d'efficacité de Common Lisp sur des matériels non spécialisés [Brooks & Gabriel 84] et les problèmes de normalisation industrielle [Boston 85], Le_Lisp n'assurera la compatibilité totale qu'avec la norme en cours de standardisation au niveau ISO [Padget&al 86, Stoyan&al 86].

Le_Lisp est un système très facilement portable [Devin 85a] écrit dans le langage de la machine virtuelle LLM3 [Chailloux 85b], indépendant des systèmes d'exploitation, qui fonctionne aujourd'hui sur les unités centrales suivantes :

- Motorola MC68000
- VAX 11
- Perkin Elmer 32
- HB68 Multics
- Intel 8088/8086
- Ridge32/SPS9
- IBM série 30xx
- PR1ME

De nombreux autres transports sont en cours ou projetés en particulier sur :

- Bell Mac 32
- SEL 32
- NS16000
- HP9000/500
- DPS7
- DPS8/GCOS
- Mini6
- Norsk Data
- Cray I

Des essais de microprogrammation de cette machine ont été entrepris [Devin 85b] et un prototype d'Unité Centrale LLM3 est en cours de développement [Audoire 85].

Une version réduite est également disponible sur les machines à base d'unité centrale de type Intel 8080 ou Zilog 80 sous système d'exploitation CP/M ou EDEN [Chailloux 83, Saint-James 84]. Cette version se nomme *Le_Lisp_80.*

Le_Lisp est destiné au développement et à la réalisation de systèmes de haut niveau axés sur la conception assistée par ordinateur, les systèmes experts, l'enseignement assisté par ordinateur, la simulation, la synthèse d'images colorées, l'écriture de

nouveaux langages applicatifs, l'informatique musicale et d'une manière générale les problèmes d'intelligence artificielle et les domaines qui y sont rattachés.

Ce manuel est un *manuel de référence* et ne doit pas être considéré comme un manuel d'initiation à Lisp ni comme un manuel de programmation en Lisp. Nous ne saurions trop conseiller aux tout débutants les livres, en français, de C. Queinnec [Queinnec 82, Queinnec 84], Kiremitdjian et J.P. Roy [K&R85], M. Cayrol [Cayrol 83], H. Farreny [Farreny 84], H. Wertz [Wertz 85], J.J. Girardot [Girardot 85], ou bien, en langue étrangère, de H. Winston et B. Horn (anglais) [Winston 84] et de H. Stoyan et G. Görz (allemand) [Stoyan&Görz 84].

Le_Lisp est un système en constante évolution du fait de ses utilisateurs divers et nombreux. Pour participer à sa mise au point, à son extension, à sa clarification aussi bien dans sa documentation que dans son implantation, abonnez-vous au système de forum distribué disponible sur *usenet* en France :

```
% readnews -n fnet.lelisp
```

Vous pouvez en outre toujours contacter l'équipe de développement par courrier électronique à l'INRIA :

```
...!mcvax!inria!lelisp                    sur usenet
```

Le_Lisp est le fruit des compétences d'un grand nombre de personnes de l'INRIA et d'ailleurs depuis plusieurs années. En particulier :

Z. Bellahsene et C. O'Donnell ont écrit le traducteur de l'assembleur Motorola vers l'assembleur MIT et l'expanseur de macros LLM3 pour le système fonctionnant sur SM90.

G. Baudet, de l'Université de Brown, a transporté le système sur la machine APOLLO.

M. Devin, F. Montagnac et P. Cipière, de l'Ecole des Mines de Sophia Antipolis, ont transporté le système sur la machine Perkin Elmer 32 sous système UNIX.

M. Devin, R. Fournier, J. Grimm et F. Montagnac, de l'Ecole des Mines et de l'INRIA à Sophia Antipolis, ont transporté le système sur Honeywell-Bull 68 sous système Multics.

M. Dana, de l'E.N.S.T., a transporté le système sur VAX sous système VMS.

J.M. Hullot a conçu et implanté les RECORD, Lucifer et son fils Ceyx.

E. Saint-James, de l'Université de Paris 6, a largement contribué à l'élaboration du système Le_Lisp_80.

B. Serlet a écrit le gestionnaire du tas ainsi que l'utilitaire *timetrace*.

O. Guillaumin a écrit l'éditeur de ligne *edlin*.

C. Jullien et B. Serpette, d'ACT Informatique, ont transporté le système sur Intel 8086/8088 sous système MSDOS/PCDOS.

E. Hardebeck et B. Serpette, d'ACT Informatique, ont transporté le système sur Lisa/MacIntosh.

L. Fallot, de l'Université de Bordeaux I, a contribué à l'écriture du chargeur dynamique sous système UNIX.

C. Queinnec a contribué à la définition du terminal virtuel.

F. Dupont a participé au portage du système sur VME, a réalisé un système de gestion

de fichiers décentralisée et a conçu le micro langage LLM3.

L. Gallot et C. Chen ont participés à la définition du système multi-fenêtres et du graphique virtuel.

F. André, Y. Autret, G. Berry, J.P. Briot, C. Caron, P. Casteran, J.L. Chatelain, O. Chénetier, P. Cointe, G. Coslado, G. Cousineau, J. DalleRive, Y. Devillers, J. Duthen, Ph. Donz, R. Ehrlich, E. Godefroy, G. Huet, C. Jullien, B. Lang, A. Laville, B. Melèse, G. Nowak, E. Neidl, J. L. Richier, A. Suarez ainsi que tous les participants des Mardis Du Lisp [LMDL 86] ont participé à l'élaboration du nouveau dialecte Le_Lisp et à sa documentation;

J. Vuillemin et G. Kahn ont apporté leur appui efficace et amical tout au long du projet qui, du moins à ses débuts, était hautement spéculatif;

De nombreux autres utilisateurs, trop nombreux pour être cités, ont également influencé, par leurs excellentes suggestions, la réalisation et l'implantation du système Le_Lisp.

La Version 15.2

La version 15.2 constitue une re-écriture presque complète de la version 15 mais en est une version *compatible et étendue*;

- Matthieu Devin s'est plus particulièrement chargé des nouvelles entrées/sorties, des nouveaux outils de mise au point, du terminal virtuel et du graphique virtuel,
- Bernard Serpette, du nouveau compilateur *complice*
- Jean Vuillemin, des nouvelles arithmétiques rationnelles et complexes
- Francis Dupont, des nouvelles impressions spécialisées et des interfaces UNIX
- Jean-Marie Hullot, des nouvelles extensions objets (Alcyone) et du graphique virtuel.
- Pierre-Louis Neumann, du système multi-tâches.
- Alain Beges, des bibliothèques de compatibilité MacLisp, CommonLisp <-> Le_Lisp et des portages de MacSyma et de Reduce.

Jérôme Chailloux
Rocquencourt,
le 20 Mai 1986

Ce manuel a été composé sur imprimante laser AGFA 400 avec le logiciel ditroff *sur système UNIX.*

Les modifications majeures par rapport à la version 15 [Chailloux 85] sont signalées par une barre verticale en marge droite.

BIBLIOGRAPHIE

[Abelson & Sussman] Harold Abelson & Gerald Jay Sussman, *Structure and Interpretation of Computer Programs,* The MIT Press, McGraw Hill Book Company, 1985.

[Audoire 85] Louis Audoire, *Un Processeur Spécialisé mLLM3 sur SPS7,* Actes des journées SM90, Versailles, Décembre 1985.

[Boston 85] , *Premier Symposium de Normalisation de Common Lisp,* Boston, Décembre 1985.

[Brook & Gabriel 84] Rodney A. Brook & Richard P. Gabriel, *A Critique of Common Lisp,* 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas.

[Burke & Carette 82] , *NIL Notes for Release 0,* Massachusetts Institute of Technology, December 1982.

[Cayrol 83] Michel Cayrol, *Le langage LISP,* Cepadues Editions, Toulouse, 1983.

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

[Chailloux 83] Jérôme Chailloux, *Le_Lisp 80 version 12, le manuel de référence,* rapport technique INRIA no 27, Juillet 1983.

[Chailloux&al 84a] Jérôme Chailloux, Matthieu Devin et Jean-Marie Hullot, *Le_Lisp : a Portable an Efficient Lisp System,* 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas.

[Chailloux 85a] Jérôme Chailloux, *Le_Lisp version 15, le manuel de référence,* Documentation INRIA, Février 1985.

[Chailloux 85b] Chailloux Jérôme, *La machine virtuelle LLM3,* rapport technique no 55, INRIA, Juin 1985.

[Cointe 82] Pierre Cointe, *Fermetures dans les lambda-interprètes. Application aux langages LISP, PLASMA et SMALLTALK,* thèse de 3ème cycle, Université de Paris VI, 1982.

[CLSUBSET 84] , *mail file,* CLSUBSET.*[COM,LSP]@SAIL.

[COMMON 84] , *mail file,* COMMON.*[COM,LSP]@SAIL.

[Dana 86] Michel Dana, *Le_Lisp v15.2 sous système VAX/VMS,* Rapport ENST, Janvier 1986.

[Devin 85a] Matthieu Devin, *Le portage du système Le_Lisp : mode d'emploi,* Rapport Technique no 50, INRIA, Mai 1985.

[Devin 85b] Devin Mattieu, *La Microprogrammation du système Le_Lisp : une première approche,* Rapport de Recherche no 441, INRIA, Septembre 85.

[Eu_Lisp 86] , *Desiderata for the standardization of LISP* (à paraître) 1986 ACM Symposium on Lisp and Functional Programming, Août 1986, Boston.

[Farreny 84] Henry Farreny, *LISP,* Masson, Paris, 1984.

[Foderaro & Sklower 81] , *Franz Lisp Manual,* Univ. of California, Berkeley, Ca., September 1981.

[Girardot 85] Jean-Jacques Girardot, *les langages et les systèmes LISP,* édi tests, Paris, 1985.

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

[Hullot 83] Jean-Marie Hullot, *Ceyx, a Multiformalism Programming Environement,* IFIP83, R.E.A. Masson (ed), North Holland, Paris 1983.

[Hullot 85a] Jean-Marie Hullot, *Programmer en Ceyx,* rapports techniques no 44-45-46, INRIA, Février 1985.

[Hullot 85b] Jean Marie Hullot, *Alcyone, La boite à outils Objets,* Rapport Technique no 60, INRIA, Novembre 1985.

[K&R 85] Georges Kiremitdjian et Jean-Pierre Roy , *Lire Lisp, le langage de l'Intelligence Artificielle,* Cedic-Nathan, Paris, 1985.

[LMDL 86] Jérôme Chailloux (éd.), *Les Comptes Rendus des Mardis du Lisp,* rapports internes du projet VLSI, INRIA, 1986.

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

[Padget&al 86] Julian Padget, Jérôme Chailloux, Thomas Christaller, Matthieu Devin, John Fitch, Tim Krumnack, Ramon Lopez, Eugen Neid, Stephen Pope, Christian Queinnec, Luc Steels, Herbert Stoyan, *Desiderata for a standardization of LISP,* (à paraître) 1986 ACM Symposium on Lisp and Functional Programming, Août 1986, Boston.

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

[Queinnec 84] Christian Queinnec, *Lisp : mode d'emploi,* Eyrolles, Paris, 1984.

[Saint-James 84] Emmanuel Saint-James, *Recursion is more efficient than iteration,* 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas.

[Steele 84] Guy L. Steele Jr., *Common Lisp, the language,* Digital Press, 1984.

[Stoyan&Görz 84], *Herbert Stoyan and* Günter Görz Lisp, Eine Einführung in die Programmierung, Springer-Verlag, Berlin, 1984.

[Stoyan&al 86] Herbert Stoyan, Julian Padget, Jérôme Chailloux, Thomas Christaller, Matthieu Devin, John Fitch, Tim Krumnack, Ramon Lopez, Eugen Neidl, Stephen Pope, Christian Queinnec, Luc Steels, *Towards a LISP standard,* (à paraître) 1986 ECAI.

[Teitelman 83] , *Interlisp Reference Manual,* XEROX PARC, October 1983.

[Vuillemin 86] Jean Vuillemin, *Arbitrary precision Arithmetics in Le_Lisp,* Rapport INRIA, (à paraître).

[Weinreb et Moon 81] , *Lisp Machine Manual,* Fourth Edition, Artificial Intelligence Laboratory, M.I.T., Cambridge, Mass., July 1981.

[Wertz 85] Harald Wertz, *Lisp, une introduction à la programmation,* Masson, Paris, 1985.

[Winston 84] H. Winston et B. Horn, *Lisp,* 2nd edition, Addison Wesley, 1984.

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

PLAN DE LECTURE

Ce manuel est divisé en cinq parties :

I - la mise en oeuvre des systèmes
II - la définition du langage et des fonctions de base
III - la bibliothèque initiale Le_Lisp
IV - la bibliothèque interactive
V - le source Le_Lisp des fichiers initiaux

Première partie

La première partie se compose d'un seul chapitre. Ce chapitre 1 contient la mise en oeuvre des différents systèmes. Sa lecture est indispensable pour le premier contact avec l'un des systèmes ou pour l'adapter à une utilisation particulière.

Deuxième partie

La deuxième partie comprend 6 chapitres :

Le chapitre 2 traite 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 ou d'interprétation.

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

Le chapitre 4 contient la description de toutes les fonctions standard travaillant sur les nombres.

Le chapitre 5 traite des structures, des types étendus et de la programmation objet.

Le chapitre 6 traite des fonctions spécialisées dans les entrées-sorties simples, la manipulation des fichiers et l'utilisation des terminaux.

Le chapitre 7 traite des fonctions systèmes. Sa lecture est réservée à ceux qui désirent réaliser des sous-systèmes spécialisés écrits en Lisp.

Troisième partie

La troisième partie contient 7 chapitres :

Le chapitre 8 décrit le paragrapheur d'expressions.

Le chapitre 9, décrit les fonctions sur les impressions spécialisées, format et manipulation d'objets circulaires ou partagés.

Le chapitre 10, décrit une arithmétique rationnelle et complexe en précision arbitraire.

Le chapitre 11 traite de tout ce qui est mise au point des programmes.

Le chapitre 12 décrit l'assembleur de la machine virtuelle LLM3, et son chargeur mémoire.

Le chapitre 13 décrit les compilateurs.

Le chapitre 14 décrit toutes les interfaces entre Le_Lisp et les bibliothèques externes.

Quatrième partie

La quatrième partie décrit tout ce qui est interaction.

Le chapitre 15 traite du terminal virtuel.

Le chapitre 16 traite des différents éditeurs vidéo intégrés au système.

Le chapitre 17, décrit un éditeur de ligne vidéo.

Le chapitre 18, décrit l'interface virtuelle multi-fenêtre sur écran haute résolution.

Le chapitre 19, décrit la gestion du dispositif de pointage virtuel.

Le chapitre 20, décrit le système graphique virtuel.

Cinquième partie

Cette cinquième partie contient le source des fichiers initiaux de la bibliothèque standard Le_Lisp :

L'annexe A, le source du fichier STARTUP
l'annexe B, le source du fichier FILES
l'annexe C, le source du fichier TOPLEVEL
l'annexe D, le source du fichier DEFS
l'annexe E, le source du fichier GENARITH
l'annexe F, le source du fichier ARRAY
l'annexe G, le source du fichier DEFSTRUCT
l'annexe H, le source du fichier SORT
l'annexe I, le source du fichier CALLEXT
l'annexe J, le source du fichier PRETTY
l'annexe K, le source du fichier TRACE
l'annexe L, le source du fichier DEBUG
l'annexe M, le source du fichier PEPE

Enfin, le plus important dans un manuel de ce type, trois tables d'index permettent un accès direct aux concepts, aux instructions LLM3 et aux fonctions et variables Le_Lisp.

CHAPITRE 1

Utilisation et Installation

1.1 Les Différents Systèmes

Le_Lisp a été porté sur les Unités Centrales suivantes :

- Motorola MC68000/68010/68020
- DEC VAX 11
- Perkin Elmer 32
- Honeywell-Bull 68
- Intel 8088/8086
- Ridge 32/SPS9
- PR1ME

Il fonctionne aujourd'hui sur les machines suivantes (le nom du système, fourni par la fonction SYSTEM, est donné entre crochets) :

- SM90 à base de MC68000, sous système SMX [sm90].
- VME à base de MC68000, support du dispositif d'affichage couleur, COLORIX, réalisé par Louis Audoire [vme].
- MicroMega à base de MC68000, sous système MIMOS [micromega].
- APOLLO à base de MC68000, sous système UNIX [apollo].
- SUN à base de MC68000/10/20, sous système UNIX [sun]
- METHEUS à base de MC68000, sous système UNIX [metheus].
- CADMUS à base de MC68000, sous système UNIX [cadmus].
- HP9000 série 300 à base de MC68020, sous système UPUX [hp9300]
- Apple MacIntosh [mac]
- VAX 11 sous système UNIX, Ultrix et VMS [vaxunix] [vaxvms].
- Perkin Elmer 32/20-40 sous système UNIX [peunix].
- Honeywell-Bull 68 sous système Multics [multics].
- IBM 4341 sous système VM/UTS [vm_uts].
- IBM PC sous système PCDOS [pcdos].
- Ridge 32/SPS9 sous système ROS [sps9]
- BellMac 32 sous UNIX système V [bell]
- IBM PC et compatibles [pcdos]
- Méta Viseur [metaviseur]

1.2 Pour Débuter avec Le_Lisp

Pour jouer Le_Lisp, il suffit, sous tous les systèmes hôtes, d'émettre sur le terminal la commande :

```
lelisp
```

dans tous les cas, le système Le_Lisp répond :

```
******* Le_Lisp (by INRIA) version 15.2 (jj-mm-aa) [system]
= Systeme standard compile.
```

dans lequel (jj-mm-aa) est la date de la dernière modification du système et [system] le type du système utilisé. Le_Lisp entre 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_Lisp 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 sur SM90 :

```
$                                   nous sommes sous SMX
$ lelisp
******* Le_Lisp (by INRIA) version 15.2 (20/Mai/86) [sm90]
= Systeme standard compile avec compilateur.
? ()
= ()                        ; la liste vide!
? (length (oblist))
= 942
? (version)                 ; indique le numéro de version
= 15.2
? (system)                  ; indique le type du système
= sm90
? (+ 1 2 3 4)
= 10
? (* (+ 1 2) (- 6 3) 2)
= 18
? (de fib (n)
?     (cond ((= n 1) 1)
?           ((= n 2) 1)
?           (t (+ (fib (1- n)) (fib (- n 2))))))
= fib
? (fib 20)
= 6765                  ; en moins de 9 secondes!
                        ; (avec les mémoires rapides)
? ^L hanoi              ; chargement d'une bibliothèque
= /usr/local/lelispv15.2/llib/hanoi.ll
? (hanoi 3)             ; que c'est joli !
= hanoi
? (end)
  Que Le_Lisp soit avec vous.
```

1.3 Le Lancement du Système sous UNIX

La commande générale de lancement du système Le_Lisp sous système d'exploitation UNIX à la forme suivante :

```
lelisp [n] [[-r] fichier] [-s]
```

dans laquelle les éléments entre crochets sont optionnels.

l'argument **n** est la taille de la zone des listes exprimée en *8k cellules de liste*. Par défaut **n**=3 (c'est-à-dire 24k cellules de liste). La limite théorique est de 128 (c'est-à-dire 1024 k cellules de liste), la limite pratique est la taille physique de la mémoire ou la taille maximale d'un processus logiciel pour le système d'exploitation hôte. A titre indicatif, une unité de *8k cellules de liste* correspond à toute la mémoire adressable d'un Z80 ou d'un PDP11.

l'argument **fichier** permet de spécifier le nom d'un fichier quelconque contenant des programmes Lisp à charger avant d'entrer dans la boucle interactive du système.

l'argument **-r fichier** permet de spécifier une image mémoire qui va être chargée avant d'entrer dans la boucle interactive du système.

l'argument **-s** permet de lancer Le_Lisp sans imprimer la bannière standard. Cet argument est souvent utilisé avec l'argument précédent pour lancer des sous-systèmes écrits en Le_Lisp ou pour utiliser Le_Lisp comme *filtre* dans des systèmes d'exploitation de type UNIX.

Après chargement de l'image standard le fichier *$HOME/.lelisp* est automatiquement chargé.

1.4 L'installation du Système sous UNIX

Cette section concerne les systèmes Le_Lisp tournant sous UNIX : VAX 11 (BSD), SM90 (SMX5.1), SPS7 (SPIX), SPS9 (ROS 3.3), SUN (BSD4.2), PerkinElmer (V7), Fortune32 (V7) ...

1.4.1 Installation du système

Le système Le_Lisp, version 15.2, est distribué sur un support magnétique (bande tar 1600 bpi, cartouche DMA, Streamer) qu'il faut recopier sur disque dans le *catalogue (directory) d'installation* du système. On choisit usuellement le catalogue */usr/local/lelisp*, ou le catalogue */usr/local/lelispv15.2* si l'on désire faire cohabiter plusieurs versions de Le_Lisp. Une fois installé le système occupera environ 3 mégaoctets dans ce catalogue.

Le catalogue d'installation contient plusieurs sous-catalogues dont un, dit *catalogue système*, porte le nom de la machine d'installation (vax, sm90, sun, ...).

```
$ lf /usr/local/lelispv15.2
LISEZ_MOI     gabriel/      llib/         lltest/       vax/
ceyx/         info/         llmod/        llub/         virbitmap/
```

```
common/     llcore/     llobj/     manuel/     virtty/
```

Dans certaines distributions le catalogue *llcore* n'existe pas. En ca cas, il faut le créer et lui donner les droits d'accès en lecture à tous les utilisateurs. Il faut de plus donner les droits d'accès en écriture à tous les utilisateurs sur le catalogue *virtty*.

L'installation du système nécessite la mise à jour de chemins d'accès (paths) absolus et la construction des images mémoire Le_Lisp.

1.4.1.1 Mise à jour des chemins d'accès absolus

Il faut se positionner dans le catalogue système et lancer la commande *newdir*, sans arguments. Il faut faire cette opération une seule fois, après recopie du support magnétique sur disque. Il faut la renouveler à chaque déplacement du système Le_Lisp vers un nouveau catalogue.

```
$ cd /usr/local/lelispv15.2/vax
$ newdir
2000
DIR=/usr/local/lelispv15.2
2000
2000
DIR=/usr/local/lelispv15.2
2000
20828
      (defvar #:system:directory "/usr/local/lelispv15.2/"))
20826
3915
(defvar #:ceyx:directory "/usr/local/lelispv15.2/ceyx/")
3913
```

1.4.1.2 Construction des images mémoire Lisp

Il faut se positionner dans le catalogue système et lancer la commance *make* avec comme paramètre le nom de l'image mémoire à construire.

Le makefile du catalogue système fournit des points d'entrée permettant de construire plusieurs images mémoires (lelisp, complice, ceyx) avec différentes configurations mémoire (normale, +, ++). Ce makefile peut être étendu à la construction de nouveaux systèmes.

Le makefile utilise la commande *config* qui construit un script shell permettant de lancer Le_Lisp en restaurant l'image mémoire créée. Il faut recopier ce script dans un catalogue de commandes de votre système, */usr/local/bin* par exemple.

Les images mémoire sont stockées dans le catalogue *llcore*. Comme elles occupent beaucoup de place mémoire disque, il peut être intéressant de monter le catalogue llcore dans une partition spéciale.

```
$ cd /usr/local/lelispv15.2/vax
$ make lelisp
config lelisp lelispbin lelispconf.ll -stack 6 -code 196 -heap 128 \
-number 0 float 2  -vector 4 -string 5 -symbol 3 -cons 4
*******  Le_Lisp  (by INRIA)  version 15.2  (20/Mai/86)  [vaxunix]
= ()
= #:system:line-mode-flag
```

```
= unix
= #:system:directory
 (load-std sav min edit env ld llcp)   pour charger l'environnement std,
 (load-cpl sav min edit env ld llcp)   pour l'environnement modulaire.
= startup
= t
? (setq #:system:name (quote lelisp))
= lelisp
? (load-std () t t t t t)
Je charge /usr/local/lelispv15.2/llib/virtty.ll
Je charge /usr/local/lelispv15.2/llib/virbitmap.ll
Je charge /usr/local/lelispv15.2/llib/pepe.ll
Je charge /usr/local/lelispv15.2/llib/defstruct.ll
Je charge /usr/local/lelispv15.2/llib/sort.ll
Je charge /usr/local/lelispv15.2/llib/array.ll
Je charge /usr/local/lelispv15.2/llib/callext.ll
Je charge /usr/local/lelispv15.2/llib/trace.ll
Je charge /usr/local/lelispv15.2/llib/pretty.ll
Je charge /usr/local/lelispv15.2/llib/genarith.ll
Je charge /usr/local/lelispv15.2/llib/debug.ll
Je charge /usr/local/lelispv15.2/llib/lapvax.ll
Je charge /usr/local/lelispv15.2/llib/llcp.ll
 (llcp-std '<nom>)  pour compiler l'environnement standard
= ()
? (llcp-std #:system:name)
Attendez, je sauve :
Systeme standard compile, editeur, environnement, chargeur et compilateur
*******  Le_Lisp  (by INRIA)  version 15.2  (20/Mai/86)  [vaxunix]
= Systeme standard compile, editeur, environnement, chargeur et compilateur
? (end)
Que Le_Lisp soit avec vous.
$ cp lelisp /usr/local/bin
$
```

L'installation est terminée!

```
$
$ lelisp
*******  Le_Lisp  (by INRIA)  version 15.2  (20/Mai/86)  [vaxunix]
= Systeme standard compile, editeur, environnement, chargeur et compilateur
?
```

En général l'installation décrite ci-dessus est suffisante. Pour certaines applications particulières il est parfois nécessaire de modifier la configuration du système, c'est à dire l'ensemble des fichiers Lisp chargés dans l'image mémoire, ou de régler finement l'organisation de l'espace mémoire Lisp.

1.4.2 Modification de la Configuration du Système

La distribution standard permet de construire le système selon trois configurations différentes, non exclusives, correspondant à trois points d'entrée du makefile :
- lelisp : environnement complet avec outils de mise au point, éditeur et compilateur standard.
- complice : environnement complet avec outils de mise au point, éditeur et compilateur modulaire IComplice.
- ceyx : configuration lelisp, enrichie du langage orienté objet Ceyx.

La configuration de chaque système est décrite dans un fichier du catalogue système (lelispconf.ll, complcconf.ll, ceyxconf.ll). Il est possible de modifier la configuration de chaque système en éditant ces fichiers. Il est aussi possible de créer un nouveau point d'entrée dans le makefile décrivant la configuration d'un nouveau système.

Exemple :
construction d'une image mémoire de nom *monlisp* contenant l'environnement standard sans éditeur, l'éditeur de ligne EDLIN, et les fonctions du séquenceur, SCHEDULE. Le système crée sera lancé par la commande *monlisp*. Il faut créer un fichier de configuration, par exemple monlispconf.ll :

```
$ cat monlispconf.ll
(load-std ()        ;    chargement de l'environnement sans sauvegarde
          t         ;    l'environnement minimum,
          ()        ;    pas d'éditeur,
          t         ;    l'environnement complet,
          t         ;    le chargeur
          t)        ;    le compilateur
(libload edlin)     ;    chargement de l'éditeur de ligne EDLIN
(libload schedule) ;     et du scheduler.
(progn
   (llcp-std #:system:name)     ; compilation et construction de l'image
   (edlin)                      ; lancement d'EDLIN après restauration
   "Bienvenue dans Mon Lisp")) ; message de bienvenue.
```

Il faut ensuite rajouter le point d'entrée *monlisp* dans le makefile Le système *monlisp* est construit avec l'interprète standard *lelispbin* (pas de modules C supplémentaires), et les tailles de zones standard SIZE.

```
monlisp:        monlispconf.ll
        config monlisp lelispbin monlispconf.ll $(SIZE)
```

Il suffit ensuite de construire le système par la commande *make monlisp*.

```
$ make monlisp
config monlisp lelispbin monlispconf.ll -stack 6 -code 196 -heap 128 \
-number 0 float 2  -vector 4 -string 5 -symbol 3 -cons 4
*******  Le_Lisp  (by INRIA)  version 15.2  (20/Mai/86)  [vaxunix]
Que Le_Lisp soit avec vous.
$
$ cp monlisp /usr/local/bin
$ monlisp
*******  Le_Lisp  (by INRIA)  version 15.2  (20/Mai/86)  [vaxunix]
= Bienvenue dans Mon Lisp
?
```

1.4.3 Modification des tailles des zones de données

Les tailles des différentes zones de données du système sont figées lors de la construction de chaque image mémoire. Ces tailles ne peuvent pas être modifiées dynamiquement. Il est donc possible de saturer l'une des zones, ce qui provoque une des erreurs fatales décrites à la section 7.10 :

```
*** Erreur Fatale : zone XXXXX pleine
```

Il est nécessaire de construire une nouvelle image mémoire pour agrandir une zone de données.

Il faut cependant noter que :
- la taille de la zone des listes peut être modifiée à chaque appel du système : il n'est donc pas nécessaire de construire une nouvelle image mémoire pour corriger l'erreur *zone des listes pleine*.
- la saturation d'une zone peut être due à une erreur de votre programme.
- la fonction Lisp GCINFO permet d'avoir des informations sur le taux de remplissage des différentes zones.

Le makefile fournit des points d'entrée qui permettent de construire chaque système (lelisp, complice, ceyx) avec différentes tailles. Par exemple le système lelisp++ est un système lelisp avec des zones mémoire très vastes.

Les variables SIZEx du makefile décrivent les tailles des différentes zones.

La taille du système *lelisp* est par exemple décrite par la variable SIZE :

```
SIZE= -stack 6 -code 256 -heap 100 -vector 3 -number 0 -float 1 \
      -string 4 -symbol 3 -cons 4
```

Chaque option "-<zone>" précise la taille de la zone <zone>. Les unités décrivant ces tailles sont décrites dans le tableau suivant :

Les zones de mémoire Le_Lisp

nom	*objet lisp*	*unité*	*taille réelle*
stack	pile	k-mot	1 = 4 k-octet
code	code compilé	k-octet	1 = 1 k-octet
heap	tas	k-octet	1 = 1 k-octet
vector	Vecteurs	k-vecteur	1 = 1 k-vecteur = 8 k-octet
number	Nombres boxés		toujours à 0 sur ces systèmes
float	Nombres Flottants	k-Flottant	1 = 1 k-flottant = 8 ou 0 k-octet(*)
string	Chaînes	k-Chaîne	1 = 1 k-chaîne = 8 k-octets
symbol	Symboles	k-Symbole	1 = 1 k-symbole = 64 k-octet
cons	Doublets	8k-Doublet	1 = 8 k-doublet = 64 k-octet

(*) certaines versions de l'interprète n'utilisent aucune place mémoire pour les nombres flottants.

Les Chaînes et les Vecteurs occupent une certaine place dans le HEAP, donnée dans le tableau suivant.

Utilisation du HEAP

Vecteur de n objets	8+4n octets
Chaîne de n caractères	9+n octets

Pour construire une image mémoire avec de nouvelle tailles il faut changer les définitions des variables SIZEx, ou bien ajouter de nouveaux points d'entrée avec d'autres paramètres SIZE, puis relancer la commande make.

Exemple :
ajout d'un point d'entrée permettant de construire un système *lelisp* avec 10.000 (10K) vecteurs.

```
SIZEV= -stack 6 -code 256 -heap 100 -vector 10 -number 0 -float 1 \
       -string 4 -symbol 3 -cons 4

lelispv :       lelispconf.ll
        config lelispv lelispbin lelispconf.ll $(SIZEV)
```

Il suffit ensuite de relancer la commande make pour construire l'image mémoire du système lelispv.

```
$ make lelispv
config lelispv ...
$ cp lelispv /usr/local/bin
$ lelispv
Le_Lisp by INRIA ...
```

1.4.4 Lien de l'interprète avec des modules C

Le chapitre 14 de ce manuel décrit comment lier l'interprète *lelispbin* et des programmes C pour construire un nouveau binaire.

Pour réaliser cette opération il faudra utiliser le fichier *lelispbin.o* du catalogue système. Ce fichier est le résultat d'un lien (par *ld -r*) de tous les modules constituant l'interprète Le_Lisp, hormis le module *lelisp.o* et, sur certaines machines, le module *mouse.o*. Ces deux derniers modules sont obtenus par la compilation des fichiers C *lelisp.c* et *mouse.c* du catalogue *common*.

1.4.5 Appel du shell

Il est possible d'appeler le *shell* au moyen de la fonction COMLINE (et du macro caractère !). Il s'agit de /bin/sh avec une interprétation locale de la commande *cd*.

```
% lelisp
*******  Le_Lisp  (by INRIA)  version 15.2  (20/Mai/86)  [vaxunix]
= Systeme standard compile, editeur, environnement, chargeur et compilateur
?
? !pwd
/usr/local/lelispv15.2/vax
= t
? !cd ../
```

```
= t
? !pwd
/usr/local/lelispv15.2
```

1.5 Le Lancement du Système sous VMS

La documentation complète de l'installation et du lancement sous VMS est décrite dans [Dana 86]. La commande générale de lancement du système Le_Lisp sous système d'exploitation VMS a la forme suivante :

```
$ LELISP
```

1.6 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 BACKSPACE ou flèche à gauche; 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) ou ^U. La ligne est effacée.

- 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 tapant deux points l'un à la suite de l'autre. Si rien de visible ne se produit, vous êtes vraisemblablement à l'intérieur d'un commentaire ou d'une chaîne de caractères.

- pour revenir à la boucle principale de l'interprète (ou dans une boucle d'inspection) dans les cas graves appuyez sur la touche BREAK, DEL ou ^C sur les systèmes fonctionnant avec UNIX ou ^C avec VMS.

- pour revenir au système hôte dans les cas vraiment désespérés (en perdant par la même occasion votre environnement Lisp) entrez ^\ sur les systèmes fonctionnant avec UNIX ou ^Y avec VMS.

- pour envoyer une ligne de commande au système hôte (si celui-ci le permet) tapez cette ligne précédée du caractère ! (point d'exclamation).

```
? !dir
```

- pour charger un fichier précédemment créé, dites simplement :

```
? ^L fichier
```

c'est-à-dire le caractère ^L (appui sur la touche CTRL et L simultanément) suivi du nom du fichier sans son extension qui est .ll par défaut.

- pour créer ou éditer un fichier, appelez l'un des éditeurs vidéo en tapant simplement :

```
? ^E fichier
```

c'est-à-dire le caractère ^E (appui sur la touche CTRL et E simultanément) suivi du nom du fichier sans son extension qui est .ll par défaut. Pour visualiser un récapitulatif des commandes, utilisez, quand vous êtes sous l'éditeur, la commande :

```
esc ?
```

- pour éditer une certaine fonction qui se trouve dans un fichier, appelez l'un des éditeurs vidéo en tapant simplement :

```
? ^F fonction
```

c'est-à-dire le caractère ^F (appui sur la touche CTRL et F simultanément) suivi du nom de la fonction. Le système essaiera de localiser le fichier d'où elle provient et appellera l'un des éditeurs video disponible sur ce fichier comme avec la commande ^E.

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

Enfin le source d'un grand nombre d'utilitaires, écrits en Lisp, est disponible : éditeur, paragrapheur, aides à la mise au point, démonstration vidéo ... Lisez-les, comprenez-les et n'hésitez pas à les étendre et à les améliorer.

1.7 Le_Lisp version 15

La version 15 possède quatre grands types de différences avec la version 14 :

- une plus grande rigueur dans l'évaluation, permettant de réduire le nombre des erreurs et d'améliorer l'efficacité du compilateur.

- une meilleure gestion de la mémoire, ce qui a modifié la plupart des fonctions sur les chaînes de caractères et les vecteurs.

- une utilisation systématique des packages pour gérer les interruptions programmables et les accès aux fonctions variables, évitant toute utilisation de la fonction FLET.

- un certain nombre de fonctions ont été modifiées, en particulier l'arithmétique, et bien évidemment les erreurs signalées ont été corrigées.

1.7.1 L'évaluateur

Le_Lisp version 15 ne permet plus les indirections sur les valeurs des symboles en position fonctionnelle : toute fonction calculée doit être annoncée par l'utilisation de la fonction FUNCALL explicitement.

```
ex : ; l'ancienne définition
     (DE FOO (F L)
         (CONS (F (CAR 1)) (FOO F (CDR L))))

     ; doit s'écrire dorénavant
     (DE FOO (F L)
```

```
(CONS (FUNCALL F (CAR L)) (FOO F (CDR L))))
```

Ainsi l'erreur de capture de valeur fonctionnelle, très difficile à mettre en lumière, disparaît. De plus le compilateur, prévenu de la présence d'une fonction calculée pourra compiler les arguments au lieu d'être obligé de rappeler l'évaluateur, perdant ainsi tout le bénéfice de la compilation.

Le_Lisp version 15 demande de fournir le bon nombre d'arguments pour tous les types d'appels de fonctions. Il est obligatoire d'utiliser la forme LEXPR en cas de nombre variable d'arguments. L'erreur qui se produit en cas de mauvais nombre d'arguments pour une EXPR est :

```
** <fonction> : liaison illégale : ...
```

Le_Lisp version 14 permettait de fournir moins d'arguments que requis à l'appel d'un EXPR mais ces EXPR une fois compilées devenaient des SUBR qui testaient alors le nombre d'arguments?!?

1.7.2 La gestion de la mémoire

Tous les types d'objets Lisp sont maintenant gérés dynamiquement : nombres flottants, symboles, chaînes de caractères, vecteurs de S-expressions et cellules de liste. Le_Lisp utilise un *tas* pour gérer dynamiquement les objets de taille variable (chaînes et vecteurs). La valeur retournée par la fonction GCINFO est modifiée, et de nouvelles fonctions sur les chaînes et les vecteurs ont été introduites.

1.7.3 Les fonctions qui ont changé

Un très petit nombre de fonctions ont changé :

- RUNTIME et TIME retournent des valeurs flottantes.

- les types des caractères (TYPECH/TYPECN) sont maintenant des noms symboliques.

- les #-macros sont de type "splice-macro"

- la plupart des indicateurs système sont maintenant dans le package #:SYSTEM:...

- un certain nombre de synonymes ont disparu:
LETSEQ -> LETS, QUIT -> END, UNWIND-PROTECT -> PROTECT ...

- un certain nombre de fonctions de compatibilité avec d'autres systèmes Lisp ont disparu en standard : DECLARE, FUNCTION ...

- les fonctions numériques ont été revues en particulier des fonctions *génériques* ont été introduites. Les sources d'incompatibilité proviennent principalement de la fonction / et de ses dérivées.

1.7.4 Utilisation des packages

Le système fait une large utilisation des packages pour gérer les interruptions programmables (dont les erreurs), et rechercher les fonctions des #-macros et du terminal virtuel.

1.7.5 La production des systèmes Le_Lisp version 15

La fabrication des systèmes Le_Lisp version 15 sur les différentes machines est complètement méta-circulaire : elle utilise toujours des programmes d'expansion du code LLM3 sur une machine cible écrits en Lisp lui-même. Ces programmes d'expansion deviennent, pour les machines *à QI très faible*, de véritables compilateurs LLM3.

1.7.6 Mise à jour de Le_Lisp version 15

Le_Lisp version 15 est régulièrement mis à jour au fur et à mesure de la signalisation des erreurs ou des besoins nouveaux exprimés par les utilisateurs. La première version date du 31/Décembre/1984. Ces mises à jour sont réalisées sans introduire d'incompatibilités par rapport à la version 15 du 31/Décembre/1984.

Attention : pour ne pas avoir des problèmes de compatibilité entre les différentes versions, il ne faut jamais utiliser des traits non documentés du système. Toute différence entre le système et sa documentation ou tout point obscur de la documentation doit être signalé.

1.7.7 La version du 1/Février/1985

Voici la liste des ajouts de la version du 1/Février/85 par rapport à la version du 31/Décembre/84 :

- les caractères sont codés sur 8 bits (au lieu de 7).
- rajout des fonctions PROBEFILE, SORTP, DO*, MIN, MAX.
- rajout de l'option 0 dans les fonctions PRINTLENGTH/LINE/LEVEL.
- précision dans les valeurs retournées par QUOTIENT et MODULO, le reste est toujours positif.
- les erreurs de type ERRUDF sont récupérables.
- rajout de l'éditeur de ligne **edlin**.
- rajout de l'interface virtuelle multi-fenêtre **bitmap**.

1.8 Le_Lisp version 15.2

Cette nouvelle version du système, largement re-écrite, se veut *totalement compatible* avec la version précédente. Les erreurs recensées dans le système et sa documentation ont été corrigées.

Les nouveaux traits suivants ont été ajoutés au système :

- au niveau de l'interprète :
  - le mot clef **&nobind** comme liste des paramètres
  - la présence de fonctions de contrôle lexicales : TAGBODY et BLOCK.
  - 2 nouveaux types de bloc d'activation (TAGBODY et BLOCK)
  - l'extension des fonctions de l'arithmétique générique Naires
  - la possibilité de mélanger des fonctions compilées et des fonctions interprétées et de redéfinir à tout moment des fonctions compilées (RESETFN).
  - le rajout de l'arithmétique spécialisée sur les nombres flottants.
- au niveau du chargeur et des compilateurs :
  - nouveau chargeur documenté
  - le nouveau compilateur modulaire *complice*.
- au niveau du système :
  - la gestion des interruptions internes
  - la gestion d'une horloge temps réel
  - le multi tâches intégré
  - la récupération de la date et l'heure locale
- au niveau des bibliothèques :
  - la fonction FORMAT
  - une nouvelle arithmétique rationnelle et complexe.
  - un système virtuel multi-fenêtres
  - de nouveaux outils de mise au point.

Les extensions/clarifications majeures par rapport à la version 15 [Chailloux 85] sont signalées dans les différents chapitres par une barre verticale en marge droite.

CHAPITRE 2

L'interprète

2.1 Les Objets de Base

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

Ces objets appartiennent aux types de base suivants :

- les objets atomiques
  - symboles
  - nombres
    - entiers
    - flottants
    - à précision arbitraire
  - chaînes de caractères
- les objets composites
  - listes
  - vecteurs

Le_Lisp permet également de définir de nouveaux types : *les types étendus.*

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

Le_Lisp fonctionne principalement avec des Unités Centrales qui possèdent des pointeurs de 32 bits ce qui permet d'adresser 4 Giga objets (2^32). Parfois une limitation du matériel limite l'espace d'adressage à 24 voire 20 bits. Dans la plupart des cas l'espace adresse fourni par ce type d'unité centrale est très confortable. A titre de comparaison, le PDP10 (de DEC), cheval de bataille des systèmes Lisp des 20 dernières années, ne pouvait adresser au maximum que 256K cellules de liste.

2.1.1 Les objets atomiques

2.1.1.1 Les symboles

Ils jouent le rôle d'identificateurs et servent à nommer les variables, les fonctions et les échappements. Ils sont créés implicitement dès leur lecture dans le flux d'entrée ou explicitement par les fonctions SYMBOL, IMPLODE, CONCAT 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 longueur limitée à 128 caractères. On peut insérer dans un P-NAME des caractères spéciaux ou des délimiteurs en encadrant tous les caractères du P-NAME avec le caractère *barre de 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 9 *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 (ERRUDV).

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-listes (ADDPROP, PUTPROP, GETPROP, REMPROP et DEFPROP). Par défaut cette liste de propriétés vaut ().

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, FEXPR, MACRO et DMACRO.

L'accès à la F-VAL est réalisé au moyen des fonctions VALFN, SETFN, REMFN et GETDEF. Si le symbole ne possède pas de définition de fonction cette propriété vaut 0.

F-TYPE (abréviation de Function TYPE) qui contient le type de la fonction stockée dans la F-VAL. L'ensemble F-VAL, F-TYPE permet à l'interprète de lancer les fonctions d'une manière extrêmement rapide. L'accès en clair, du F-TYPE des symboles est réalisé au moyen des fonctions TYPEFN et SETFN. Si le symbole ne possède pas de définition de fonction cette propriété vaut ().

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 paragrapheur 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.

O-VAL (abréviation de Object VALue) qui peut contenir une S-expression

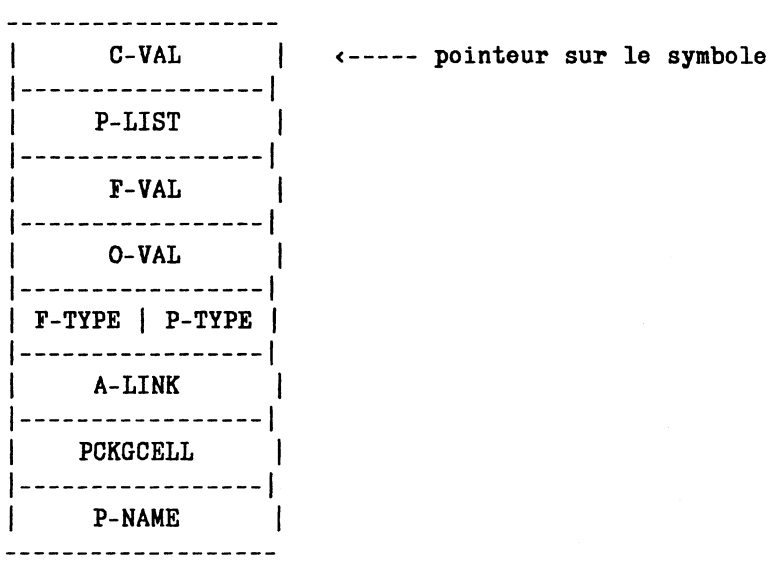
quelconque et être utilisé pour y ranger des valeurs spéciales. Ce champ est particulièrement utilisé dans les langages objet. Par défaut cette propriété vaut ().

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

PCKGCELL (abréviation de Package Cell) qui contient le nom du package auquel appartient le symbole. L'accès au package est réalisé par les fonctions SYMBOL et PACKAGECELL.

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 symboliques (qui contiennent leur propre valeur en valeur) dont voici la liste : || T LAMBDA FLAMBDA MLAMBDA SUBR0 SUBR1 SUBR2 SUBR3 NSUBR FSUBR EXPR FEXPR MACRO DMACRO QUOTE.
- les fonctions prédéfinies
- les variables système

2.1.1.2 Les nombres

Le_Lisp manipule des nombres entiers sur 16 bits (permettant de calculer dans l'intervalle : [-2^15 , +2^15-1] c'est-à-dire de -32768 à +32767, des nombres flottants sur 31, 32, 48 ou 64 bits (en fonction du système utilisé) et possède des bibliothèques de calcul en précision arbitraire.
Le P-NAME d'un nombre est la représentation de sa valeur dans la base de conversion des nombres en sortie (voir la fonction OBASE). La valeur d'un nombre est ce nombre lui-même.

2.1.1.3 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 32767 caractères actuellement. Les chaînes de caractères sont rangées dans un espace mémoire spécial, géré dynamiquement avec compaction en temps linéaire. La valeur d'une chaîne de caractères est cette chaîne elle-même, nul besoin de la *quoter* donc.
Chaque chaîne de caractères peut également avoir un type qui lui est propre. Par défaut ce type est STRING.

```
ex :  "Foo Bar"      correspond à la chaîne     Foo Bar
      """abc"""                                "abc"
      """"                                     "
      #:foo:"abc"    est une chaîne de type FOO
```

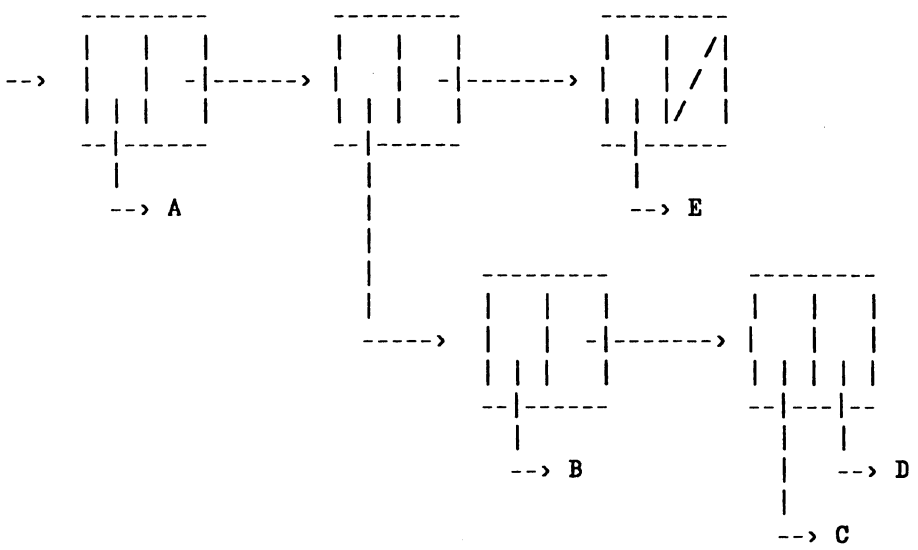
2.1.2 Les objets composites

Ils sont formés au moyen d'autres objets Lisp.

2.1.2.1 Les listes

Les listes sont représentées d'une manière standard : un élément de liste est stocké dans une *cellule de liste* constituée d'un doublet de pointeurs (un CONS) dont la partie gauche (le CAR) contient un pointeur vers l'élément, et la partie droite (le CDR) contient un pointeur sur l'élément suivant ou bien un marqueur spécial indiquant la fin de la liste. Le_Lisp utilise le symbole || (c'est-à-dire le symbole dont le nom a pour longueur 0) comme marqueur de fin de liste. Pour des raisons historiques, le symbole NIL possède toujours en valeur le marqueur de fin de liste donc pour Le_Lisp le symbole ||.

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



Le_Lisp permet *d'étiqueter* n'importe quelle cellule de liste (CONS). Cette étiquette est une marque spéciale qui peut être mise, enlevée et testée sur n'importe quelle cellule de liste (au moyen des fonctions spécialisées TCONS, TCONSP, TCONSMK et TCONSCL) qui est invisible à toutes les autres fonctions de manipulation de listes et qui ne ralentit ni ne modifie l'accès aux constituants des cellules de liste. Ces étiquettes permettent entre autres de définir de nouveaux types utilisateur (voir la section sur les types étendus).

2.1.2.2 Les vecteurs de S-expressions

Le_Lisp possède un type vecteur de S-expressions qui permet d'accéder à des objets Lisp en mode indexé. La représentation externe d'un vecteur est la suivante :

```
#[<s1> <s2> ... <sN>]
```

dans laquelle <s1> ... <sN> sont les différents éléments du vecteur. L'accès à un élément de vecteur est très rapide. Un vecteur ne peut pas dépasser 32767 éléments actuellement. Un vecteur étant un objet Lisp à part entière, il est possible de créer des vecteurs de vecteurs. Les vecteurs sont rangés dans un espace mémoire spécial, géré dynamiquement avec compaction en temps linéaire. La valeur d'un vecteur est ce vecteur lui-même, nul besoin de le *quoter* donc. Chaque vecteur peut également avoir un type qui lui est propre. Par défaut ce type est VECTOR.

```
ex :  #[a b #[x y z] d e]    est un vecteur de 5 éléments dont le
                             3ème est un vecteur de 3 éléments.
      #:foo:#[1 2]           est un vecteur de 2 éléments,
                             de type FOO
```

2.2 Le Fonctionnement de Base de l'Interprète

2.2.1 L'évaluation des objets atomiques

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 un symbole auquel on n'a pas encore donné de valeur) provoque, lors de son évaluation, l'erreur ERRUDV dont le libellé est :

```
** <fn> : variable indéfinie : <symb>      ou bien
** <fn> : undefined variable :   <symb>
```

dans lequel le nom du symbole incriminé <symb> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur (en général la fonction évaluateur EVAL ou SYMEVAL).

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. Dans ce cas les variables reçoivent une valeur qu'elles ne gardent que le temps de l'exécution d'une fonction (ce sont les paramètres des fonctions).
- comme variable rémanente *(own variable d'ALGOL).* Dans ce cas ces variables sont locales à une fonction mais ne perdent pas leur valeur entre deux invocations de cette fonction. 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 L'évaluation des objets composites

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 la *liste des arguments* de la fonction.
La valeur d'une forme est la valeur retournée par l'application de la fonction aux arguments.

La valeur d'un vecteur de S-expressions est ce même vecteur. Tout comme les nombres et les chaînes de caractères il n'est pas nécessaire de *quoter* les vecteurs qui sont toujours considérés comme des constantes.

2.3 L'évaluation des Fonctions

Une fonction (le CAR d'une forme) peut être un symbole ou une liste spéciale. Le CDR de la forme est la liste des arguments de la fonction. Si cette liste d'arguments n'est pas terminée par () ou n'est pas vide, l'erreur ERRBAL se produit dont le libellé est :

```
** <fn> : mauvaise liste d'arguments : <a>      ou bien
** <fn> : bad arguments list : <a>
```

dans lequel <fn> est le nom de la fonction invoquée et <a> la fin, non vide, de la liste des arguments, c'est-à-dire son dernier CDR.

```
ex : (CONS 1 . 2)       ->
     ** CONS : mauvaise liste d'arguments : 2
     (IF () 2 3 . 4)    ->
     ** IF : mauvaise liste d'arguments : 4
```

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

Si aucune fonction n'a été associée à ce symbole, après un traitement spécifique aux types étendus (voir cette section), l'erreur ERRUDF se produit dont le libellé est :

```
** <fn> : fonction indéfinie : <symb>      ou bien
** <fn> : undefined function : <symb>
```

dans lequel le nom du symbole incriminé <symb> est imprimé ainsi que le nom de la fonction <fn> ayant provoqué l'erreur (en général l'une des fonctions interprète EVAL, APPLY ou FUNCALL).

```
ex : (CONS 'A 'B)        ->  (A . B)
     (SETQ KONS 'CONS)   ->  CONS
     (KONS 'X 'Y)        ->
     ** EVAL : fonction indéfinie : KONS
```

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

```
ex : (3 '(1 2 3))  ->
     ** EVAL : fonction indéfinie : 3
```

Dans le cas où la fonction est une liste, il s'agit d'une déclaration explicite de fonction anonyme : le premier élément de la liste doit être l'un des symboles spéciaux LAMBDA, FLAMBDA ou MLAMBDA.

```
ex : ((LAMBDA (x) (+ x x 2)) 5)        ->  12
     ((FLAMBDA (x) x) (+ x x))         ->  (+ x x)
     ((LAMBDA (x) x) (LAMBDA (x) x))   ->  ???
     ; quelle peut être la valeur de cette évaluation ?
```

L'appel d'une fonction calculée est réalisé en utilisant la fonction FUNCALL. Lisp est l'un des rares langages dans lequel il est possible d'écrire des appels très agréables du genre :

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

Le_Lisp possède deux grandes classes de fonctions : les fonctions écrites en langage machine LLM3 et les fonctions écrites en Lisp. Toute fonction écrite en Lisp peut être traduite en langage machine LLM3 par le compilateur. Chacune de ces deux classes comprend quatre types de fonctions :
- les fonctions qui évaluent leurs arguments, Elles se nomment SUBR en LLM3 et EXPR en Lisp.
- les fonctions qui n'évaluent pas leurs arguments. Elles se nomment FSUBR en LLM3 et FEXPR en Lisp.
- les macro fonctions simples. Elles se nomment MSUBR en LLM3 et MACRO en Lisp.
- les macro fonctions remplaçantes. Elles se nomment DMSUBR en LLM3 et DMACRO en Lisp.

2.3.1 Les SUBR

Les SUBR sont des fonctions écrites en langage machine LLM3. Ces fonctions possèdent des arguments qui sont *toujours évalués.* Il existe des SUBR à nombre fixe d'arguments (SUBR0 SUBR1 SUBR2 SUBR3) 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 l'erreur ERRWNA apparait dont le libellé est :

```
** <fn> : mauvais nombre d'arguments : <n>      ou bien
** <fn> : wrong number of arguments : <n>
```

dans lequel le nom de la SUBR incriminée <fn> est imprimé ainsi que le nombre d'arguments requis <n> pour cette fonction.

Pour les SUBR à nombre variable d'arguments (NSUBR) l'interprète teste parfois le nombre minimum d'arguments à l'appel et peut provoquer une erreur identique à celle des SUBR (ERRWNA).

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

```
Ex : (CONS)           =>  ** CONS : mauvais nombre d'arguments : 2
     (CONS 'A)        =>  ** CONS : mauvais nombre d'arguments : 2
     (CONS 'A 'B)     =>  (A . B)
     (CONS 'A 'B 'C) =>  ** CONS : mauvais nombre d'arguments : 2
     (APPLY 'CONS '())          =>  ** CONS : mauv......ments : 2
     (APPLY 'CONS '(A))         =>  ** CONS : mauv......ments : 2
     (APPLY 'CONS '(A B))       =>  (A . B)
     (APPLY 'CONS '(A B C))     =>  ** CONS : mauv......ments : 2
     (FUNCALL 'CONS)            =>  ** CONS : mauv......ments : 2
     (FUNCALL 'CONS '(A))       =>  ** CONS : mauv......ments : 2
     (FUNCALL 'CONS '(A B))     =>  (A . B)
     (FUNCALL 'CONS '(A B C))   =>  ** CONS : mauv......ments : 2
```

2.3.2 Les FSUBR

Les FSUBR sont également des fonctions écrites en langage machine, exécutées extrêmement rapidement et résidentes dans l'interprète dès son lancement. Ces fonctions possèdent des arguments en nombre variable qui ne sont *jamais évalués* par l'interprète mais le cas échéant par la fonction elle-même. 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.

Il est possible de rajouter des fonctions de ce type en les écrivant directement en langage machine LLM3 ou en faisant compiler des FEXPR.

2.3.3 Les MSUBR

Les MSUBR sont également des fonctions écrites en langage machine LLM3. Elles proviennent de la compilation de MACRO. Ces fonctions possèdent des arguments en nombre variable qui ne sont *jamais évalués* par l'interprète.

2.3.4 Les DMSUBR

Les DMSUBR sont également des fonctions écrites en langage machine LLM3. Elles proviennent de la compilation de DMACRO. Ces fonctions possèdent des arguments en nombre variable qui ne sont *jamais évalués* par l'interprète.

2.3.5 Les EXPR

Les EXPR sont des fonctions écrites en Lisp et interprétées par les fonctions d'évaluation (EVAL, APPLY et FUNCALL). Ces fonctions possèdent, à leur définition, un certain nombre de *paramètres* représentés par une liste (ou un arbre) de variables <lvar> ainsi qu'un corps de fonction constitué d'une suite quelconque d'expressions <s1> ... <sN> à évaluer.

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 l'indicateur d'une fonction de type EXPR, <lvar> est la liste (ou l'arbre) des paramètres et <s1> ... <sN> le corps de la fonction.

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

La définition des fonctions de type EXPR (c'est-à-dire l'association d'une LAMBDA-expression à un symbole) est réalisée au moyen de la fonction DE.

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

1 - liaison des valeurs des arguments aux paramètres de la fonction après avoir sauvé les anciennes valeurs des noms des paramètres dans la pile. L'appel d'une fonction de type EXPR a lieu *par valeur.*

2 - évaluation des différentes expressions du corps <s1> ... <sN>. La valeur retournée par la fonction sera la valeur de la dernière évaluation (c'est-à-dire celle de <sN>)

3 - destruction des liaisons effectuées en 1 - et restitution des anciennes valeurs des noms des paramètres sauvées dans la pile.

La liaison des valeurs des arguments aux paramètres de la fonction utilise une procédure récursive qui considère que la liste des paramètres <lvar> est un arbre. La liaison des valeurs des arguments aux paramètres de la fonction va s'effectuer entre les feuilles de l'arbre des paramètres et la liste des valeurs des arguments. Ce type de liaison permet d'éclater la valeur d'un argument dans différentes variables à l'appel d'une fonction.

S'il manque ou s'il reste des éléments dans la liste des valeurs des arguments après les liaisons, l'erreur ERRWNA apparaît dont le libellé est :

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

dans lequel <fn> est le nom de la fonction appelée (ou bien LAMBDA en cas de fonction anonyme) et <s> le reste de la liste des valeurs des arguments s'il y en avait de trop et () s'il en manquait.

Si une valeur ne peut pas être liée à un arbre de paramètres (c'est-à-dire si une valeur atomique de la liste des valeurs doit être liée à un paramètre non atomique), l'erreur ERRILB apparaît dont le libellé est :

```
** <fn> : liaison illégale : (<p> <v>)      ou bien
** <fn> : illegal bind : (<p> <v>)
```

dans lequel <fn> est le nom de la fonction appelée (ou bien LAMBDA en cas de fonction anonyme) et (<p> <v>) une liste formée de l'arbre des paramètres <p> et de la valeur <v> incriminée.

Si l'arbre des paramètres ne contient pas que des symboles de type variable, l'erreur ERRBPA apparaît dont le libellé est :

```
** <fn> : mauvais paramètre : <s>      ou bien
** <fn> : bad parameter : <s>
```

dans lequel <fn> est le nom de la fonction appelée (ou bien LAMBDA en cas de fonction anonyme) et <s> le paramètre défectueux.

Voici, décrit en Lisp, ce mécanisme de liaison généralisée :

```
(DE BINDVAR (tvar lval)
    ; tvar : l'arbre des paramètres
    ; lval : la liste des valeurs des arguments
    ; empile (au moyen de la fonction PUSH) les
    ; suites  "valeur - variable"
    (COND ((NULL tvar)
               (WHEN lval
                     (ERROR 'EVAL 'ERRWNA lval)))
          ((VARIABLEP tvar)
              (PUSH (SYMEVAL tvar))
              (PUSH tvar)
              (SET tvar lval))
          ((CONSP tvar)
               (IF (ATOM lval)
                   (ERROR 'EVAL 'ERRILB (LIST tvar lval))
                   (BINDVAR (CAR tvar) (CAR lval))
                   (BINDVAR (CDR tvar) (CDR lval))))
```

```
        (T (ERROR 'EVAL 'ERRBPA tvar))))
```

Exemples de liaison des EXPR

```
((LAMBDA (x y z) (LIST x y z)) (1+ 1) (1+ 2) (1+ 3))
-> (2 3 4)

((LAMBDA (x y z) (LIST x y z)) (1+ 1))
** LAMBDA : mauvais nombre d'arguments : ()

((LAMBDA (x y z) (LIST x y z)) 1 2 3 4 5)
** LAMBDA : mauvais nombre d'arguments : (4 5)

((LAMBDA x x) (1+ 1) (1+ 2) (1+ 3))
-> (2 3 4)

((LAMBDA (x y . z) (LIST x y z))
    (1+ 1) (1+ 2) (1+ 3) (1+ 4))
-> (2 3 (4 5))

((LAMBDA ((x . y) . z) (LIST x y z)) (CONS 1 2) 'C))
-> (1 2 (C))

((LAMBDA (x (y . z)) (LIST x y z)) 'a 'b)
** LAMBDA : liaison illégale : ((y . z) b)

((LAMBDA (t 1) (LIST x)) 'a 'b)
** LAMBDA : mauvais paramètre : t

((LAMBDA (x 1) (LIST x)) 'a 'b)
** LAMBDA : mauvais paramètre : 1
```

2.3.6 Les EXPR &nobind

Si la liste des paramètres d'une fonction de type EXPR se compose uniquement de la clef **&nobind** la liaison s'opère d'une manière différente. Aucune variable n'est liée. Le nombre d'argument de l'appel est retourné par l'appel de la fonction (ARG) et les différents arguments sont recherchés au moyen de la fonction (ARG <n>), <n> étant le numéro de l'argument compté à partir de 0. Ce trait permet de définir des fonctions à nombre variable d'arguments sans fabriquer une liste des valeurs des arguments.
Cette constructions s'apparente aux LEXPR de MacLisp.

```
ex : ? (DE GOG &NOBIND
     ?    (LIST (ARG) (ARG 0) (ARG 1)))
     = GOG
     ? (GOG 10 11)
     = (2 10 11)
     ? (GOG 10 11 12 13 14 15)
     = (6 10 11)
     ? (DE MAGOG &NOBIND
     ?     (IF (> (ARG) 2)
     ?         (ERROR 'MAGOG "pas en lecture/écriture" (ARG))
     ?         (LET ((arg1 (IF (< (ARG) 1) 'defaul1 (ARG 0)))
     ?               (arg2 (IF (< (ARG) 2) 'defaul2 (ARG 1))))
     ?              (LIST arg1 arg2))))
     = MAGOG
     ? (MAGOG)
     = (defaut1 defaut2)
```

```
? (MAGOG 10)
= (10 defaut2)
? (MAGOG 10 11)
= (10 11)
? (MAGOG 10 11 12 13)
                    ** MAGOG : pas en lecture/écriture : 4
```

2.3.7 Les FEXPR

Les FEXPR sont des fonctions écrites en Lisp et interprétées par les fonctions standard d'évaluation (EVAL, APPLY ou FUNCALL). Tout comme les EXPR ces fonctions possèdent des paramètres en nombre quelconque, qui sont décrits dans une liste (ou un arbre) <lvar> de paramètres 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 FLAMBDA-expression, de la forme :

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

dans laquelle le symbole FLAMBDA est un indicateur de fonction de type FEXPR, <lvar> est la liste (ou l'arbre) des paramètres et <s1> ... <sN> le corps de la fonction.

ex : `(FLAMBDA (var val) (SET var (EVAL val)))`

La définition des fonctions de type FEXPR (c'est-à-dire l'association d'une FLAMBDA-expression à un symbole) est réalisée au moyen de la fonction DF.

Ces fonctions ne diffèrent des EXPR que par le mode de liaison des paramètres : c'est l'ensemble des arguments *non évalués,* c'est-à-dire le CDR de la forme elle-même, qui est lié à la liste des paramètres <lvar> en utilisant la même procédure BINDVAR que pour les EXPR.

L'appel de ces fonctions peut provoquer les mêmes erreurs que pour les EXPR : ERRWNA, ERRILB, ERRBPA.

2.3.8 Les MACRO

L'interprète reconnaît un autre type de fonctions, les MACRO. Tout comme les EXPR et les FEXPR, ces fonctions écrites en Lisp possèdent des paramètres en nombre quelconque, qui sont rangés dans une liste <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 MLAMBDA-expression, de la forme :

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

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

ex : `(MLAMBDA (n var) (LIST 'SETQ var (LIST 'CDR var)))`

La définition des fonctions de type MACRO (c'est-à-dire l'association d'une MLAMBDA-expression à un symbole) est réalisée au moyen de la fonction DM.

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 (évidemment non

évaluée) comme argument en utilisant la procédure BINDVAR, 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 pris comme argument. Il est donc possible de réaliser des modifications physiques de la forme elle-même à la première évaluation de la MACRO.

L'utilisation des MACRO, véritable sport en soi, permet d'étendre le langage très facilement et très puissamment, Lisp servant à la fois de langage de description et d'expansion des MACRO.

L'appel de ces fonctions peut provoquer les mêmes erreurs que pour les EXPR : ERRWNA, ERRILB, ERRBPA.

2.3.9 Les DMACRO

L'interprète reconnaît un deuxième type de MACRO, les DMACRO. La définition des fonctions de type DMACRO est réalisée au moyen de la fonction DMD.

L'évaluation d'une fonction de ce type se distingue de l'évaluation des MACRO par deux traits :
- c'est le CDR de la forme (non évalué) qui est utilisé comme argument (à la manière des FEXPR),
- au retour de la première évaluation la forme toute entière est remplacée physiquement par la valeur retournée au moyen de la fonction DISPLACE (d'où le nom de Displace MACRO).

L'utilisation de ce type de MACRO, bien que moins générale que les MACRO précédentes, couvre en fait la plupart des utilisations des MACRO. Elles sont donc les plus largement utilisées.

L'appel de ces fonctions peut provoquer les mêmes erreurs que pour les EXPR : ERRWNA, ERRILB, ERRBPA.

2.4 La Définition des Fonctions

Il existe deux types d'utilisation des fonctions :

- une utilisation globale : les fonctions sont définies d'une manière globale et gardent leur définition tant qu'on ne les modifie pas explicitement. Ce type de définition ne permet pas de retrouver dynamiquement les définitions antérieures (voir les fonctions DE/DF/DM/DMD)

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

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 prédéfinies qui permettent de réaliser des définitions statiques ou dynamiques à peu de frais (voir la section : définition des fonctions du chapitre suivant).

2.5 Les Packages

Le_Lisp possède un espace de noms multiples : chaque symbole, unique dans le système, possède un nom externe (pname) qui n'est pas forcément unique et un nom de package auquel il appartient. Un nom de package est également un symbole qui peut à son tour posséder un nom de package ... Le package global est le package ||. Il est donc possible de gérer des *hiérarchies de noms.* Dans Le_Lisp, les packages permettent :
- de partager l'espace des noms entre plusieurs programmes,
- d'invoquer des fonctions dans des espaces de noms spécifiques,
- de réaliser très économiquement des *lancements de méthodes avec hiérarchie simple à la SmallTalk.*

2.6 Les Types Etendus

Le_Lisp permet de définir d'autres types d'objets. Ces types sont toujours construits au moyen d'une cellule de liste étiquetée (TCONS) dont le CAR contient le nom symbolique du type étendu et le CDR sa valeur, d'un vecteur typé ou d'une chaîne typée.
Ces types étendus sont reconnus par l'évaluateur et l'imprimeur Le_Lisp.

Quand l'évaluateur (EVAL) évalue une forme dont le premier élément est une cellule de liste étiquetée dont le CAR est un symbole ou s'il doit évaluer un vecteur typé ou une chaîne typée, il cherche s'il n'existe pas de fonction définie de nom EVAL dans le package resultant de la fonction TYPE-OF sur son argument.
Si cette fonction existe, elle est appelée par EVAL au lieu de provoquer une erreur. Cette recherche utilise la fonction GETFN avec un 3ème argument égal à (). La remontée hiérarchique dans les packages s'arrêtera au package global exclu, ce qui évite de faire boucler l'évaluateur.
Attention : la recherche de la fonction d'évaluation a lieu après l'évaluation normale. Il est donc impossible actuellement d'utiliser des noms de fonctions définies comme nom de type étendu.

Quand l'imprimeur (PRINT) doit éditer un objet Lisp, il cherche s'il n'existe pas de fonction définie de nom PRIN dans le package dont le nom est le résultat de la fonction TYPE-OF appliquée à son argument. Si cette fonction existe, elle est appelée par PRINT au lieu d'imprimer l'objet d'une manière standard. Cette recherche utilise la fonction GETFN avec un 3ème argument égal à (). La remontée hiérarchique dans les packages s'arrêtera au package global exclu, ce qui évite de faire boucler l'imprimeur.
Le fonctionnement de l'imprimeur peut se décrire en Lisp :

```
(IF (GETFN (TYPE-OF x) 'PRIN)
    (FUNCALL (GETFN (TYPE-OF x) 'PRIN ()) x)
    (PRIN x))
```

A titre d'exemple définissons le type étendu *liste-de-chiffres.* L'entrée d'une telle liste se fera au moyen de la #-macro <>.

```
#<cccccccc>
```

dans laquelle les chiffres ..ccc.. sont encadrés des délimiteurs <>.

Voici la définition en Lisp de la fonction de lecture :

```
? (DEFSHARP |<| ()
?    (LIST (LET (x c v)
?               (WHILE (NEQ (SETQ c (READCN)) #/>)
?                      (NEWL x c))
?               (SETQ v (APPLY 'VECTOR (NREVERSE x)))
?               (TYPEVECTOR v 'LISTE-DE-CHIFFRES)
?               v)))
= <
? ; la lecture suivante produira :
? '#<01234>
= #:LISTE-DE-CHIFFRES:#[48 49 50 51 52]
? (TYPE-OF '#<123>)
= LISTE-DE-CHIFFRES
? ; le quote ' est obligatoire car le toplevel évalue ce qui est lu :
? #<1234>
= ** eval : fonction indefinie : liste-de-chiffres
? ; pour remédier à cela, définissons la fonction d'évaluation
? ; de ce type étendu la plus simple : la fonction identité
? (DE #:LISTE-DE-CHIFFRES:EVAL (x) x)
= #:LISTE-DE-CHIFFRES:EVAL
? ; Voyons si les listes de chiffres sont devenues des constantes ?
? #<01234>
= #"LISTE-DE-CHIFFRES:#[48 49 50 51 52]
? ; définissons maintenant l'imprimeur de ce type étendu.
? (DE #:LISTE-DE-CHIFFRES:PRIN (x)
?     (PRIN "#<")
?     (MAPVECTOR 'PRINCN x)
?     (PRIN ">"))
= #:LISTE-DE-CHIFFRES:PRIN
? ; essayons le passage : READ -> EVAL -> PRINT
? #<0246>
= #<0246>
? (CONS #<0234> #<067>)
= (#<0234> . #<067>)
```

2.7 La 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 oeuvre 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.

```
; package spécial
(DEFVAR #:sys-package:colon 'eval)

; boucle principale
(DE :TOPLEVEL ()
    (TAG :ERROR
         (PRINT "<Toplevel>")
         (SETQ IT (LET ((STACK))
                       (:EVAL (WITH ((PROMPT ":? ")) (READ)))))
         (PRINT ":= " IT))
    (:TOPLEVEL))

; traitement des erreurs de l'interprète
(DEFVAR #:system:error-flag t)
(DEFVAR #:system:debug t)

(DE :ERROR (f m a)
    (WHEN #:system:error-flag (PRINT "** " f " : " m " : " a))
    (EXIT :ERROR))

  ; trace globale de cet interprète
(DE :TRACE ()
    (TRACE :EVAL :EVLIS :EPROGN :EVALAMBDA :BINDVAR :UNBINVAR
           :APPLY :FUNCALL))

(DE :UNTRACE ()
    (UNTRACE))

  ; évaluation d'une forme quelconque
(DE :EVAL (forme)
    (COND
      ((OR (NUMBERP forme) (VECTORP forme) (STRINGP forme))
          (IF (GETFN (TYPE-OF forme) ':EVAL ())
              (:FUNCALL (GETFN (TYPE-OF forme) ':EVAL ()) forme)
              forme))
      ((SYMBOLP forme)
         (IF (BOUNDP forme)
             (SYMEVAL forme)
             (:ERROR ':EVAL 'ERRUDV forme)))
      (T (LET ((fonct (CAR forme)) (lval (CDR forme)))
           (COND
              ((SYMBOLP fonct)
                  (COND ((TYPEFN fonct)
                             (:EVALINTERNAL (TYPEFN fonct)
                                            (VALFN fonct)
```

```
                                        lval))
                                ((AND (TCONSP forme) (GETFN fonct ':EVAL ()))
                                    (:FUNCALL (GETFN fonct ':EVAL ()) forme))
                                (T (:EVAL (CONS (:ERROR ':EVAL 'ERRUDF fonct)
                                                larg)))))
                        ((OR (NUMBERP fonct) (VECTORP fonct) (STRINGP fonct))
                            (:EVAL (CONS (:ERROR ':EVAL 'ERRUDF fonct) larg)))
                        ((EQ (CAR fonct) 'LAMBDA)
                            (:EVALINTERNAL 'EXPR (CDR fonct) lval))
                        ((EQ (CAR fonct) 'FLAMBDA)
                            (:EVALINTERNAL 'FEXPR (CDR fonct) lval))
                        ((EQ (CAR fonct) 'MLAMBDA)
                            (:EVALINTERNAL 'MACRO (CDR fonct) lval))
                        (T (:EVAL (CONS (:ERROR ':EVAL 'ERRUDF fonct) larg))))))))
```

; évaluation d'une fonction suivant son type

```
(DE :EVALINTERNAL (ftype fval lval)
    (SELECTQ ftype
      ((SUBR0 SUBR1 SUBR2 SUBR3)
         (CALL fval (:EVAL (CAR lval))
                    (:EVAL (CADR lval))
                    (:EVAL (CADDR lval))))
      (NSUBR (CALLN fval (:EVLIS lval)))
      (MSUBR (CALL fval forme () ()))
      (DMSUBR (DISPLACE forme (CALL fval (CDR forme) () ()))
      (FSUBR
         ; toutes les FSUBRS sont interprétées directement
         ; car elles peuvent re appeler EVAL.
         ; Seules les plus importantes sont définies ici.
         (SELECTQ fonct
             (QUOTE  (car lval))
             (IF     (IF (:EVAL (CAR lval))
                         (:EVAL (CADR lval))
                         (:EVAL (CADDR lval))))
             (PROGN  (:EPROGN lval))
             (DE     (SETFN (CAR lval) 'EXPR    (CDR lval)))
             (DF     (SETFN (CAR lval) 'FEXPR   (CDR lval)))
             (DM     (SETFN (CAR lval) 'MACRO   (CDR lval)))
             (DMD    (SETFN (CAR lval) 'DMACRO (CDR lval)))
             (SETQ   (UNTIL lval
                            (SET (NEXTL lval)
                                 (:EVAL (NEXTL lval)))))
             (T      (:ERROR 'EVAL 'ERRUDF fonct))))
      (EXPR    (:EVALAMBDA fval (:EVLIS lval)))
      (FEXPR   (:EVALAMBDA fval lval))
      (MACRO   (:EVAL (:EVALAMBDA fval forme)))
      (DMACRO  (:EVAL (DISPLACE forme
                            (:EVALAMBDA fval (CDR forme)))))
      (T       (:EVAL (CONS (:ERROR ':EVAL 'ERRUDF fonct)
                            lval))))))))
```

; construit la liste des valeurs des évaluations
; de tous les éléments de l

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

; évalue le corps l

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

; applique une F-VAL de type : (<lvar> <s1> ... <sN>)
; à la liste d'arguments lval ;

```
(DE :EVALAMBDA (fval lval)
    (:PUSH ())
    (:BINDVAR (CAR fval) lval)
    (PROTECT (:EPROGN (CDR fval))
             (:UNBINDVAR)))
```

; réalise une liaison d'arbre généralisée

```
(DE :BINDVAR (lvar lval)
    (COND ((NULL lvar)
           (WHEN lval
                 (:ERROR ':EVAL 'ERRWNA lval)))
          ((VARIABLEP lvar)
           (:PUSH (IF (BOUNDP lvar) (SYMEVAL lvar) '<undef>))
           (:PUSH lvar)
           (SET lvar lval))
          ((CONSP lvar)
           (IF (ATOM lval)
               (:ERROR ':EVAL 'ERRILB (LIST lvar lval))
               (:BINDVAR (CAR lvar) (CAR lval))
               (:BINDVAR (CDR lvar) (CDR lval))))
          (T (:ERROR ':EVAL 'ERRBPA lvar))))

(DE :UNBINDVAR ()
    (UNTILEXIT :DONE
               (SET (OR (:POP) (EXIT :DONE)) (:POP))))
```

; avec les fonctions de manipulation de pile

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

(DE :POP () (NEXTL stack))
```

; et pour les fanatiques, un véritable :APPLY
; applique la fonction fonct aux arguments lval

```
(DE :APPLY (fonct lval)
    (COND
      ((SYMBOLP fonct)
          (:APPLYINTERNAL (TYPEFN fonct)
                          (VALFN fonct)
                          lval))
      ((OR (NUMBERP fonct) (VECTORP fonct) (STRINGP fonct))
          (APPLY (:ERROR ':APPLY 'ERRUDF fonct) lval))
```

```
      ((EQ (CAR fonct) 'LAMBDA)
          (:APPLYINTERNAL 'EXPR fonct lval))
      ((EQ (CAR fonct) 'FLAMBDA)
          (:APPLYINTERNAL 'FEXPR fonct lval))
      ((EQ (CAR fonct) 'MLAMBDA)
          (:APPLYINTERNAL 'MACRO fonct lval))
      (T (:APPLY (:ERROR ':APPLY 'ERRUDF fonct) lval))))
```

; application d'une fonction suivant son type

```
(DE :APPLYINTERNAL (ftyp fval lval)
    (SELECTQ ftyp
      ((SUBR0 SUBR1 SUBR2 SUBR3)
        (CALL fval (CAR lval) (CADR lval) (CADDR lval) ))
      (NSUBR  (CALLN fval lval))
      (MSUBR  (CALL fval (CONS fonct lval) () ()))
      (DMSUBR (CALL fval lval () ()))
      (FSUBR  (CALL fval lval () () ))
      (EXPR   (:EVALAMBDA fval lval))
      (FEXPR  (:EVALAMBDA fval (LIST lval)))
      (MACRO  (:EVAL (:EVALAMBDA fval (CONS fonct lval))))
      (DMACRO (:EVAL (DISPLACE forme (:EVALAMBDA fval lval))))
      (T (APPLY (:ERROR ':APPLY 'ERRUDF fonct) lval))))
```

; application à la FUNCALL

```
(DE :FUNCALL (fonct . larg)
    (:APPLY fonct larg))
```

CHAPITRE 3

Les fonctions prédéfinies

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 d'arguments attendus, 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)
- <symb> pour un symbole
- <n> pour un nombre
- <strg> pour une chaîne de caractères ou un objet qui peut se convertir en chaîne au moyen de la fonction STRING.
- <vect> pour un vecteur de S-expressions
- <ch> pour un caractère (c'est-à-dire le premier caractère d'une chaîne de caractères ou d'un objet qui peut se convertir en chaîne de caractères au moyen de la fonction STRING)
- <cn> pour le code interne 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, DM ou DMD. Ces descriptions ne seront que les équivalents Lisp de ces fonctions et ne représenteront que leurs caractéristiques fondamentales.

De nombreux tests de type sont réalisés par ces fonctions. Les erreurs suivantes peuvent apparaître :

- ERRNAA dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas un atome : <e>      ou bien
** <fn> : not an atom : <e>
```

- ERRNLA dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas une liste : <e>      ou bien
** <fn> : not a list : <e>
```

- ERRSYM dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas un symbole : <e>      ou bien
** <fn> : not a symbol : <e>
```

- ERRNVA dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas une variable : <e>      ou bien
** <fn> : not a variable : <e>
```

- ERRNSA dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas une chaîne : <e>      ou bien
** <fn> : not a string : <e>
```

dans lequel le nom de l'argument défectueux <s> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur.

3.1 Les Fonctions d'Evaluation

(EVAL <s> <env>) [SUBR à 1 ou 2 arguments]

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). Le 2ème argument <env>, s'il est fourni, est un environnement lexical fourni par la fonction STEPEVAL (voir la section 5 du chapitre 7).

```
ex : (EVAL '(1+ 55))                          ->  56
     (EVAL (LIST '+ 8 '(1+ 3) 3))             ->  15
     (EVAL (LIST (CAR '(CDR)) ''(A B C)))     ->  (B C)
```

(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 (c'est-à-dire celle du dernier élément de <l>). Si <l> n'est pas une liste bien formée, l'erreur ERRBAL se déclenche.

EPROGN peut être défini en Lisp de la manière suivante :

```
(DE EPROGN (l)
    (IF (ATOM l)
        (IF (NULL l)
            ()
            (ERROR 'EPROGN 'ERRBAL l))
```

```
                (IF (ATOM (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)  123  -> 3
```

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

évalue séquentiellement les différentes expressions <s1> ... <sN>. PROG1 retourne la valeur de la première évaluation (c'est-à-dire celle de <s1>).

PROG1 peut être défini en Lisp de la manière suivante :

```
(DF PROG1 (prem . rest)
    (LET ((result (EVAL prem)))
         (EPROGN rest)
         result))
```

; ou bien sous la forme d'une EXPR :

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

```
ex : (PROG1 (PRIN 1) (PRIN 2) (PRIN 3))  123  -> 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 (exch var1 var2)
    (LIST 'SETQ var1
          (LIST 'PROG1 var2
                (LIST 'SETQ var2 var1))))))
```

; avec cette MACRO un appel de type :

```
(EXCH v1 v2)
```

; est expansé en :

```
(SETQ v1 (PROG1 v2 (SETQ v2 v1)))
```

(PROG2 <s1> <s2> ... <sN>) [FSUBR]

évalue séquentiellement les différentes expressions <s1> <s2> ... <sN>. PROG2 retourne la valeur de la deuxième évaluation (c'est-à-dire celle de <s2>).

PROG2 peut être défini en Lisp de la manière suivante :

```
(DF PROG2 (prem second . rest)
    (EVAL prem)
    (LET ((result (EVAL second)))
         (EPROGN rest)
         result))
```

; ou bien sous la forme d'une EXPR :

```
(DE PROG2 (prem second . l) second)
```

```
ex : (PROG2 (PRIN 1) (PRIN 2) (PRIN 3))  123  -> 2
```

Comme PROG1, PROG2 est toujours utilisé pour réaliser des effets de bord.

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

évalue en séquence les différentes expressions <s1> ... <sN>. PROGN retourne la valeur de la dernière évaluation (c'est-à-dire 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))   123  ->  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 prédéfini qui facilite cette écriture, le caractère apostrophe *(quote)* ' . Ce caractère placé devant une expression quelconque <s> fabriquera à la lecture la liste (QUOTE <s>).
Pour des raisons de lisibilité, les fonctions de sortie impriment les listes de la forme (QUOTE <s>) en utilisant cette même notation '<s>.

QUOTE peut être défini en Lisp de la manière suivante :

```
(DF QUOTE (s) s)
```

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

(FUNCTION <fn>) [FSUBR]

si un environnement lexical est présent retourne la *fermeture lexicale* composée de la fonction <fn> et de cet environnement lexical. S'il n'y a pas d'environnement lexical, FUNCTION est équivalent à QUOTE. Le_Lisp v15.2 ne construit un environnement lexical que pour les étiquettes définies par la fonction TAGBODY et les noms des blocs définis par la fonction BLOCK. Cette fonction ne peut pas être décrite en Lisp.

(ARG <n>) [SUBR à 0 ou 1 argument]

est utilisée à l'intérieur d'une EXPR &nobind. Sans argument, (ARG) retourne le nombre d'arguments du dernier appel d'une fonction de type EXPR &nobind. Avec un argument numérique (ARG <n>) retourne la valeur du <n>ième argument du dernier appel d'une fonction de type EXPR &nobind. Le numéro du premier argument est 0. Si <n> est plus grand que le nombre d'arguments le résultat est indéterminé, ARG n'effectuant aucun test de validité.

(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 plutôt écrits précédés 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
```

3.2 Les Fonctions d'Application

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

la valeur d'une lambda-expression de type LAMBDA, FLAMBDA ou MLAMBDA est cette lambda-expression elle-même. Ces nouvelles fonctions ont été rajoutées 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))
```

FLAMBDA peut être défini en Lisp de la manière suivante :

```
(DF FLAMBDA l (CONS 'FLAMBDA l))
```

MLAMBDA peut être défini en Lisp de la manière suivante :

```
(DF MLAMBDA l (CONS 'MLAMBDA l))
```

; ou bien sous la forme d'une MACRO

```
(DM LAMBDA l (KWOTE l))
```

; cette définition est plus exacte car elle préserve l'adresse
; physique de la LAMBDA-expression.

```
ex : (LAMBDA (x) x)                -> (LAMBDA (x) x)
     (MAPC (LAMBDA (x) (PRIN x))
```

```
'(A B C)) ABC                -> ()
```

3.2.1 Les fonctions d'application simple

(APPLY <fn> <s1> ... <sN> <l>) [SUBR à 2 ou N arguments]

Les arguments <s1> ... <sN> sont optionnels. APPLY retourne la valeur de l'application de la fonction <fn> à la liste d'arguments <l>, augmentée des arguments <s1> ... <sN> s'ils sont présents. L'argument <l> doit être une liste (peut-être vide) sous peine de provoquer l'erreur ERRBAL. La fonction <fn> peut être une fonction de n'importe quel type SUBR, NSUBR, FSUBR, MSUBR, DMSUBR, EXPR, FEXPR, MACRO ou DMACRO.

```
ex : (APPLY 'CONS (LIST (1+ 1) (1+ 2)))     -> (2 . 3)
     (APPLY 'CONS (1+ 1) (LIST (1+ 3)))     -> (2 . 4)
     (APPLY 'LIST 1 2 3 '(4 5))             -> (1 2 3 4 5)
     (APPLY (LAMBDA (x y) (+ x y)) (LIST (1+ 8) (- 10 3)))
     -> 16
     (APPLY (FLAMBDA (x y) (CONS x y)) '((1+ 1) (1- 2)))
     -> ((1+ 1) 1- 2)
```

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

est une autre forme de la fonction APPLY.
L'appel (FUNCALL <fn> <s1> ... <sN>) est équivalent
à l'appel (APPLY <fn> (LIST <s1> ... <sN>)).
toutefois FUNCALL ne construit pas effectivement de liste et est donc bien plus économique qu'APPLY.

FUNCALL peut être défini en Lisp de la manière suivante :

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

```
ex : (FUNCALL (LAMBDA (x y) (CONS x y)) 'A 'B)   -> (A . B)
     (FUNCALL '+ (1+ 1) (1+ 2) (1+ 3))           -> 9
     (SETQ KONS 'CONS)                           -> KONS
     (FUNCALL KONS (1+ 1) (1+ 2))                -> (2 . 3)
```

3.2.2 Les fonctions d'application de type MAP

Ces fonctions permettent d'appliquer une fonction successivement à un ensemble de listes arguments. La fonction appliquée peut avoir un nombre quelconque d'arguments, et les applications cessent quand l'une des listes arguments se termine. Pour pouvoir appliquer des arguments *invariants,* il suffit de les inclure dans une liste circulaire par exemple au moyen de la fonction CIRLIST. Bien évidemment au moins l'une de ces listes arguments doit pouvoir se terminer pour éviter de faire boucler ces fonctions d'application.

Dans les descriptions en Lisp accompagnant ces fonctions nous utiliserons les fonctions auxiliaires suivantes :

```
(DE ALLCAR (e)
    ; e : une liste de listes
```

```
    ; retourne la liste de tous leur CAR
    (IF (NULL e) () (CONS (CAAR e) (ALLCAR (CDR e)))))

(DE ALLCDR (e)
    ; e : une liste de listes
    ; retourne la liste de tous leur CDR
    (IF (NULL e) () (CONS (CDAR e) (ALLCDR (CDR e)))))
```

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

applique la fonction <fn> avec toutes les listes <li> comme arguments puis avec tous les CDR de ces listes jusqu'à ce qu'une de ces listes se termine. MAPL retourne toujours () en valeur. MAP est un ancien nom strictement synonyme à MAPL.

MAPL peut être défini en Lisp de la manière suivante :

```
(DE MAPL (f . l)
    (WHEN (EVERY 'CONSP l)
          (APPLY f l)
          (APPLY 'MAPL f (ALLCDR l))))
```

```
ex : (MAPL 'PRINT '(A (B C) D) '(X Y Z))
     (A (B C) D)(X Y Z)
     ((B C) D)(Y Z)
     (D)(Z)
     -> ()
```

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

applique 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'à ce qu'une de ces listes se termine. MAPC retourne toujours () en valeur.

MAPC peut être défini en Lisp de la manière suivante :

```
(DE MAPC (f . l)
    (WHEN (EVERY 'CONSP l)
          (APPLY f (ALLCAR l))
          (APPLY 'MAPC f (ALLCDR l))))
```

```
ex : (MAPC 'PRINT '(A (B C) D) '(X Y Z))
     AX
     (B C)Y
     DZ
     -> ()
```

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

applique la fonction <fn> avec toutes les listes <li> comme arguments puis avec tous les CDR de ces listes jusqu'à ce qu'une de ces listes se termine. MAPLIST est donc identique fonctionnellement à la fonction MAP 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 (f . l)
    (WHEN (EVERY 'CONSP l)
          (CONS (APPLY f l)
                (APPLY 'MAPLIST f (ALLCDR l)))))
```

```
ex : (MAPLIST 'LENGTH '(A (B C) D))   -> (3 2 1)
```

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

applique 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'à ce qu'une de ces listes se termine. 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 (f . l)
    (WHEN (EVERY 'CONSP l)
          (CONS (APPLY f (ALLCAR l))
                (APPLY 'MAPCAR f (ALLCDR l)))))
```

```
ex : (MAPCAR 'CONS '(a b c) '(1 2))    -> ((a . 1) (b . 2))
     (MAPCAR 'LIST '(a b c d e f) (CIRLIST 1 2))
     -> ((a 1) (b 2) (c 1) (d 2) (e 1) (f 2))
```

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

applique la fonction <fn> avec toutes les listes <li> comme arguments puis avec tous les CDR de ces listes jusqu'à ce qu'une de ces listes se termine. MAPCON est identique fonctionnellement à la fonction MAP. Toutefois chaque application doit retourner une liste en valeur et MAPCON retourne la liste des valeurs des applications qui sont rassemblées au moyen de la fonction NCONC (voir cette fonction). Si l'une de ces valeurs n'est pas une liste, elle est ignorée et ne figure pas dans la liste résultat.

MAPCON peut être défini en Lisp de la manière suivante :

```
(DE MAPCON (f . l)
    (WHEN (EVERY 'CONSP l)
          (NCONC (APPLY f l)
                 (APPLY 'MAPCON f (ALLCDR l)))))
```

```
ex : (MAPCON 'LIST '(1 2 3) '(4 5 6))
     -> ((1 2 3) (4 5 6) (2 3) (5 6) (3) (6))
     (MAPCON (LAMBDA (x) (LIST (CAR (LAST x))))
             '(a b c))   ->   (c c c)
```

; l'écriture suivante fait boucler la fonction LAST au troisième tour

```
(MAPCON 'LAST '(A B C))   -> ? ?
```

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

applique 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'à ce qu'une de ces listes se termine. MAPCAN est identique fonctionnellement à la fonction MAPC. Toutefois chaque application doit retourner une liste en valeur et MAPCAN retourne la liste des valeurs des applications qui sont rassemblées au moyen de la fonction NCONC (voir cette fonction). Si l'une de ces valeurs n'est pas une liste, elle est ignorée et ne figure pas dans la liste résultat.

MAPCAN peut être défini en Lisp de la manière suivante :

```
(DE MAPCAN (f . l)
    (WHEN (EVERY 'CONSP l)
          (NCONC (APPLY f (ALLCAR l))
                 (APPLY 'MAPCAN f (ALLCDR l)))))
```

```
ex : (MAPCAN (LAMBDA (x y) (list (1+ x) (1- y)))
             '(1 2 3)
             '(1 2 3))
     -> (2 0 3 1 4 2)
     (MAPCAN 'LIST '(a b c d)
                   (cirlist 1 2)
                   '(w x y z)
                   (cirlist 0))
     -> (a 1 w 0 b 2 x 0 c 1 y 0 d 2 z 0)
```

3.2.3 Autres fonctions d'application

(EVERY <fn> <l1> ... <ln>) [SUBR à N arguments]

applique la fonction <fn> avec tous les CAR de toutes les listes <li> comme arguments puis avec tous les CADR puis avec tous les CADDR ... tant que la valeur d'une application retourne une valeur différente de () ou bien qu'une de ces listes se termine. En ce cas EVERY retourne la valeur de la dernière application. Du fait de la construction des valeurs booléennes en Lisp, cette fonction est surtout utilisée pour appliquer un prédicat à tous les éléments d'une liste.

EVERY peut être défini en Lisp de la manière suivante :

```
(DE EVERY (f . l)
    ; cette description suppose que toutes les listes
    ; arguments sont de même taille.
    (IF (CAAR l)
        (IFN (CDAR l)
             (APPLY f (ALLCAR l))
             (AND (APPLY f (ALLCAR l))
                  (APPLY 'EVERY f (ALLCDR l))))
        T))
```

```
ex : (EVERY 'CONSP '((1) (2) (3)))     ->  (3)
     (EVERY 'EQ '(1 2 3) '(1 2 3))     ->  T
     (EVERY 'EQ '(1 2) '(1 2 3))       ->  ()
```

(ANY <fn> <l1> ... <ln>) [SUBR à N arguments]

applique 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'à ce que la valeur d'une application retourne une valeur différente de () ou bien qu'une de ces listes se termine. ANY retourne la valeur de la première application différente de () ou bien () si l'une des listes argument se termine. Du fait de la construction des valeurs booléennes en Lisp, cette fonction est surtout utilisée pour appliquer un prédicat à tous les éléments d'une liste.

ANY peut être défini en Lisp de la manière suivante :

```
(DE ANY (f . l)
    ; cette description suppose que toutes les listes
    ; arguments sont de même taille.
    (WHEN (CAR l)
          (OR (APPLY f (ALLCAR l))
              (APPLY 'ANY f (ALLCDR l)))))
```

```
ex : (ANY 'CONSP '(1 "Foo" (1) 10))    ->   (1)
     (ANY '= '(1 2 3) '(10 2 30))      ->   2
```

(MAPVECTOR <fn> <vect>) [SUBR à 2 arguments]

applique la fonction <fn>, qui doit être une fonction à un argument, à chaque élément du vecteur <vect>, en commençant par l'élément d'indice 0. MAPVECTOR retourne () en valeur.

MAPVECTOR peut être défini en Lisp de la manière suivante :

```
(DE MAPVECTOR (fn vect)
    (FOR (i 0 1 (1- (VLENGTH vect)))
         (FUNCALL fn (VREF vect i))))
```

```
ex : (MAPVECTOR 'PRIN #[a b c]) abc -> ()
```

(MAPOBLIST <fn>) [SUBR à 1 argument]

applique successivement la fonction <fn>, qui doit être à 1 argument, à 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 parce qu'elle ne construit pas la liste complète des symboles, liste qui possède typiquement plus de 2000 éléments.

```
ex : fonction qui imprime toutes les NSUBR du système
```

```
(DE PRINTNSUBR ()
    (MAPOBLIST (LAMBDA (symb)
                       (WHEN (EQ (TYPEFN symb) 'NSUBR)
                             (PRINT symb)))))
```

(MAPCOBLIST <fn>) [SUBR à 1 argument]

applique successivement la fonction <fn>, qui doit être à 1 argument, à tous les éléments de la liste des symboles *(l'OBLIST)*. Chaque évaluation doit retourner une liste et l'ensemble de ces listes est rassemblé au moyen de la fonction NCONC. Si l'une de ces valeurs n'est pas une liste, elle est ignorée et ne figure pas dans la liste résultat. Cette fonction est donc équivalente à :

(MAPCAN <fn> (OBLIST))

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

```
ex : fonction qui retourne, une liste de type
          (symb OBJVAL symb OBJVAL ...)
     pour tous les symboles qui en possèdent une.

     (DE FINDOVAL ()
         (MAPCOBLIST (LAMBDA (symb)
                             (WHEN (OBJVAL symb)
                                   (LIST symb
                                         (OBJVAL symb))))))
```

(MAPLOBLIST <fn>) [SUBR à 1 argument]

applique successivement la fonction <fn>, qui doit être à 1 argument, à tous les éléments de la liste des symboles *(l'OBLIST)*. Chaque évaluation doit retourner une valeur booléenne et MAPLOBLIST retourne la liste des symboles pour lesquels cette valeur est vraie.

MAPLOBLIST peut être défini en Lisp de la manière suivante :

```
    (DE MAPLOBLIST (fn)
        (MAPCOBLIST
            (LAMBDA (s)
                    (WHEN (FUNCALL fn s) (LIST s)))))

ex : pour retourner le nombre de fonctions du système évaluez :

     (LENGTH (MAPLOBLIST 'TYPEFN))

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

     (DE FINDFSUBR ()
         (MAPLOBLIST (LAMBDA (symb)
                             (EQ (TYPEFN symb) 'FSUBR)))
```

3.3 Les Fonctions Manipulant l'Environnement

Ces fonctions vont pouvoir changer l'environnement dynamique temporairement. Le terme environnement est pris ici comme l'ensemble des liaisons variable-valeur. Au sortir de ces 5 fonctions l'environnement initial (c'est-à-dire 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 (c'est-à-dire une lambda-expression).
<l> est une liste dont les éléments ont la forme :

- <var> ou bien
- (<var> <val>)

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

La fonction LET va lier dynamiquement et en parallèle toutes les variables <vari> aux valeurs <vali> si elles sont fournies ou bien avec () par défaut, 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 ainsi de définir et d'initialiser des variables locales très facilement.

LET est une forme abrégée d'appel de fonctions anonymes de type EXPR et peut donc provoquer les mêmes erreurs : ERRBPA, ERRILB ou ERRBAL.

La forme
```
(LET ((<var> <val>)) <s1> ... <sN>)
```
correspond à l'appel
```
((LAMBDA (<var>) <s1> ... <sN>) <val>)
```
et la forme
```
(LET ((<var1> <val1>) ... (<varN> <valN>))
     <s1> ... <sN>)
```
correspond à l'appel
```
((LAMBDA (<var1> ... <varN>) <s1> ... <sN>)
 <val1> ... <valN>)
```

LET peut être défini en Lisp de la manière suivante :

```
(DMD LET (lv . body)
     (COND ((NULL lv)
            (CONS (CONS 'LAMBDA (CONS () body))))
           (T (CONS (CONS 'LAMBDA
                          (CONS (MAPCAR
                                  (LAMBDA (l)
                                      (IF (CONSP l)
                                          (CAR l)
                                          l))
                                  lv)
                                body))
                    (MAPCAR (LAMBDA (l)
                                (IF (CONSP l)
                                    (CADR l)
                                    ()))
                            lv)))))
```

; ou d'une manière plus lisible (enfin pour certains) :

```
(DMD LET (varl . body)
     (IF (NULL varl)
         `((LAMBDA () ,@body))
         `((LAMBDA ,(MAPCAR (LAMBDA (l) (IF (CONSP l) (CAR l) l))
                            `,varl)
                   ,@body)
           ,@(MAPCAR (LAMBDA (l) (IF (CONSP l) (CADR l) ()))
                     `,varl))))
ex : (LET ((i 10) (j 20)) (+ i j))      -> 30
     (LET ((a 'foo) ((b . c) (cons 1 2))) (list a b c))
     -> (foo 1 2)
     (LET ((i 10) j k) (LIST i j k))  -> (10 () ())
```

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

est identique à la fonction LET mais va permettre de calculer le nom des variables qui vont être liées. <lvar> est un argument qui, une fois évalué, doit livrer un arbre de symboles; <lval> est un argument qui, une fois évalué, doit livrer une liste de valeurs. LETV va lier les symboles de l'arbre <lvar> avec les valeurs de la liste <lval> puis, dans ce nouvel environnement, évaluer les différentes expressions <s1> ... <sN>. LETV retourne en valeur la valeur de la dernière évaluation c'est-à-dire celle de <sN> puis restaure l'environnement dans l'état précédant l'appel. LETV permet donc de lier des variables dont le nom est calculé. LETV peut provoquer les mêmes erreurs (ERRBPA, ERRILB, ERRWNA ou ERRBAL) que l'évaluation d'une fonction anonyme.

LETV peut être défini en Lisp de la manière suivante :

```
(DMD LETV (lvar lval . body)
     `((LAMBDA (,(EVAL lvar)) ,@body) ,lval))
ex : (LETV '(a (b . c) d) '(1 (2 3) 4) (LIST a b c d))
     -> (1 2 (3) 4)
```

(LETVQ <lvar> <lval> <s1> ... <sN>) [FSUBR]

cette fonction est identique à la fonction précédente toutefois le premier argument (l'arbre des variables <lvar>) n'est pas évalué.

LETVQ peut être défini en Lisp de la manière suivante :

```
(DMD LETVQ (lvar lval . body)
     `((LAMBDA (,lvar) ,@body) ,lval))
ex : (LETVQ (a . b) '(1 2) (LIST a b))    -> (1 (2))
```

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

ces fonctions sont identiques 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 LETS lui-même. Il n'y a aucune différence entre les fonctions LETS, SLET et LET*.

Ainsi la forme :

```
(LETS ((<var1> <val1>) (<var2> <val2>) ... (<varN> <valN>))
      <s1> ... <sN>)
```

correspond à :

```
(LET ((var1> <val1>))
     (LET ((var2> <val2>))
          .....
          (LET ((varN> <valN>))
               <s1> ... <sN>) ... ))
```

et également à :

```
((LAMBDA (<var1>)
         ((LAMBDA (<var2>)
                  ........
                  ((LAMBDA (<varN>)
                       <s1> ... <sN>)
                   <valN>) ...)
          <val2>)
   <val1>)
```

```
ex : (let ((i 10)) (slet ((i 20) (j (+ i i))) (list i j)))
     -> (20 40)
```

(LETN <symb> <l> <s1> ... <sN>) [FSUBR]

permet d'appeler une fonction nommée de type EXPR.

<symb> est le nom temporaire de la fonction
<l> est une liste de couples variable-valeur comme dans la fonction LET
<s1> ... <sN> est un corps de fonction.

La fonction LETN 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 <symb> la même lambda-expression que dans la fonction LET. LETN permet donc la définition dynamique et l'appel d'une fonction récursive.

La forme :
(LETN <symb> ((<var1> <val1>) ... (<varN> <valN>))
<s1> .. <sN>)
est équivalente à :
(FLET ((<symb> (<var1> ... <varN>) <s1> ... <sN>))
(<symb> <val1> ... <valN>))

LETN peut être défini en Lisp de la manière suivante :

```
(DMD LETN (name lvar . body)
     `(FLET ((,name ,(MAPCAR 'CAR `,lvar) ,@body))
            (,name ,@(MAPCAR 'CADR `,lvar)))))))))
```

La fonction INDEX qui retourne l'indice de la sous-chaîne pname1 de la chaîne pname2 à partir de l'indice n

```
ex : (DE INDEX (pname1 pname2 n)
         (LETN INXAUX1 ((p1 (PNAME pname1))
                        (p2 (NTHCDR n (PNAME pname2)))
```

```
          (n n))
    (COND ((< (LENGTH p2) (LENGTH p1)) ())
          ((<> (CAR p1) (CAR p2))
           (INXAUX1 p1 (CDR p2) (1+ n)))
          (T (LETN INXAUX2 ((pp1 (CDR p1))
                            (pp2 (CDR p2)))
                   (COND ((NULL pp1) n)
                         ((= (CAR pp1) (CAR pp2))
                          (INXAUX2 (CDR pp1)
                                   (CDR pp2)))
                         (T (INXAUX1 p1 (CDR p2)
                                     (1+ n)))))))))
```

3.4 Les Fonctions de Définition de Fonctions

Ces fonctions vont permettre de définir de nouvelles fonctions. Toutes testent la validité de leurs arguments :
- les noms des fonctions doivent être des symboles
- toutes les variables de la liste ou de l'arbre des paramètres doivent également être des symboles.

Si ce n'est pas le cas, l'erreur ERRBDF ou ERRBPA apparaît dont les libellés sont par défaut :

```
** <fn> : mauvaise définition : <symb>      ou bien
** <fn> : bad definition : <symb>

** <fn> : mauvais paramètre : <e>      ou bien
** <fn> : bad parameter : <e>
```

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).

3.4.1 Les définitions des fonctions statiques

Toutes ces fonctions vont changer de façon permanente les fonctions associées aux symboles.

#:SYSTEM:PREVIOUS-DEF-FLAG [Variable]

Toute redéfinition de fonction peut être 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 #:SYSTEM:PREVIOUS-DEF. Ce sauvetage est contrôlé par la variable #:SYSTEM:PREVIOUS-DEF-FLAG : si elle est vraie (c'est-à-dire si sa valeur est différente de ()) la sauvegarde a lieu, si elle est fausse aucune sauvegarde n'est effectuée. Par défaut cette variable vaut ().

#:SYSTEM:REDEF-FLAG [Variable]

De plus l'indicateur #:SYSTEM:REDEF-FLAG permet d'être averti ou non en cas de redéfinition. Si #:SYSTEM:REDEF-FLAG est faux (égal à ()), ce qui est l'option par défaut, un message d'avertissement est imprimé en cas de tentative de redéfinition d'une fonction. Si cet indicateur est vrai (différent de ()) les redéfinitions ne provoquent pas d'impression de message. Le message a la forme :

```
** <fn> : fonction redéfinie : <symb>      ou bien
** <fn> : redefined function : <symb>
```

dans lequel <fn> est la fonction de définition qui a été appelée et <symb> le nom de la fonction qui vient d'être redéfinie.

Toutes les fonctions de définition ajoutent également sur la P-liste du symbole, sous l'indicateur #:SYSTEM:LOADED-FROM-FILE, le nom du fichier à partir duquel la fonction a été chargée. Ce nom doit se trouver dans la variable #:SYSTEM:LOADED-FROM-FILE. Si la valeur de cette variable est égale à (), rien n'est rajouté sur la P-liste du nom de la fonction.

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

permet de définir statiquement de nouvelles EXPR. <symb> 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 <symb> en valeur.

```
; aux tests de validité près, la forme
(DE <symb> <lvar> <s1> ... <sN>)
; est équivalente à
(SETFN '<symb> 'EXPR '(<lvar> <s1> ... <sN>)

ex : (DE FOO (x) (IF (NULL (CDR x)) x (FOO (CDR x))))   -> FOO
     (VALFN 'FOO))
     -> ((x) (IF (NULL (CDR x)) x (FOO (CDR x))))
     (TYPEFN 'FOO)     -> EXPR
     (FOO '(A B C))    -> (C)
```

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

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

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

DF retourne en valeur le nom <symb> de la fonction définie.

```
; aux tests de validité près, la forme
(DF <symb> <lvar> <s1> ... <sN>)
; est équivalente à
(SETFN '<symb> 'FEXPR '(<lvar> <s1> ... <sN>)

ex : (DF INCR (var . val)
         (SET var (IF (CONSP val)
                      (+ (EVAL var) (EVAL (CAR val)))
                      (1+ (EVAL var)))))          -> INCR
```

```
(TYPEFN 'INCR)                          -> FEXPR
(SETQ nb 7)                             -> 7
(INCR nb)                               -> 8
nb                                      -> 8
```

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

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

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

DM retourne en valeur le nom <symb> de la fonction définie.

```
; aux tests de validité près, la forme
(DM <symb> <lvar> <s1> ... <sN>)
; est équivalente à
(SETFN '<symb> 'MACRO '(<lvar> <s1> ... <sN>)
```

```
ex : (DM DECR (decr var . val)
         (LIST 'SETQ var
               (IF val
                   (LIST '- var (CAR val))
                   (LIST '1- var))))
     -> DECR
(SETQ N 10)  -> 10
(DECR N)     -> 9
```

(DMD <symb> <lvar> <s1> ... <sN>) [FSUBR]
(DEFMACRO <symb> <lvar> <s1> ... <sN>) [FSUBR]

permet de définir statiquement une nouvelle DMACRO de nom <symb> qui accepte un arbre de variables <lvar>. La liaison s'effectuera entre l'arbre des variables <lvar> et le CDR de l'appel de cette DMACRO. DMD retourne <symb> en valeur. Il n'y a aucune différence entre les fonctions DMD et DEFMACRO.

DMD peut être défini en Lisp de la manière suivante :

```
(DF DMD (nom larg . corps)
    (EVAL `(DM ,nom -l-
               (LETVQ ,larg (CDR -l-)
                      (DISPLACE -l-
                           ,(IF (AND (CONSP corps)
                                     (CONSP (CAR corps))
                                     (NULL (CDR corps)))
                                (CAR corps)
                                (CONS 'PROGN corps)))
                      -l-))))
```

```
; aux tests de validité près, la forme
(DMD <symb> <lvar> <s1> ... <sN>)
; est équivalente à
(SETFN '<symb> 'DMACRO '(<lvar> <s1> ... <sN>)
```

```
ex : (DMD IF (test then else1 . elseN)
          `(COND ((,test ,then) (T ,else1 ,@elseN))))
```

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

permet de définir statiquement une fonction de type SUBR ou FSUBR. <symb> est le nom d'un symbole auquel sera rattachée une fonction écrite en langage machine qui débute à l'adresse <adr>. Le type est fourni par l'argument <type>. Cette fonction, peu utilisée mais présente par souci d'homogénéité, est une forme FSUBRée de SETFN.

```
; aux tests de validité près, la forme
(DS <symb> <type> <adr>)
; est équivalente à
(SETFN '<symb> '<type> '<adr>)
```

3.4.2 L'utilisation avancée des MACRO

L'écriture de MACRO est un véritable sport en soi. Le_Lisp encourage leur utilisation au détriment des FEXPR qui provoquent des captures de variables. Pour faciliter l'écriture des MACRO il existe le macro-caractère accent grave (backquote) qui est décrit dans le chapitre sur les entrées/sorties et les 2 fonctions suivantes qui permettent de contrôler le résultat des expansions de ces MACRO.

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

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

MACROEXPAND1 peut être défini en Lisp de la manière suivante :

```
(DE MACROEXPAND1 (x)
    ; expanse une macro à un seul niveau
    (COND ((EQ (TYPEFN (CAR x)) 'MACRO)
               (APPLY (CONS 'LAMBDA (VALFN (CAR x))) x))
          ((EQ (TYPEFN (CAR x) 'DMACRO)
               (APPLY (CONS 'LAMBDA (VALFN (CAR x))) (CDR x)))
          ((EQ (TYPEFN (CAR x)) 'MSUBR)
               (CALL (VALFN (CAR x)) x () ()))
          ((EQ (TYPEFN (CAR x)) 'DMSUBR)
               (CALL (VALFN (CAR x)) (CDR x) () ()))
          (T x))))
ex : (DMD FOO (x1 x2) `(MCONS ,x1 FOO ,x2))   -> FOO
     (MACROEXPAND1 '(FOO 10 20))          -> (MCONS 10 FOO 20)
```

(MACROEXPAND <s>) [SUBR à 1 argument]

permet d'expanser dans l'expression <s> les appels de MACRO, de DMACRO, de MSUBR et de DMSUBR à tous les niveaux.

MACROEXPAND peut être défini en Lisp de la manière suivante :

```
(DE MACROEXPAND (x)
    ; expanse un appel tant qu'il s'agit d'une macro
    (COND ((ATOM x) x)
          ((SYMBOLP (CAR x))
             (COND ((EQ (TYPEFN (CAR x)) 'MACRO)
```

```
                        (MACROEXPAND
                            (APPLY (CONS 'LAMBDA (VALFN (CAR x)))
                                   x)))
                       ((EQ (TYPEFN (CAR x)) 'DMACRO)
                        (MACROEXPAND
                           (APPLY (CONS 'LAMBDA (VALFN (CAR x)))
                                  (CDR x))))
                       ((EQ (TYPEFN (CAR x)) 'MSUBR)
                        (MACROEXPAND
                           (CALL (VALFN (CAR x)) x () ())))
                       ((EQ (TYPEFN (CAR x)) 'DMSUBR)
                        (MACROEXPAND
                           (CALL (VALFN (CAR x)) (CDR x) () ())))
                       ((EQ (CAR x) 'QUOTE) x)
                       (T (LET ((s ()))
                             (WHILE (CONSP x)
                                 (NEWL s (MACROEXPAND (NEXTL x))))
                             (PROG1 (NREVERSE s) (RPLACD s x))))))
          (T (LET ((s ()))
                  (WHILE (CONSP x)
                         (NEWL s (MACROEXPAND (NEXTL x))))
                  (PROG1 (NREVERSE s) (RPLACD s x))))))
ex : (DMD FOO (x1 x2) `(MCONS ,x1 FOO ,x2))        -> FOO
     (DM BAR (bar x) `(FOO ,x 30))                 -> BAR
     (MACROEXPAND '(BAR (FOO 10 20)))
                  ->   (MCONS (MCONS 10 FOO 20) FOO 30)
```

3.4.3 La 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 posséder 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)
                                    (KWOTE (EVAL val)))
                           lvarclot))
          (lvar (CADR fnt))
          (corps (CDDR fnt)))
         `(LAMBDA ,lvar
             ((LAMBDA ,lvarclot
                  (PROTECT (PROGN ,@corps)
                           ,@(MAPCAR
                               '(LAMBDA (slot var)
                                   `(RPLACA
                                      (CDR ,`(QUOTE ,slot))
                                      ,var))
                               listval  lvarclot)))
              ,@listval)))))
```

```
ex : ; un générateur d'entier naturel
     (setfn 'nextint  'expr
            (let ((n 1))
                 (cdr (closure '(n)
                               '(lambda () (setq n (1+ n)))))))

ex : ; un générateur des nombres de la suite de Fibonacci
     (setfn 'fib 'expr
               (let ((x 1) (y 1))
                    (cdr (closure '(x y)
                                  '(lambda ()
                                       (setq y (prog1 (+ x y)
                                                      (setq x y))))))))

     (nextint)     -> 2
     (nextint)     -> 3
     n             ->
                        ** EVAL : variable indéfinie : n
     (fib)         -> 2
     (fib)         -> 3
     (fib)         -> 5
```

3.4.4 Les 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 listes. Chaque sous-liste à la forme :
(<symb> <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 chacun des symboles <symb>. Ces fonctions, qui sont toujours de type EXPR, possèdent 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 c'est-à-dire celle de <sN>. Au sortir du FLET, les symboles <symb> reprendront leur ancienne définition (s'ils en avaient une).

Du fait de l'introduction des interruptions programmables, FLET n'a plus grand emploi. De plus son utilisation introduit de très mauvaises performances dans le code compilé à cause de l'aspect totalement dynamique de la redéfinition, et est donc à éviter.

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

3.5 Les Variables-Fonctions

On appelle *variable-fonction* une fonction à nombre variable d'arguments qui permet d'accéder en lecture et en écriture (get/set) à une donnée Lisp. L'accès en lecture est réalisé par l'appel de cette fonction avec N-1 arguments et l'accès en écriture est réalisé par l'appel de cette fonction avec N arguments, le dernier étant la nouvelle valeur de la variable fonction. Le type de la donnée Lisp manipulée par ces variables-fonctions peut être :
- interne au système Le_Lisp (comme par exemple la base de conversion en sortie des nombres, les tampons d'entrée ou de sortie ...)
- propre à un programme spécifique, en ce cas n'importe quel objet Lisp peut être utilisé.

Une variable-fonction possède toutes les propriétés des fonctions (car c'en est une) mais peut également être *liée dynamiquement* à la manière des variables Lisp au moyen de la fonction suivante.

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

<l> est une liste de la forme :
((<symb1> <arg11> ... <arg1N>) ... (<symbN> <argN1> ... <argNN>))
dans laquelle les <symbi> sont les noms des différentes variables-fonctions et <argi1> ... <argiN> les différents arguments de ces fonctions accédant aux données en écriture. WITH va réaliser ces écritures après avoir sauvé les valeurs de ces variables-fonctions appelées en lecture puis évaluer les différentes expressions <s1> ... <sN> et retourner en valeur la valeur de <sN>. Mais, à la manière d'une liaison dynamique, WITH va rappeler les variables fonctions de nouveau en écriture avec les valeurs de ces variables fonctions précédemment sauvées. Ces restitutions de valeurs sont protégées à la manière de la fonction PROTECT.

WITH peut être défini en Lisp de la manière suivante :

```
(DMD WITH (l . body)
     (LET ((var (MAPCAR (LAMBDA (x) (GENSYM)) l)))
          `(LET (,@(MAPCAR
                     (LAMBDA (var l)
                        (LIST var
                              (FIRSTN (1- (LENGTH l))
                                      l)))
                     var
                     l))
             (PROTECT
               (PROGN ,@l ,@body)
               ,@(MAPCAR
                    (LAMBDA (var l)
                       (APPEND1 (FIRSTN (1- (LENGTH l)) l)
                                var))
                    var
                    l)))))
```

```
ex : ; OBASE est la variable fonction système d'accès à
     ; la base de numération des nombres en sortie
```

```
(OBASE)                              -> 10
(PRIN  100)                          -> 100
(WITH ((OBASE 16)) (PRIN 100))       -> 64
(OBASE)                              -> 10

; (WITH ((OBASE 16)) (PRIN 100))  équivaut à :
; (LET ((xxx (OBASE)))
;      (PROTECT (PROGN (OBASE 16)
;                      (PRIN 100))
;               (OBASE xxx)))
```

```
ex : (DE MONINDIC (pl i . n)
       (IF (CONSP n)
           (PLIST pl 'indic (CAR n))
           (PLIST pl 'indic)))

(MONINDIC 'x '(a))                          -> (a)
(MONINDIC 'x)                               -> (a)
(WITH ((MONINDIC 'x 20)) (MONINDIC 'x))     -> 20
(MONINDIC 'x)                               -> (a)
```

3.6 Les Fonctions de Contrôle de Base

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 c'est-à-dire 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.

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

si la valeur de l'évaluation de <s1> est différente de (), 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 () 1 2 3)  -> 3

     ; voici la fonction d'ACKERMANN décrite avec des IF

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

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

si la valeur de l'évaluation de <s1> est égale à (), 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 (test then . else)
    (IF (NULL (EVAL test))
        (EVAL then)
        (EPROGN else)))
```

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

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

si la valeur de l'évaluation de <s1> est différente de (), 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 (), WHEN retourne (). WHEN permet de construire une structure de contrôle de type :

si <s1> alors <s2> ... <sN> sinon ()

WHEN peut être défini en Lisp de la manière suivante :

```
(DF WHEN (test . body)
    (IF (EVAL test)
        (EPROGN body)
        ()))
```

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

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

si la valeur de l'évaluation de <s1> est égale à (), 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 (), UNLESS retourne (). UNLESS permet de construire une structure de contrôle de type :

si <s1> alors () sinon <s2> ... <sN>

UNLESS peut être défini en Lisp de la manière suivante :

```
(DF UNLESS (test . body)
    (IF (EVAL test)
        ()
        (EPROGN body)))
```

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

(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 (). OR retourne cette valeur. Si aucune expression n'est fournie, cette fonction retourne ().

OR peut être défini en Lisp de la manière suivante :

```
(DF OR 1
    (LETN OREVAL ((l l))
          (IF (NULL (CDR l))
              (EVAL (CAR l))
              (LET ((resul (EVAL (CAR l))))
                   (IF resul
                       resul
                       (OREVAL (CDR l)))))))
```

```
ex : (OR)              -> ()
     (OR ())           -> ()
     (OR 1 2)          -> 1
     (OR () () 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 (), à ce moment AND retourne () 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
        (LETN ANDEVAL ((l l))
              (IF (NULL (CDR l))
                  (EVAL (CAR l))
                  (IF (EVAL (CAR l))
                      (ANDEVAL (CDR l))
                      ())))))
```

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

(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>)

<ss> est une expression quelconque et <s1> ... <sN> le corps de la clause. COND va sélectionner et évaluer le corps de la première clause dont la valeur de l'évaluation de son premier élément <ss> est différente de (). COND évalue

alors en séquence les différentes expressions <s1> ... <sN> de la clause sélectionnée et retourne la valeur de la dernière évaluation <sN>. Si la clause sélectionnée n'a qu'un élément <ss>, COND retourne la valeur de l'évaluation de <ss> (c'est-à-dire 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 sélectionnée, COND retourne (). Pour forcer la sélection de la dernière clause, il est d'usage d'utiliser comme sélecteur la constante symbolique T dont la valeur est toujours différente de () (voir la section sur les prédicats).

COND peut être défini en Lisp de la manière suivante :

```
(DF COND 1
    (LETN CONDEVAL ((l 1))
          (WHEN l
                (IF (OR (ATOM l) (ATOM (CAR l)))
                    (ERROR 'COND 'ERRSXT l)
                    (LET ((select (EVAL (CAAR l))))
                         (IF select
                             (IF (CDAR l)
                                 (EPROGN (CDAR l))
                                 select)
                             (CONDEVAL (CDR l)))))))))
```

; donc : est équivalent à :

```
(COND                          (COND
  (p1 e11 e12 e13)               (p1 (PROGN e11 e12 e13))
  (p2 e21 e22)                   (p2 (PROGN e21 e22))
  (p3)                           ((LET ((aux p3)) aux))
  (p4 e41))                      (p4 e41))
```

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

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

comme pour la fonction COND, SELECTQ va sélectionner 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 atome (en utilisant le prédicat EQUAL).
- les différents éléments du CAR (non évalué) de la clause si celui-ci est une liste. La comparaison utilise la fonction de recherche MEMBER. Cette dernière possibilité permet donc de sélectionner des objets en nombre quelconque.

Dès qu'une clause est sélectionnée, 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 sélectionnée, SELECTQ retourne ().
SELECTQ permet donc de construire des aiguillages sur des valeurs constantes.
Comme pour la fonction COND, il est possible de forcer la sélection de la dernière clause en utilisant la constante symbolique T en première position d'une clause.

SELECTQ peut être défini en Lisp de la manière suivante :

```
(DF SELECTQ (sel . cl)
    (LETN SEL1 ((sel (EVAL sel)) (cl cl))
       (COND
          ((NULL cl)
             ; plus de clauses
             ())
          ((OR (ATOM cl) (ATOM (CAR cl)))
             ; mauvaise clause
             (ERROR 'SELECTQ 'ERRSXT cl))
          ((EQ (CAAR cl) T)
             ; sélection de la clause toujours vraie
             (EPROGN (CDAR cl)))
          ((AND (ATOMP (CAAR cl)) (EQUAL sel (CAAR   cl)))
             ; sélection simple
             (EPROGN (CDAR cl)))
         ((MEMBER sel (CAAR  cl))
             ; sélection générale
             (EPROGN (CDAR cl)))
         (T (SEL1 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
```

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

tant que la valeur de l'évaluation de <s> est différente de (), WHILE va évaluer en séquence les différentes expressions <s1> ... <sN>. WHILE retourne toujours () 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 la constante symbolique T possède toujours une valeur différente de ().

WHILE peut être défini en Lisp de la manière suivante :

```
(DF WHILE (test . corps)
    (LETN WHILE1 ()
          (IF (NULL (EVAL test))
              ()
              (EPROGN corps)
              (WHILE1))))
```

; WHILE est équivalent à la forme Lisp :

```
(LETN WHILE ()
     (IFN <s> () <s1> ... <sN> (WHILE)))
```

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

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

tant que la valeur de l'évaluation de <s> est égale à (), UNTIL va évaluer en séquence les différentes expressions <s1> ... <sN>. UNTIL retourne la valeur de la 1ère évaluation de <s> différente de () (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 (test . corps)
    (LETN UNTIL1 ()
          (OR (EVAL test)
              (PROGN (EPROGN corps) (UNTIL1)))))
```

; UNTIL est équivalent (à la valeur retournée près) à la forme Lisp :

```
(WHILE (NOT <s>) <s1> ... <sN>)
```

```
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 entier <n>. REPEAT évalue alors <n> fois les différentes expressions <s1> ... <sN>. Si <n> n'est pas un nombre, l'erreur ERRNIA se déclenche. Si la valeur de <n> n'est pas un nombre strictement positif, la boucle n'est pas exécutée. Dans tous les cas REPEAT retourne la valeur T.

REPEAT peut être défini en Lisp de la manière suivante :

```
(DF REPEAT (n . l)
    (LET ((n (EVAL n)))
         (IF (FIXP n)
             (WHILE (>= n 0) (EPROGN l) (DECR n))
             (ERROR 'REPEAT 'ERRNIA n)))
    T)
```

```
ex : (REPEAT 10 (PRIN '*)) **********    -> T
     (REPEAT 0 (PRIN '*))  -> T
```

(FOR (<var> <in> <ic> <ntl> <e1> ... <eN>) <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, <in> est la valeur numérique initiale de cette variable en entrant dans la boucle, <ic> est l'incrément numérique à chaque passage dans la boucle et <ntl> est la valeur maximale (ou minimale dépendant du signe de <ic>) que peut prendre la variable de contrôle. FOR retourne la valeur de (PROGN <e1> ... <eN>).

FOR peut être défini en Lisp de la manière suivante :

```
(DMD FOR ((var init step end . res) . body)
     (UNLESS (VARIABLEP var)
          (ERROR 'FOR 'ERRNVA var))
     (UNLESS (AND init step end)
          (ERROR 'FOR 'ERRSXT (LIST var init step end)))
     (LET ((endvar (CONCAT 'FOR (GENSYM)))
           (test (IF (NUMBERP step)
                     (IF (> step 0) '<= '>=)
                     `(IF (> ,step 0) '<=
                          (IF (< ,step 0) '>=
                              (ERROR 'FOR "increment nul" 0))))))
          `(LET ((,var ,init)
                 (,endvar ,end))
                (WHILE (,test ,var ,endvar)
                       ,@body
                       (INCR ,var ,step))
                ,@res))))
ex : (FOR (i 0 1 9 'ok) (PRIN i))   0123456789 -> ok
```

3.7 Les Fonctions de Contrôle Lexical

Le_Lisp v15.2 introduit la notion de contrôle lexical ou textuel. Les structures de contrôle décrites dans cette section ont une portée purement lexicale. Une utilisation incorrecte peut provoquer une des deux erreurs ERRNAB ou ERRXIA dont les libellés par défaut sont :

```
** <fn> : pas de portée lexicale : <e>     ou bien
** <fn> : no lexical scope : <e>

** <fn> : bloc lexical périmé : <e>     ou bien
** <fn> : inactive lexical scope : <e>
```

3.7.1 Les fonctions de contrôle lexical primitives

(BLOCK <symb> <e1> ... <eN>) [FSUBR]

<symb> est le nom symbolique, non évalué d'un bloc lexical qui évalue les différentes expressions <e1> ... <eN>. Par défaut BLOCK retourne la valeur de la dernière évaluation, c'est-à-dire celle de <eN>. Toutefois ces évaluations séquencielles peuvent être interrompues par la fonction suivante.

(RETURN-FROM <symb> <e>) [FSUBR]

<symb> doit être le nom d'un bloc lexical de type BLOCK et n'est pas évalué. RETURN-FROM sort du bloc lexical nommé en retournant la valeur de l'évaluation de <e>. Si cette fonction apparaît en dehors de la portée lexicale du bloc en question, l'erreur ERRNAB se déclenche.

(RETURN ‹s›) [FSUBR]

cette fonction correspond à l'appel :
(RETURN-FROM () ‹s›)

(TAGBODY ‹ec1› ... ‹ecN›) [FSUBR]

‹ec1› ... ‹ecN› sont soit des symboles considérés comme des étiquettes soit des listes qui sont des formes à évaluer. TAGBODY va évaluer séquentiellement les différentes formes et retourner () par défaut. L'évaluation séquentielle de ces formes peut être interrompue par la fonction suivante.

(GO ‹symb›) [FSUBR]

‹symb› est le nom d'une étiquette du TAGBODY courant et permet de reprendre l'évaluation séquentielle à partir de celle-ci. Si ‹symb› n'est pas une étiquette d'un TAGBODY lexical, l'erreur ERRNAB se déclenche.

exemples d'utilisation des fonctions de contrôle lexicales

```
ex : (BLOCK GAL 1 (RETURN-FROM GAL 2) 3)            -> 2
     (BLOCK () 1 (BLOCK AMANT 2 (RETURN 3) 4) 5)    -> 3
     (BLOCK DELA (EVAL '(RETURN-FROM DELA 10)))
                 ** RETURN-FROM : pas de portée lexicale : DELA
     (BLOCK REINE (LET ((y #'(LAMBDA () (RETURN-FROM REINE 10))))
                       (FUNCALL y)))                 -> 10
     (LET (y) (BLOCK ALLA
                     (SETQ y #'(LAMBDA () (RETURM-FROM ALLA 10))))
              (FUNCALL y))
                  ** RETURN-FROM : bloc lexical périmé : ALLA

     (LET ((n 5) 1)
          (TAGBODY
             tour  (IF (<= n 0)
                       (GO magne)
                       (NEWL 1 (DECR n))
                       (GO tour))
             magne)
          1)                                    -> (0 1 2 3 4)
```

Le cas tordu de l'aluminium book [Steele 84]

```
(DE CEX (f g x)
    (if (= x 0)
        (FUNCALL f)
        (BLOCK here
               (+ 5 (CEX g
                         #'(LAMBDA () (RETURN-FROM here 4))
                         (- x 1)))))))

(CEX () () 2)      doit retourner (et retourne) 4
```

3.7.2 Les fonctions d'itération de type PROG

(PROG <l> <ec1> ... <ecN>) [MACRO]

permet de combiner les fonctions BLOCK, LET et TAGBODY.

PROG peut être défini en Lisp de la manière suivante :

```
(DMD PROG (l . corps)
     `(BLOCK ()
             (LET ,l
                  (TAGBODY ,@corps))))
```

(PROG* <l> <ec1> ... <ecN>) [MACRO]

est équivalente à la fonction précédente, toutefois les liaisons des variables ont lieu en séquence (et non en parallèle) au moyen de la forme LET*.

PROG* peut être défini en Lisp de la manière suivante :

```
(DMD PROG* (l . corps)
     `(BLOCK ()
             (LET* ,l
                   (TAGBODY ,@corps))))
```

3.7.3 Les fonctions d'itération de type DO

(DO <lv> <lr> <ec1> ... <ecN>) [MACRO]

cette macro, empruntée à Maclisp, est une fonction très générale d'itération à une ou plusieurs variables de contrôle. Elle est également une combinaison des fonctions LET, BLOCK et TAGBODY augmentée d'une structure d'itération.

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

  (<symb> <val> <inc>)

<symb> est le nom de la variable de contrôle
<val> est l'expression de sa valeur initiale (avant d'entrer dans la boucle) par défaut cette valeur est ().
<inc> est l'expression de son incrément 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 (), 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.

- <ec1> ... <ecN> est le corps du DO, analogue à la fonction TAGBODY.

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 (c'est-à-dire en parallèle)

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

3) si le test a retourné (), le corps du DO ‹ec1› ... ‹ecN›
est exécuté à la manière d'un PROG.

4) arrivé à la fin du corps du DO, les différentes expressions ‹inc1› ... ‹incN› des triplets sont évaluées et sont utilisées (si elles existent) pour lier les variables de contrôle ‹sym1› ... ‹symN›. Puis le processus reprend en 2).

DO peut être défini en Lisp de la manière suivante :

Cette définition n'optimise pas les BLOCK et les TAGBODY vides

```
(DMD DO (lv (test . resul) . corps)
     `(BLOCK ()
            (LET ,(MAPCAR (LAMBDA (x) (LIST (CAR x) (CADR x)))
                          lv)
                 (UNTIL ,test
                     (TAGBODY ,@corps)
                     (PSETQ ,@(MAPCAN (LAMBDA (x)
                                        (WHEN (CONSP (CDDR x))
                                            (LIST (CAR x) (CADDR x))))
                                      lv)))
                 ,@resul)))
```

Une manière d'écrire la fonction LENGTH

```
(DE LNGT (l)
    (DO ((x l (CDR x))
         (j 0 (1+ j)))
        ((ATOM x) j)))
```

Cette fonction est transformée en :

```
(DE LNGT (l)
    (LET ((x l) (j 0))
         (UNTIL (ATOM x)
                (PSETQ x (CDR x) j (1+ j)))
         j))
```

Une manière d'écrire la fonction REVERSE

```
(DE REV (l)
    (DO ((x l (CDR x))
         (y () (CONS (CAR x) y)))
        ((ATOM x) y)))
```

Cette fonction est transformée en :

```
(DE REV (l)
    (LET ((x l) (y ()))
         (UNTIL (ATOM x)
                (PSETQ x (CDR x) y (CONS (CAR x) y)))
         y))
```

(DO* <lv> <lr> <ec1> ... <ecN>) [MACRO]

est identique à la fonction précédente, toutefois les liaisons ont lieu séquentiellement et non plus en parallèle. DO* est à DO, ce que LET* et à LET.

DO* peut être défini en Lisp de la manière suivante :

Cette définition n'optimise pas les BLOCK et les TAGBODY vides

```
(DMD DO* (lv (test . resul) . corps)
     `(BLOCK ()
            (LET* ,(MAPCAR (LAMBDA (x) (LIST (CAR x) (CADR x)))
                           lv)
                  (UNTIL ,test
                      (TAGBODY ,@corps)
                      (SETQ ,@(MAPCAN (LAMBDA (x)
                                        (WHEN (CONSP (CDDR x))
                                            (LIST (CAR x) (CADDR x))))
                                      lv)))
                  ,@resul)))
```

Exemple d'une fonction d'exponentiation

```
(DE EXPT (m n)
    (DO* ((resultat m (* m resultat))
          (exposant n (1- exposant))
          (compteur (1- exposant) (1- exposant)))
         ((= compteur 0) resultat)))
```

Cette fonction est transformée en :

```
(DE EXPT (m n)
    (LET* ((resultat m)
           (exposant n)
           (compteur (1- exposant)))
          (UNTIL (= compteur 0)
                 (SETQ resultat (* m resultat)
                       exposant (1- exposant)
                       compteur (1- exposant)))
          resultat))
```

3.8 Les Fonctions de Contrôle Dynamiques 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.

(TAG <symb> <s1> ... <sN>) [FSUBR]

permet de définir un échappement de nom <symb>. 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 (c'est-à-dire celle de <sN>). En revanche si au cours de ces évaluations un appel du type (EXIT <symb> <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
              (LETN AUXFN ((l))
                    (COND ((NULL l) ())
                          ((EQ l e) (EXIT trouve l))
                          ((CONSP l)
                             (AUXFN (CAR l))
                             (AUXFN (CDR l)))
                          (T ())))))
     (PRESENT '(1 (2 . 3) 4) 3)     -> 3
     (PRESENT '(1 (2 . 3) 4) 5)     -> ()
```

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

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

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

permet de sortir de l'échappement de nom <symb> (c'est-à-dire retourner au dernier (TAG <symb> ...) ou (EVTAG <symb> ...) défini dynamiquement) après avoir évalué les différentes expressions <s1> ... <sN> dans l'environnement du EXIT. Si <symb> n'est pas le nom d'un échappement, EXIT déclenche l'erreur ERRUDT dont le libellé est :

```
** <fn> : échappement indéfini : <symb>     ou bien
** <fn> : undefined tag : <symb>
```

dans lequel <fn> est la fonction appelée (EXIT, EVEXIT ou UNEXIT) et <symb> est le nom de l'échappement.

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

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

(UNEXIT <symb> <s1> ... <sN>) [FSUBR]

permet de sortir de l'échappement de nom <symb>, c'est-à-dire retourner au dernier (TAG <symb> ...) ou (EVTAG <symb> ...) défini dynamiquement, puis, dans l'environnement de définition du TAG ou du EVTAG, évaluation des différentes expressions <s1> ... <sN>.

```
ex : (LET ((i 10)) (TAG OUT (LET ((i 20)) (EXIT OUT i))))
                   -> 20
     (LET ((i 10)) (TAG OUT (LET ((i 20)) (EVEXIT 'OUT i))))
                   -> 20
```

```
(LET ((i 10)) (TAG OUT (LET ((i 20)) (UNEXIT OUT i))))
                -> 10
```

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

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

```
(UNTILEXIT <symb> <s1> ... <sN>)
```

; correspond donc à l'appel :

```
(TAG <symb> (WHILE T <s1> ... <sN>))
```

(LOCK <fn> <s1> ... <sN>) [FSUBR]

en premier lieu LOCK évalue l'argument <fn> qui doit retourner en valeur une fonction à 2 arguments. Cette fonction se nomme la *fonction de verrouillage* Puis LOCK évalue comme un PROGN les différentes expressions <s1> ... <sN>. Si au cours de ces évaluations un échappement se produit, la fonction de verrouillage est automatiquement appelée avec 2 arguments : le premier est le nom de l'échappement en cours et le second la valeur que retourne cet échappement. La valeur retournée par cette fonction de verrouillage (si elle en retourne une) devient la valeur de la fonction LOCK. Si l'évaluation du corps du LOCK se termine normalement, la fonction de verrouillage est également appelée avec comme nom d'échappement () et comme valeur celle de l'expression <sN>, mais c'est la valeur retournée par la fonction de verrouillage qui devient la valeur de LOCK.

Voici une fonction de verrouillage qui est ineffective car elle relance un échappement de même nom avec la même valeur.

```
(LAMBDA (tag val) (IF tag (EVEXIT tag val) val))
```

Voici une fonction de verrouillage qui arrête tous les échappements en provoquant une erreur :

```
(LAMBDA (tag val)
        (IF tag
            (ERROR 'LOCK 'ERRUDT tag)
            val))
```

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

évalue les différentes expressions <s1> ... <sN> et retourne la valeur de la 1ère évaluation (c'est-à-dire celle de <s1>). Son comportement dans le cas le plus simple est donc équivalent à la fonction 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 par exemple de restaurer automatiquement des environnements complexes en cas de sorties extraordinaires ou d'erreurs.

PROTECT peut être défini en Lisp de la manière suivante :

```
(DF PROTECT (e1 . l)
    (LOCK (LAMBDA (tag val)
                  (EPROGN l)
                  (IF tag
                      (EVEXIT tag val)
                      val))
          (EVAL e1)))
```

```
ex : ; Fonction d'impression d'un nombre en hexadécimal
     ; décalé de 2 positions à gauche :
     ? (DE PRINHEX (n)
     ?    (PROTECT (PROGN (OBASE 16)
     ?                    (PRINT (* n 4)))
     ?             (OBASE 10)))
     = 10
     ? (PRINHEX 100)
     = 190
     ? (PRINHEX 'foo)
               *** : l'argument n'est pas un nombre entier : foo
     ? 100
     = 100     ; la base de sortie a été restaurée!
```

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

après avoir dépilé <n> blocs d'activation de la pile d'exécution dynamique (voir la fonction CSTACK), évalue les différentes expressions <s1> ... <sN> et retourne la valeur de la dernière évaluation (c'est-à-dire celle de <sN>). Ces évaluations sont donc réalisées dans un environnement antérieur. S'il ne reste pas au moins <n> blocs d'activation dans la pile d'exécution dynamique l'erreur fatale ERRFSUD se déclenche dont le libellé est :

```
***** Erreur fatale : pile vide.      ou bien
***** Fatal error : stack underflow.
```

Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (LET ((x 10)) (LET ((x '(a))) (UNWIND 1 x)))
     -> 10
     (LET ((x 10))
          (PROTECT (LET ((x '((a))) (UNWIND 2 x))
                   (PRINT 'ok)))
     ok
     -> 10
```

(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 ERRUDT (échappement indéfini) est déclenchée. Cette fonction permet donc de *filtrer* les échappements.

CATCH-ALL-BUT peut être défini en Lisp de la manière suivante :

```
(DMD CATCH-ALL-BUT (l . corps)
     `(LOCK (LAMBDA (tag val)
              (COND ((NULL tag) val)
                    ((MEMQ tag ',l) (EVEXIT tag val))
                    (T (ERROR 'CATCH-ALL-BUT
                              'ERRUDT
                              tag))))
            ,@corps)))
```

(BACKTRACK <symb> <l> <e1> ... <eN>) [FSUBR]

permet de réaliser une structure de contrôle de type *retour arrière*. <symb> est un symbole qui est le nom du retour arrière, <l> est une liste de variables à protéger et <e1> ... <eN> les différentes actions de ce retour arrière. BACKTRACK évalue <e1> et retourne sa valeur mais si au cours de cette évalation un échappement de nom <symb> se produit, il y a abandon de cette évaluation, restauration des valeurs des variables de la liste <l> et évaluation de <e2> qui peut à son tour être abandonnée au profit de <e3> ... Pour sortir d'un retour arrière d'une manière extraordinaire il faut utiliser la forme :

```
(EXIT BACKTRACK <valeur de retour>).
```

BACKTRACK peut être défini en Lisp de la manière suivante :

```
(DMD BACKTRACK (name lvar . body)
     (UNLESS (SYMBOLP name)
             (ERROR 'BACKTRACK 'ERRNAA name))
     `(TAG backtrack
           ,@(IF (NULL lvar)
                 (MAPCAR (LAMBDA (e)
                           `(TAG ,name ,e
                                 (EXIT backtrack)))
                         body)
                 `((LET ((backtrack (LIST ,@lvar)))
                        ,@(MAPCAN
                            (LAMBDA (e)
                              `((TAG ,name ,e
                                     (EXIT backtrack))
                                (DESETQ ,lvar backtrack)))
                            body))))))))))
```

```
ex : (PPRINT (MACROEXPAND1
       '(BACKTRACK RE
                   (s1 s2)
                   (ESSAI1 s1)
                   (ESSAI2 s2)
                   (ESSAI3 s1 s2))))

     (TAG backtrack
          (LET ((backtrack (LIST s1 s2)))
               (TAG re (ESSAI1 s1) (EXIT backtrack))
               (DESETQ (s1 s2) backtrack)
               (TAG re (ESSAI2 s2) (EXIT backtrack))
               (DESETQ (s1 s2) backtrack)
```

```
(TAG re (ESSAI3 s1 s2) (EXIT backtrack))
(DESETQ (s1 s2) backtrack)))
```

3.9 Les Prédicats de Base

En Lisp la valeur booléenne *fausse* est assimilée à la valeur () et la valeur booléenne *vraie* est assimilée à toute valeur différente de () (ce qui laisse un très grand choix de représentation ...).
Certains prédicats devant retourner une valeur booléenne *vraie* utiliseront la constante symbolique T (initiale du TRUE anglais) qui par définition possède une valeur différente de () (la valeur du symbole T est le symbole T lui-même).
La règle d'utilisation de cette valeur par les prédicats est la suivante : tout prédicat qui est faux pour la valeur () retourne le premier argument si le prédicat est vérifié et () dans le cas contraire; tout prédicat qui est vrai pour la valeur () retourne T si le prédicat est vérifié et () dans le cas contraire.

La plupart des noms de ces fonctions se terminent par la lettre P, qu'il faut prendre dans le sens de *(P)rédicat.*

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

(FALSE <e1> ... <eN>) [SUBR à 0 ou N arguments]

retourne la valeur booléenne fausse, c'est-à-dire la valeur (), quelque soit le nombre ou la valeur de ses arguments.

(TRUE <e1> ... <eN>) [SUBR à 0 ou N arguments]

retourne la valeur booléenne vraie, c'est-à-dire la constante symbolique T, quelque soit le nombre ou la valeur de ses arguments.

(NULL <s>) [SUBR à 1 argument]

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

NULL peut être défini en Lisp de la manière suivante :

```
(DE NULL (s)
   (IF (EQ s ()) T ()))
ex : (NULL ())     -> T
     (NULL T)      -> ()
```

(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 préférable 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]
(ATOMP <s>) [SUBR à 1 argument]

teste si <s> est un atome, c'est-à-dire un symbole, un nombre ou une chaîne de caractères. ATOM retourne T si ce test est vérifié, () dans le cas contraire. Pour permettre la relation (ATOM x) == (NOT (CONSP x)), ATOM retourne T si son argument est un vecteur de S-expressions. Il n'y a aucune différence entre les fonctions ATOM et ATOMP.

```
ex : (ATOM ())            -> T
     (ATOM 'ARGH)         -> T
     (ATOM #[1 2])        -> T
     (ATOM "OUPS")        -> T
     (ATOM 42)            -> T
     (ATOM '(A B))        -> ()
```

(CONSTANTP <s>) [SUBR à 1 argument]

teste si <s> est une constante, c'est-à-dire un objet qui évalué retourne le même objet (ou bien l'objet spécial NIL). Sont des constantes les types de nombres de base, les chaînes de caractères, les vecteurs de S-expressions et les symboles ne répondant pas au prédicat VARIABLEP. CONSTANTP retourne T si le test est vérifié et () dans le cas contraire.

CONSTANTP peut être défini en Lisp de la manière suivante :

```
(DE CONSTANTP (s)
    (IF (OR (FIXP s)
            (FLOATP s)
            (VECTORP s)
            (STRINGP s)
            (AND (SYMBOLP s) (NOT (VARIABLEP s))))
        T
        ()))
```

```
ex : (CONSTANTP ())       -> T
     (CONSTANTP NIL)      -> T
     (CONSTANTP 'NIL)     -> T
     (CONSTANTP 123)      -> T
     (CONSTANTP 1.14)     -> T
     (CONSTANTP "Barf")   -> T
     (CONSTANTP #[1 2])   -> T
     (CONSTANTP 'ARGH)    -> ()
     (CONSTANTP '(A B))   -> ()
```

(SYMBOLP <s>) [SUBR à 1 argument]

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

```
ex : (SYMBOLP ())         -> T
     (SYMBOLP 'ARGH)      -> T
     (SYMBOLP #[1 2])     -> ()
     (SYMBOLP "OUPS")     -> ()
     (SYMBOLP 44)         -> ()
     (SYMBOLP '(A B))     -> ()
```

(VARIABLEP <s>) [SUBR à 1 argument]

teste si <s> est un symbole qui n'est pas une constante symbolique c'est-à-dire un symbole différent de (), ||, NIL ou de T. Retourne <s> si le test est vérifié et () dans le cas contraire.

```
ex : (VARIABLEP ())        -> ()
     (VARIABLEP NIL)       -> ()
     (VARIABLEP 'NIL)      -> ()
     (VARIABLEP 'ARGH)     -> ARGH
     (VARIABLEP 123)       -> ()
     (VARIABLEP "Barf")    -> ()
     (VARIABLEP #[1 2])    -> ()
     (VARIABLEP '(A B))    -> ()
```

(NUMBERP <s>) [SUBR à 1 argument]

teste si <s> est un nombre. NUMBERP retourne le nombre <s> si le test est vérifié et () dans le cas contraire. Les tests des différents types de nombres sont décrits au chapitre 4.

```
ex : (NUMBERP ())          -> ()
     (NUMBERP 'ARGH)       -> ()
     (NUMBERP #[1 2])      -> ()
     (NUMBERP "OUPS")      -> ()
     (NUMBERP 44)          -> 44
     (NUMBERP '(A B))      -> ()
```

(VECTORP <s>) [SUBR à 1 argument]

teste si <s> est un vecteur. VECTORP retourne <s> si le test est vérifié et () dans le cas contraire.

```
ex : (VECTORP ())          -> ()
     (VECTORP 'ARGH)       -> ()
     (VECTORP #[1 2])      -> #[1 2]
     (VECTORP "OUPS")      -> ()
     (VECTORP 44)          -> ()
     (VECTORP '(A B))      -> ()
```

(STRINGP <s>) [SUBR à 1 argument]

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

```
ex : (STRINGP ())          -> ()
     (STRINGP 'ARGH)       -> ()
     (STRINGP #[1 2])      -> ()
     (STRINGP "OUPS")      -> "OUPS"
     (STRINGP 44)          -> ()
     (STRINGP '(A B))      -> ()
```

(CONSP <s>) [SUBR à 1 argument]

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

```
ex : (CONSP ())        -> ()
     (CONSP NIL)       -> ()
     (CONSP 'NIL)      -> ()
     (CONSP 'ARGH)     -> ()
     (CONSP #[1 2])    -> ()
     (CONSP "OUPS")    -> ()
     (CONSP 44)        -> ()
     (CONSP '(A B))    -> (A B)
```

(LISTP <s>) [SUBR à 1 argument]

teste si <s> est une liste qui peut être vide (c'est-à-dire égale au symbole ||). LISTP retourne le symbole T si le test est vérifié et () dans le cas contraire. La représentation de la liste vide à la fois sous la forme () ou sous la forme d'un symbole (|| pour Le_Lisp) pose le problème du statut de ce symbole : || est à la fois un symbole et la représentation de la liste vide. Le_Lisp, comme la plupart des *post-maclisp Lisps,* résout cette ambiguité en introduisant deux fonctions testant les listes, CONSP et LISTP.

```
ex : (LISTP ())        -> T
     (LISTP ||)        -> T
     (LISTP '||)       -> T
     (LISTP NIL)       -> T
     (LISTP 'NIL)      -> ()
     (LISTP 'ARGH)     -> ()
     (LISTP #[1 2])    -> ()
     (LISTP "OUPS")    -> ()
     (LISTP 44)        -> ()
     (LISTP '(A B))    -> T
```

(NLISTP <s>) [SUBR à 1 argument]

teste si <s> n'est ni une liste ni le symbole marquant la liste vide (c'est-à-dire pour Le_Lisp le symbole ||). NLISTP retourne l'argument <s> si le test est vérifié et () dans le cas contraire. Cette fonction est donc l'inverse de la fonction précédente (à la valeur retournée près).

NLISTP peut être défini en Lisp de la manière suivante :

```
(DE NLISTP (x)
    (IF (OR (NULL x) (CONSP x))
        ()
        x))
```

```
ex : (NLISTP ())       -> ()
     (NLISTP NIL)      -> ()
     (NLISTP 'NIL)     -> NIL
     (NLISTP 'ARGH)    -> ARGH
     (NLISTP #[1 2])   -> #[1 2]
     (NLISTP "OUPS")   -> "OUPS"
     (NLISTP 44)       -> 44
     (NLISTP '(A B))   -> ()
```

(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 () 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, des vecteurs 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, la fonction EQVECTOR pour tester l'égalité des vecteurs et la fonction EQSTRING pour tester l'égalité des chaînes.

```
ex : (EQ () NIL)              -> T
     (EQ NIL 'NIL)            -> ()
     (EQ 'A (CAR '(A)))       -> T
     (EQ #[1 2] #[1 2])       -> T ou ()
                              (en fonction des implantations)
     (EQ "STRR" "STRR")       -> T ou ()
                              (en fonction des implantations)
     (EQ (1+ 119) 120)        -> T ou ()
                              (en fonction des implantations)
     (EQ '(A B) '(A B))       -> ()
     (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) () 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. A la valeur retournée près, si <s1> et <s2> sont des symboles, EQUAL est identique à la fonction EQ; si <s1> et <s2> sont des nombres entiers ou flottants EQUAL est identique à la fonction EQN ou FEQN; si <s1> et <s2> sont des vecteurs, EQUAL est identique à la fonction EQVECTOR; si <s1> et <s2> sont des chaînes de caractères, EQUAL est identique à la fonction EQSTRING. Si <s1> et <s2> sont des listes, EQUAL teste si elles possèdent la même structure (c'est-à-dire si les listes possèdent les mêmes éléments). Si le test est verifié, EQUAL retourne toujours T, dans le cas contraire EQUAL retourne ().
Pour les nombres des arithmétiques étendues, il est nécessaire d'utiliser la fonction =.

EQUAL peut être défini en Lisp de la manière suivante :

```
(DE EQUAL (s1 s2)
    (TAG NO (IF (EQUALAUX s1 s2) T ())))

(DE EQUALAUX (s1 s2)
```

```
(COND ((SYMBOLP s1)
       (IF (SYMBOLP s2)
           (EQ s1 s2)
           (EXIT NO ())))
      ((OR (FIXP s1) (FLOATP s1))
       (IF (OR (FIXP s2) (FLOATP s2))
           (= s1 s2)
           (EXIT NO ())))
      ((VECTORP s1)
       (IF (VECTORP s2)
           (EQVECTOR s1 s2)
           (EXIT NO ())))
      ((STRINGP s1)
       (IF (STRINGP s2)
           (EQSTRING s1 s2)
           (EXIT NO ())))
      (T (IF (CONSP s2)
             (AND (EQUALAUX (CAR s1) (CAR s2))
                  (EQUALAUX (CDR s1) (CDR s2)))
             (EXIT NO ()))))
```

```
ex : (EQUAL () ())                              -> T
     (EQUAL 'a '|A|)                            -> ()
     (EQUAL 1214 (1+ 1213))                     -> T
     (EQUAL 10 10.)                             -> T
     (EQUAL #[1 2] #[1 2])                      -> T
     (EQUAL #[1 2 3] #[1 2])                    -> ()
     (EQUAL "Foo bar" "Foo bar")                -> T
     (EQUAL '(A (B . C) D) '(A (B . C) D))      -> T
     (EQUAL '(A B C) '(A B C D))                -> ()
```

; McCarthy a proposé de chercher des solutions non triviales
; pour <s> tel que (EQUAL <s> (EVAL <s>)).
; Voici trois solutions proposées par Christian Queinnec :

```
(SETQ s1 '((LAMBDA (s) (LIST (CAR s)
                             (SUBST s '= (CADR s))))
           '((LAMBDA (s)
                (LIST (CAR s)
                      (SUBST s '= (CADR s))))
             '=)))

(EQUAL s1 (EVAL s1))    ->   T

(SETQ s2 '((LAMBDA (s) (LIST (CAR s) (KWOTE s)))
           '((LAMBDA (s) (LIST (CAR s) (KWOTE s))))))

(EQUAL s2 (EVAL s2))    ->   T

(SETQ s3 '((FLAMBDA (s) `(,s ,s))
           (FLAMBDA (s) `(,s ,s))))

(EQUAL s3 (EVAL s3))    ->   T
```

(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) () T))
```

3.10 Les Fonctions sur les Listes

3.10.1 Les fonctions de recherche dans les listes

Toutes ces fonctions acceptent comme argument <l> une liste qui peut être vide (). Dans tous les autres cas l'erreur de type ERRNLA est déclenchée.

(CAR <l>) [SUBR à 1 argument]

si <l> est une liste, retourne son premier élément. Si <l> égal (), CAR retourne (). Le CAR d'un symbole autre que ||, d'un nombre, d'un vecteur ou d'une chaîne de caractères est indéterminé et provoque de ce fait l'erreur ERRNLA. Cette erreur n'est plus engendrée dans du code compilé.

```
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 <l>) [SUBR à 1 argument]

si <l> est une liste, retourne cette liste sans son premier élément. Le CDR de () est () par définition. Le CDR d'un symbole autre que ||, d'un nombre, d'un vecteur ou d'une chaîne de caractères est indéterminé et provoque également l'erreur ERRNLA à l'interprétation mais non à la compilation.

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

(C....R <l>) [SUBR à 1 argument]

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

(CADR <l>) est équivalent à (CAR (CDR <l>))
(CDAAR <l>) est équivalent à (CDR (CAR (CAR <l>)))
(CADAAR <l>) est équivalent à (CAR (CDR (CAR (CAR <l>))))

(MEMQ <symb> <l>) [SUBR à 2 arguments]

si le symbole <symb> est un élément de la liste <l>, retourne la partie de la liste <l> commençant au symbole <symb>, sinon retourne (). Cette fonction utilise le prédicat EQ pour tester la présence du symbole <symb> dans la liste <l>. On utilise souvent cette fonction comme prédicat pour 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)
    (COND ((ATOM l) ())
          ((EQ at (CAR l)) l)
          (T (MEMQ at (CDR l)))))

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

(MEMBER <s> <l>) [SUBR à 2 arguments]

si l'expression <s> est un élément de la liste <l>, retourne la partie de la liste <l> commençant à l'élément <s>, sinon retourne (). 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)
    (COND ((ATOM l) ())
          ((EQUAL s (CAR l)) l)
          (T (MEMBER s (CDR l)))))

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

(TAILP <s> <l>) [SUBR à 2 arguments]

si <s> est un des CDR de <l>, TAILP retourne <s> sinon ().

peut être défini en Lisp de la manière suivante :

```
(DE TAILP (s l)
    (COND ((ATOM l) ())
          ((EQ s l) s)
          (T (TAILP s (CDR l)))))

ex : (SETQ l '(A B C D) ll (CDDR l))    -> (C D)
     (TAILP ll l)                       -> (C D)
     (TAILP '(C D) '(A B C D))          -> ()
     ; car TAILP utilise EQ
```

(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>, NTHCDR retourne (). 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)
     (NTHCDR 0 '(A B))         -> (A B)
     (NTHCDR 10 '(A B))        -> ()
```

(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))         -> ()
```

(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 (s)
    (IF (ATOM s)
        0
        (1+ (LENGTH (CDR s)))))
```

```
ex : (LENGTH '(A (B C) D E )   -> 4
     (LENGTH '(A B . C))       -> 2
     (LENGTH "Esar")           -> 0
```

3.10.2 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 contienne la liste des
; cellules allouables, CONS se décrit 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 s '(X Y Z))                 -> (X Y Z)
     (EQUAL (CONS (CAR s) (CDR s)) s)  -> T
```

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

cette fonction est équivalente à :

```
(CONS <s2> <s1>)
```

mais l'ordre des arguments est inversé.

```
ex : (XCONS 1 2)                  -> (2 . 1)
     (SETQ x 10)                  -> 10
     (XCONS (INCR x) (INCR x))    -> (12 . 11)
```

(NCONS <s>) [SUBR à 1 argument]

cette fonction correspond à :

```
(CONS <s> ())  ou bien à  (LIST <s>)
```

(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 en supposant au moins 2 arguments :

```
(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 (mcons . l)
    (LIST 'CONS (CADR l)
          (IF (NULL (CDDDR l))
              (CADDR l)
              (CONS 'MCONS (CDDR l)))))
```

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

(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> ())) ...)).

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 (NEWR r (EVAL (NEXTL l))))
         r))
```

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

(KWOTE <s>) [SUBR à 1 argument]

construit une liste de 2 éléments dont le 1er est le symbole 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
```

(MAKELIST <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.

MAKELIST peut être défini en Lisp de la manière suivante :

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

```
ex : (MAKELIST 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>. Les arguments qui ne sont pas des listes sont ignorés sauf le dernier <lN> qui, atomique, devient le dernier CDR de la liste résultat. Pour copier le premier niveau d'une liste <l>, il faut utiliser la forme : (APPEND <l> ()).

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) (APPEND2 (CDR l1) l2))))
```

```
ex : (APPEND '(A B C) '(X Y))      ->  (A B C X Y)
     (APPEND () '(X Y) '(Z))       ->  (X Y Z)
     (APPEND '(A) 1 '(B) 2 '(C))   ->  (A B C)
     (SETQ l1 '(A B C))            ->  (A B C)
     (SETQ l2 (APPEND l1))         ->  (A B C)
     (EQ l1 l2)                    ->  T
     (SETQ l3 (APPEND l1 ()))      ->  (A B C)
     (EQ l1 l3)                    ->  ()
```

(APPEND1 <l> <s>) [SUBR à 2 arguments]

est équivalent à l'appel :

(APPEND <l> (LIST <s>))

ce qui permet donc de copier le 1er niveau de la liste <l> et de lui rajouter en queue l'élément <s>.

APPEND1 peut être défini en Lisp de la manière suivante :

```
(DE APPEND1 (l s)
    (APPEND l (LIST s)))
```

```
ex : (APPEND1 '(A B) 'C)    ->  (A B C)
```

(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)
    (LETN REV2 ((s s) (r ()))
          (IF (ATOM s)
              r
              (REV2 (CDR s) (CONS (CAR s) r)))))
```

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

```
; On prétend parfois que la fonction suivante
; inverse aussi son argument :
(DE REV (l)
    (IF (NULL (CDR l))
        l
        (CONS (CAR (REV (CDR l)))
              (REV (CONS (CAR l)
                         (REV (CDR (REV (CDR l))))))))))
```

(COPYLIST <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. Cette fonction ne copie que les cellules de liste. Pour copier les chaînes de caractères, les cellules de liste étiquetées et les vecteurs de S-expressions il faut utiliser la fonction suivante, COPY.

COPYLIST peut être défini en Lisp de la manière suivante :

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

```
ex : (COPYLIST 'A)                -> A
     (COPYLIST '(A (B (C (D))))   -> (A (B (C (D))))
     (COPYLIST '#(A B . #(D)))    -> (A B D)
```

```
; cette fonction ne traite pas les listes circulaires
; ou partagées. Pour cela il faut définir la fonction :
(DE CIRCOPYLIST (l)
    (LET ((d) (e))
         (LETN CIRC1 ((l l))
               (COND ((ATOM l) l)
                     ((CDR (ASSQ l d)))
                     (T (SETQ e (CONS () ())
                              d (ACONS l e d))
                        (RPLAC e
                               (CIRC1 (CAR l))
                               (CIRC1 (CDR l))))))))
```

(COPY <s>) [SUBR à 1 argument]

retourne une copie complète de l'objet <s> y compris les atomes de type chaînes de caractères, les cellules de liste étiquetées et les vecteurs de S-expressions.

COPY peut être défini en Lisp de la manière suivante :

```
(DE COPY (s)
    (COND ((ATOM s)
           (COND
            ((STRINGP s) (SUBSTRING s 0))
            ((VECTORP s)
               (LET ((v (MAKEVECTOR (VLENGTH s) ())))
                    (FOR (i 0 1 (1- (VLENGTH s)))
                         (VSET v i
                             (COPY (VREF s i))))
                    v))
            (T s)))
          ((TCONSP s)
             (TCONS (COPY (CAR s)) (COPY (CDR s))))
          (T (CONS (COPY (CAR s)) (COPY (CDR s))))))
```

```
ex : (SETQ strg "gdy jest")          -> "gdy jest"
     (EQ strg strg)                  -> T
     (EQ strg (COPY strg))           -> ()
     (EQUAL strg (COPY strg))        -> T
     (SETQ vect #[1 2]))             -> #[1 2]
     (EQ vect vect)                  -> T
     (EQ vect (COPY vect))           -> ()
     (EQUAL vect (COPY vect))        -> T
     (COPY '#(A B . #(D)))           -> #(A B . #(D))
```

(FIRSTN <n> <l>) [SUBR à 2 arguments]

retourne une copie des <n> premiers éléments de la liste <l>. Si la liste <l> possède moins de <n> éléments (c'est-à-dire 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)
     (FIRSTN 0 '(A B C))            -> ()
```

(LASTN <n> <l>) [SUBR à 2 arguments]

retourne une copie des <n> derniers éléments de la liste <l>. Si la liste contient moins d'éléments que demandé (c'est-à-dire 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)
    (REVERSE (FIRSTN n (REVERSE l))))
```

```
ex : (LASTN 2 '(A B C D E))    -> (D E)
     (LASTN 10 '(A B C))       -> (A B C)
     (LASTN 0 '(A))            -> ()
```

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

fabrique une nouvelle copie de l'expression <s> en substituant à chaque occurrence de l'expression <s2>, l'expression <s1>. Cette substitution a lieu à tous les niveaux de <s>. Cette fonction utilise le prédicat EQUAL pour réaliser le test et fabrique une nouvelle expression en partageant un maximum de cellules de liste avec l'expression initiale.
Il faut utiliser la forme :

(NSUBST <s1> <s2> (COPY <s>))

pour obtenir une copie complète de l'expression.

SUBST peut être défini en Lisp de la manière suivante :

```
(DE SUBST (new old s)
    (COND ((EQUAL s old) new)
          ((ATOM s) s)
          (T (LET ((car (SUBST new old (CAR s)))
                   (cdr (SUBST new old (CDR s))))
                  (IF (AND (EQ car (CAR s))
                           (EQ cdr (CDR s)))
                      s
                      (CONS car cdr))))))
```

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

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

retourne une copie de <l> dans laquelle toutes les occurrences du symbole <symb> ont été enlevées. Cette modification ne porte que sur le premier niveau de <l>.

REMQ peut être défini en Lisp de la manière suivante :

```
(DE REMQ (symb l)
    (COND ((ATOM l) l)
          ((EQ symb (CAR l)) (REMQ symb (CDR l)))
          (T (CONS (CAR l) (REMQ symb (CDR l))))))
```

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

(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 REMOVE (s l)
    (COND ((ATOM l) l)
          ((EQUAL s (CAR l)) (REMOVE s (CDR l)))
          (T (CONS (CAR l) (REMOVE s (CDR l))))))
ex : (REMOVE '(A) '(A B (A) (C A B) (A) S))
          -> (A B (C A B) S)
     (REMOVE '(A) '((A) (A) (A))          -> ()
```

3.10.3 Les fonctions sur les cellules de liste étiquetées

Ces fonctions utilisent un *bit invisible* qui peut être positionné et testé pour chaque cellule de liste. La représentation externe d'une cellule de liste étiquetée (c'est-à-dire d'une cellule de liste dont le bit invisible est positionné) est par défaut :

#(<car> . <cdr>)

pour les fonctions d'entrée et de sortie. Toutefois il est possible de faire évaluer et de faire imprimer ces cellules de liste étiquetées au moyen de fonctions utilisateur (voir au chapitre 2 la section traitant des types étendus).

(TCONSP <s>) [SUBR à 1 argument]

retourne <s> si <s> est un CONS étiqueté sinon TCONSP retourne (). Cette fonction ne peut pas être décrite en Lisp

(TCONSMK <s>) [SUBR à 1 argument]

force le bit invisible sur le CONS <s>. Cette fonction ne peut pas être décrite en Lisp.

```
ex : (SETQ x '(a . b))     -> (a . b)
     (TCONSMK x)           -> #(a . b)
     x                     -> #(a . b)
```

(TCONSCL <s>) [SUBR à 1 argument]

retire le bit invisible sur le CONS <s>. Cette fonction ne peut pas être décrite en Lisp.

```
ex : (SETQ x '#(a . b))     -> #(a . b)
     (TCONSCL x)            -> (a . b)
     x                      -> (a . b)
```

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

fabrique un CONS étiqueté avec <s1> en position CAR et <s2> en position CDR.

TCONS peut être défini en Lisp de la manière suivante :

```
(DE TCONS (car cdr)
    (TCONSMK (CONS car cdr)))
ex : (SETQ x (TCONS 'a '(b c)))        -> #(a b c)
     (TCONS 'd (CONS 'e x))            -> #(d e . #(a b c))
```

3.10.4 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 physique 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, l'erreur de type ERRNLA 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, l'erreur de type ERRNLA se produit.

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

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

remplace le CAR de la liste <l> par <s1> et le CDR de la liste <l> par <s2> et retourne la liste modifiée. Si l'argument <l> n'est pas une liste, l'erreur de type ERRNLA apparaît.

RPLAC peut être défini en Lisp de la manière suivante :

```
(DE RPLAC (l s1 s2)
    (IF (ATOM l)
        (ERROR 'RPLAC 'ERRNLA l)
        (RPLACA l s1)
        (RPLACD l s2)
        l))
ex : (SETQ l1 '(A B) l2 l1)      -> (A B)
     (RPLAC l1 1 '(2))           -> (1 2)
     l1                          -> (1 2)
     l2                          -> (1 2)
```

(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>. Si <ln> n'est pas une liste, elle est transformée en (PROGN <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)
    (IF (ATOM l)
        (ERROR 'DISPLACE 'ERRNLA l)
        (IF (ATOM ln)
            (RPLAC l 'PROGN (LIST ln))
            (RPLAC l (CAR ln) (CDR ln)))))
```

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

(PLACDL <l> <s>) [SUBR à 2 arguments]

permet d'accrocher dans le CDR de <l> une nouvelle cellule dont le CAR est <s>. PLACDL retourne cette nouvelle cellule. Cette étrange fonction est en fait très utilisée pour fabriquer des listes *dans le bon ordre* en un seul parcours non récursif.

PLACDL peut être défini en Lisp de la manière suivante :

```
(DE PLACDL (l s)
    (CDR (RPLACD l (CONS s ()))))
```

```
ex : (DE EVLIS (l)
         (COND ((NULL l) ())
               ((NULL (CDR l)) (LIST (EVAL (CAR l))))
               (T (LET ((tete (LIST (EVAL (CAR l)))))
                       (EVLISAUX (CDR l) tete)
                       tete)))))

     (DE EVLISAUX (rest cour)
         (WHEN rest
               (EVLISAUX (CDR rest)
               (PLACDL cour (EVAL (CAR rest))))))
```

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

concatène physiquement toutes les listes <li> (c'est-à-dire place dans le CDR du dernier élément de <li-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 forçage dans le dernier CDR de la liste d'un pointeur sur le 1er élément de cette même liste. Les arguments qui ne sont pas des listes sont ignorés sauf le dernier <lN> qui, atomique, devient le dernier CDR de la liste résultat.

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

```
(DE NCONC2 (l1 l2)
    ; suppose que l1 et l2 sont des listes bien formées
    (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 G))   -> (A B C D E F G)
     X2                         -> (A B C D E F G)
     (NCONC () '(A) () '(B))    -> (A B)
     (SETQ X1 '(A B C))         -> (A B C)
     (NCONC X1 X1)              -> (A B C A B C A B ...
```

(NCONC1 <l> <a>) [SUBR à 2 arguments]

rajoute physiquement à la fin de la liste <l> le nouvel élément <a>.

NCONC1 peut être défini en Lisp de la manière suivante :

```
(DE NCONC1 (l a)
    (NCONC l (LIST a)))
```

```
ex : (SETQ L1 '(A B C) L2 L1)   -> (A B C)
     (NCONC1 L1 'D)             -> (A B C D)
     l2                         -> (A B C D)
```

(CIRLIST <e1> ... <eN>) [SUBR à N arguments]

fabrique une liste circulaire comprenant les éléments <e1> ... <eN>. Cette fonction fabrique une nouvelle liste et ne modifie donc pas les éléments composant la liste circulaire. CIRLIST permet entre autres de fabriquer des invariants dans les fonctions de type MAP.

CIRLIST peut être défini en Lisp de la manière suivante :

```
(DE CIRLIST l
    (NCONC l l))
```

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

(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 NREVERSE (l)
    (IF (CONSP l)
        (LETN NR ((l l) (r))
              (IF (NULL (CDR l))
                  (RPLACD l r)
                  (NR (CDR l) (RPLACD l r))))
        l))
```

```
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)
     (NREVERSE L1)          -> (E D C B A)
     L1                     -> (A)
     L2                     -> (B A)
     L3                     -> (E D C B A)
```

(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 être manipulée avec précaution car elle modifie physiquement des structures Le_Lisp (en particulier les modifications peuvent être bouleversantes en cas de structures partagées), mais elle est évidemment beaucoup plus rapide que la fonction traditionnelle REVERSE.

NRECONC peut être défini en Lisp de la manière suivante :

```
(DE NRECONC (l s)
    (IF (CONSP l)
        (NCONC (NREVERSE 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)
```

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

```
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 <symb> <l>) [SUBR à 2 arguments]

enlève physiquement toutes les occurrences du symbole <symb> au premier niveau de la liste <l> et retourne la liste ainsi modifiée. Si cette liste était partagée (par exemple si une variable pointait sur elle) les anciens pointeurs sur cette liste sont imprédictibles. Cette fonction utilise le prédicat EQ. Pour réaliser une nouvelle copie d'une liste, il faut utiliser la fonction REMQ.

DELQ peut être défini en Lisp de la manière suivante :

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

```
ex : (SETQ l '(A B C B B D))   ->  (A B C B B D)
     (SETQ l (DELQ 'B l))      ->  (A C D)
     (DELQ 'A l)               ->  (C D)
     l                         ->  (A C D)
     (SETQ l (DELQ 'A l))      ->  (C D)
     l                         ->  (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> et retourne la liste ainsi modifiée. Si cette liste était partagée (par exemple si une variable pointait sur elle) les anciens pointeurs sur cette liste sont imprédictibles. Cette fonction utilise le prédicat EQUAL. Pour réaliser une nouvelle copie d'une 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)
     (SETQ l (DELETE '(B) l))      ->  (A C B D)
     l                             ->  (A C B D)
```

(SORT <fn> <l>) [SUBR à 2 arguments]

réalise le tri physique de la liste <l>. Les éléments de cette liste vont être comparés au moyen du prédicat à 2 arguments <fn>. Si cette liste était partagée (par exemple si une variable pointait sur elle) les anciens pointeurs sur cette liste sont imprédictibles. SORT retourne la liste <l> modifiée en valeur. Le tri réalisé est *instable* : les éléments identiques de la liste de départ ne sont pas forcément au même endroit dans la liste résultat.

SORT peut être défini en Lisp de la manière suivante :

```
(DE SORT (fn l)
    (IF (NULL (CDR l))
        l
        (LET ((L1 l) (L2))
             (SETQ l  (NTHCDR (1- (DIV (LENGTH l) 2)) l)
                   l2 (CDR l))
```

```
                (RPLACD l ())
                (FFUSION (SORT fn l1)
                         (SORT fn l2)))))
    (DE FFUSION (l1 l2)
        ; fusionne physiquement les 2 listes triées
        (UNLESS (FUNCALL fn (CAR l1) (CAR l2))
                (PSETQ l1 l2 l2 l1))
        (PROG1 l1
               (WHILE (AND (CDR l1) l2)
                      (WHEN (FUNCALL fn (CAR l2) (CADR l1))
                            (RPLACD l1
                                    (PROG1 l2
                                           (SETQ l2 (CDR l1)))))
                      (NEXTL l1))
               (WHEN l2 (RPLACD l1 l2))))
```

(SORTL <l>) [SUBR à 1 argument]

réalise le tri physique alphabétique de la liste <l>.

SORTL peut être défini en Lisp de la manière suivante :

```
(DE SORTL (l) (SORT 'ALPHALESSP l))
```

```
ex : (MAPCAR 'SORTL
             '((requiem aeternam dona eis domine)
               (et lux perpetua luceat eis)
               (in memoria aeterna eris justus)
               (ab auditione mala non timebit)))
->           ((aeternam domine dona eis requiem)
              (eis et luceat lux perpetua)
              (aeterna eris in justus memoria)
              (ab auditione mala non timebit))
```

(SORTP <l>) [SUBR à 1 argument]

réalise le tri physique des symboles de la liste <l>, packages inclus.

SORTP peut être défini en Lisp de la manière suivante :

```
(DE SORTP (l) (SORT 'PKGCMP l))
```

avec :

```
(DE PKGCMP (s1 s2)
    (IF (EQ (PACKAGECELL s1) (PACKAGECELL s2))
        (ALPHALESSP s1 s2)
        (PKGCMP (PACKAGECELL s1) (PACKAGECELL s2))))
```

```
ex : (SORTP '(A #:B:A Z #:B:C:A Y))   ->
     (A Y Z #:B:A #:B:C:A)
```

(SORTN <l>) [SUBR à 1 argument]

réalise le tri physique numérique de la liste <l>.

SORTN peut être défini en Lisp de la manière suivante :

```
(DE SORTN (l) (SORT '< l))
ex : (SORTN '(6 4 8 6 5 8 7))    -> (4 5 6 6 7 8 8)
```

3.10.5 Les fonctions sur les A-listes

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

```
((clef1 . val1) (clef2 . val2)    ...    (clefN . valN))
```

Chaque élément d'une A-liste est un couple constitué d'une clef (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 clef.

Pour toutes les fonctions qui vont être décrites, l'argument <al> doit être une A-liste, et les clefs 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). Tous les éléments qui ne sont pas des CONS sont ignorés.

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

rajoute un élément à la A-liste <l>. L'élément est composé de la clef <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 : (ACONS '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 clefs
<l2> doit être une liste de valeurs
PAIRLIS retourne une nouvelle A-liste formée à partir de la liste des clefs <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 (AND (CONSP l1) (LISTP l2))
        (ACONS (CAR l1)
               (CAR l2)
               (PAIRLIS (CDR l1) (CDR l2) al))
        al))
ex : (PAIRLIS '(JA JESTEM WIELKI)  '(LE MERLE CHANTE) ())
       -> ((JA . LE) (JESTEM . MERLE) (WIELKI . CHANTE))
     (PAIRLIS '(X Y Z) '(A (B)) '((A . X) (B . Y)))
       -> ((X . A) (Y B) (Z) (A . X) (B . Y))
```

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

retourne l'élément de la A-liste <al> dont la clef (le CAR) est égale au symbole <symb>, sinon ASSQ retourne ().

ASSQ peut être défini en Lisp de la manière suivante :

```
(DE ASSQ (symb al)
    (COND ((ATOM al) ())
          ((AND (CONSP (CAR al))
                (EQ (CAAR al) symb))
           (CAR al))
          (T (ASSQ symb (CDR al)))))
```

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

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

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

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

ATTENTION : on ne peut pas distinguer la valeur () 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 <symb> <al>) [SUBR à 2 arguments]

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

RASSQ peut être défini en Lisp de la manière suivante :

```
(DE RASSQ (symb al)
    (COND ((ATOM al) ())
          ((AND (CONSP (CAR al))
                (EQ (CDAR al) symb))
           (CAR al))
          (T (RASSQ symb (CDR al)))))
```

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

(ASSOC <s> <al>) [SUBR à 2 arguments]

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

ASSOC peut être défini en Lisp de la manière suivante :

```
(DE ASSOC (s al)
   (COND
      ((ATOM al) ())
      ((AND (CONSP (CAR al))
            (EQUAL s (CAAR al)))
       (CAR al))
      (T (ASSOC s (CDR al)))))
```

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

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

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

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

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

(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 ((ATOM al) ())
          ((AND (CONSP (CAR al))
                (EQUAL s (CDAR al)))
           (CAR al))
          (T (RASSOC a (CDR al)))))
```

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

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

retourne une copie de l'expression <s> dans laquelle toutes les occurrences des clefs de la A-liste <al> ont été remplacées par leurs valeurs associées correspondantes. La copie retournée partage le maximum de cellules avec l'expression initiale. Cette fonction utilise le prédicat 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))
        (LET ((car (SUBLIS al (CAR s)))
              (cdr (SUBLIS al (CDR s))))
             (IF (AND (EQ car (CAR s))
                      (EQ cdr (CDR s)))
                 s
                 (CONS car cdr)))))
```

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

3.11 Les Fonctions sur les Symboles

3.11.1 Les 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 teste toujours si un symbole possède une valeur. Si ce n'est pas le cas il provoque l'erreur ERRUDV. Il existe en plus deux fonctions particulières d'accès aux symboles.

(BOUNDP <symb>) [SUBR à 1 argument]

Cette fonction teste si l'argument <symb> est un symbole qui possède une valeur. BOUNDP retourne T si ce test est vérifié et () dans le cas contraire. Cette fonction permet d'éviter l'erreur ERRUDV (variable indéfinie). Si l'argument n'est pas un symbole cette fonction retourne (). Du fait de la représentation des valeurs indéfinies il n'est pas possible de décrire cette fonction en Lisp.

```
ex : (BOUNDP T)          -> T
     (BOUNDP ())         -> T      ; car () est identique à ||
     (BOUNDP 'foofoo)    -> ()
                ; si foofoo n'a pas de valeur
```

(SYMEVAL <symb>) [SUBR à 1 argument]

retourne la valeur du symbole <symb>. 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 également l'erreur ERRUDV si le symbole est indéfini.

```
ex : (SETQ FOO 'BAR)     -> BAR
     (BOUNDP 'FOO)       -> T
     (SYMEVAL 'FOO)      -> BAR
     (SYMEVAL 'NIEMA)    ->
                    ** SYMEVAL : variable indéfinie : NIEMA
          ; si NIEMA ne possède pas de valeur
```

3.11.2 Les fonctions de modification des valeurs des symboles

(DEFVAR <symb> <e>) [FSUBR]

Bien que les variables n'aient pas besoin d'être déclarées, il est d'usage de définir les variables globales au moyen de cette fonction qui va affecter la valeur du symbole <symb>, qui n'est pas évalué, avec la valeur de <e> qui est évaluée. DEFVAR retourne <symb> en valeur. Comme pour les autres fonctions de définition, si la valeur du symbole #:SYSTEM:LOADED-FROM-FILE est différente de (), il y a ajout de cette valeur sur la P-liste du symbole <symb> sous l'indicateur #:SYSTEM:LOADED-FROM-FILE.

DEFVAR peut être défini en Lisp de la manière suivante :

```
(DF DEFVAR (VAR VAL)
    (SET VAR (EVAL VAL))
    (WHEN #:SYSTEM:LOADED-FROM-FILE
          (PUTPROP VAR
                   #:SYSTEM:LOADED-FROM-FILE
                   '#:SYSTEM:LOADED-FROM-FILE))
    VAR)
ex : (DEFVAR foo 100)    -> foo
     foo                 -> 100
```

(MAKUNBOUND <symb>) [SUBR à 1 argument]

change la valeur du symbole <symb> de telle manière que tout nouvel accès à sa valeur déclenche l'erreur ERRUDV (variable indéfinie). MAKUNBOUND retourne <s> en valeur.

```
ex : (SETQ x 10)        -> 10
     x                  -> 10
     (MAKUNBOUND 'x)    -> x
     x
                        ** EVAL : variable indéfinie : X
```

(SET <symb> <s>) [SUBR à 2 arguments]

change la valeur du symbole <symb> avec <s>. SET retourne en valeur <s>.

```
ex : (SET 'x (1+ 5))              -> 6
     x                            -> 6
```

(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 symbole <symi> est initialisé avec l'expression correspondante <si>. Ces affectations ainsi que les évaluations des expressions <si> sont réalisées *en séquence.* 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 elles non plus.

```
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 retourne <s1> en valeur.

PSETQ peut être défini en Lisp de la manière suivante :

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

```
ex : (SETQ x 10 y 11 z 12)  -> 12
     (PSETQ x y y z z x)    -> 11
     x                      -> 11
     y                      -> 12
     z                      -> 10
```

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

réalise le *destructuring SET* du système *NIL* [White 79]. <l1> est un arbre de variables, <l2> est une liste de valeurs. DESET va affecter aux variables de l'arbre des variables les valeurs correspondantes de l'arbre de valeurs. DESET utilise le même algorithme que l'évaluateur pour lier un arbre de paramètres à une liste 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)
               (ERROR 'DESET 'ERRWNA l2)))
          ((VARIABLEP l1) (SET l1 l2) T)
          ((ATOM l1) (ERROR 'DESET 'ERRBPA l1))
          ((AND (CONSP l1) (CONSP l2))
           (DESET (CAR l1) (CAR l2))
           (DESET (CDR l1) (CDR l2)))
          (T (ERROR 'DESET 'ERRILB (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 <l1> <l2>)
            ; est donc équivalent à
(DESET (QUOTE <l1>) <l2>)
```

(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 symbole.

; Cette fonction (à 1 argument) correspond en Lisp à :

```
(PROG1 (CAR <symb>) (SETQ <symb> (CDR <symb>)))
```

; à 2 arguments elle correspond à :

```
(PROG1 (SETQ <sym2> (CAR <sym1>))
       (SETQ <sym1> (CDR <sym1>)))
```

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

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

<symb> (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.

NEWL peut être défini en Lisp de la manière suivante :

```
(DMD NEWL (symb s)
     `(SETQ ,symb (CONS ,s ,symb)))
```

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

(NEWR <symb> <s>) [FSUBR]

<symb> (qui n'est pas évalué) doit être un symbole dont la valeur est une liste. NEWR ajoute en queue de cette liste la valeur de <s> et retourne en valeur la nouvelle liste formée. Si la valeur de <symb> n'est pas une liste, NEWR crée pour nouvelle valeur de <symb> la liste composée de l'élément <s>.

NEWR peut être défini en Lisp de la manière suivante :

```
(DMD NEWR (symb s)
     `(SETQ ,symb (NCONC ,symb (NCONS ,s))))
```

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

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

<symb> 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 fournie à l'appel de la fonction ou de 1 dans le cas contraire. Cette fonction utilise l'arithmétique générique. INCR déclenche une erreur si <symb> n'est pas une variable.

(INCR <symb> <n>) est équivalent à : (SETQ <symb> (+ <symb> <n>))
et (INCR <symb>) est équivalent à : (SETQ <symb> (1+ <symb>))

INCR peut être défini en Lisp de la manière suivante :

```
(DF INCR (var . val)
    (IF (VARIABLEP var)
        (SET var
             (IF (CONSP val)
                 (+  (SYMEVAL var) (EVAL (CAR val)))
                 (1+ (SYMEVAL var))))
        (ERROR 'INCR 'ERRNVA var)))
```

```
ex : (SETQ x 10)     -> 10
     (INCR x)        -> 11
     x               -> 11
     (INCR x 3)      -> 14
     (INCR x 2.5)    -> 16.5
     (INCR x)        -> 17.5
     x
```

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

<symb> 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 fournie à l'appel de la fonction ou de 1 dans le cas contraire. Cette fonction utilise l'arithmétique générique. DECR déclenche une erreur si <symb> n'est pas une variable.

(DECR <symb> <n>) est équivalent à : (SETQ <symb> (- <symb> <n>))
et (DECR <symb>) est équivalent à : (SETQ <symb> (1- <symb>))

DECR peut être défini en Lisp de la manière suivante :

```
(DF DECR (var . val)
    (IF (VARIABLEP var)
        (SET var
             (IF (CONSP val)
                 (-  (SYMEVAL var) (EVAL (CAR val)))
                 (1- (SYMEVAL var))))
        (ERROR 'DECR 'ERRNVA var)))
```

```
ex : (SETQ x 10)       -> 10
     (DECR x)          -> 9
     x                 -> 9
     (DECR x 3)        -> 6
     (DECR x 0.5)      -> 5.5
     (DECR x)          -> 4.5
```

3.11.3 Les fonctions sur les P-listes

En Lisp, les P-listes (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-liste. Les recherches sur les P-listes s'effectuent donc deux éléments par deux éléments.

Chaque symbole possède une P-liste qui lui est propre et qui est initialisée à () à la création du symbole.

Les arguments des fonctions sur les P-listes sont :

<pl> - un symbole dont on veut utiliser la P-liste . Les fonctions sur les P-listes déclenchent l'erreur ERRNAA ou ERRNVA si cet argument n'est pas un symbole ou si on veut écrire dans la P-liste du symbole ||.

<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> <l>) [SUBR à 1 ou 2 arguments]

si l'argument <l> n'est pas fourni, retourne la P-liste associée au symbole <pl>. Si le symbole <pl> ne possède pas de P-liste, PLIST retourne (). Si l'argument <l> est fourni, il devient la nouvelle valeur de la P-liste du symbole <pl> qui est retournée en valeur.
PLIST ne peut pas être défini simplement en Lisp.

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

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

retourne la valeur associée à l'indicateur <ind> dans la P-liste du symbole <pl>. Si l'indicateur n'existe pas, GETPROP retourne (). Il n'y a aucune différence entre les fonctions GETPROP et GET.

ATTENTION : on ne peut pas discerner une valeur égale à () associée à un indicateur avec l'absence de cet indicateur (voir la fonction suivante).

GETPROP peut être défini en Lisp de la manière suivante :

```
(DE GETPROP (pl ind)
    (LETN GET1 ((pl (PLIST pl)))
          (COND ((ATOM pl) ())
                ((EQ (CAR pl) ind) (CADR pl))
                (T (WHEN (CONSP (CDR pl))
                         (GET1 (CDDR pl)))))))

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

(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 réussit, GETL retourne la partie de la P-liste du symbole <pl> commençant à cet indicateur. GETL permet de résoudre l'ambiguité de la fonction précédente.

GETL peut être défini en Lisp de la manière suivante :

```
(DE GETL (pl l)
    ; pl = une P-liste
    ; l  = une liste d'indicateurs
    (LETN GETL1 ((pl (PLIST pl)))
          (COND
             ((ATOM pl) ())
             ((MEMQ (CAR pl) l) pl)
             (T (WHEN (CONSP (CDR pl))
                      (GETL1 (CDDR pl)))))))
```

```
ex : (PLIST 'rose)                -> (nom commun genre feminin)
     (GETL 'rose '(genre nom))    -> (nom commun genre feminin)
     (GETL 'rose '(taille genre)) -> (genre feminin)
     (GETL 'rose '(type taille))  -> ()
```

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

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

ADDPROP peut être défini en Lisp de la manière suivante :

```
(DE ADDPROP (pl pval ind)
   (PLIST pl (MCONS ind pval (PLIST pl))))
```

```
ex : (PLIST 'PLT)              -> (I1 A I2 B)
     (ADDPROP 'PLT 'C 'I1)     -> C
     (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-liste du symbole <pl>, sa valeur associée prend la nouvelle valeur <pval>, sinon l'indicateur <ind> et sa valeur associée <pval> sont ajoutés en tête de la P-liste du symbole <pl> (d'une manière identique à la fonction ADDPROP). PUTPROP retourne <pval> en valeur.

PUTPROP peut être défini en Lisp de la manière suivante :

```
(DE PUTPROP (pl pval ind)
    (LETN PUT1 ((p (PLIST pl)))
          (COND ((ATOM p) (ADDPROP pl pval ind))
                ((EQ (CAR p) ind)
                   (RPLACA (CDR p) pval) pval)
                (T (PUT1 (WHEN (CONSP (CDR p))
                               (CDDR p))))))
    pval)
```

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

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

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

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

```
ex : (PLIST 'rose ())           ->  ()
     (DEFPROP rose 123 prix)    ->  123
     (DEFPROP rose rouge couleur)  ->  rouge
     (PLIST 'rose)              ->  (couleur rouge prix 123)
```

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

enlève de la P-liste du symbole <pl> l'indicateur <ind> s'il existe ainsi que sa valeur associée. REMPROP retourne l'ancienne P-liste à partir de la propriété <ind> si celle-ci existait et () dans le cas contraire.
L'utilisation conjuguée des fonctions REMPROP et ADDPROP permet d'utiliser les P-listes comme des piles de propriétés-valeurs.

REMPROP peut être défini en Lisp de la manière suivante :

```
(DE REMPROP (pl ind)
    (LETN REM1 ((pl1 pl) (pl2 (PLIST pl)))
          (COND ((ATOM pl2) ())
                ((EQ (CAR pl2) ind)
                   (IF (ATOM pl1)
                       (PLIST pl1
                              (WHEN (CONSP (CDR pl2))
                                    (CDDR pl2)))
                       (RPLACD pl1
                               (WHEN (CONSP (CDR pl2))
                                     (CDDR pl2)))))
                (T (REM1 pl2
                         (WHEN (CONSP (CDR pl2))
                               (CDDR pl2)))))))
```

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

3.11.4 L'accès aux définitions des fonctions

Le_Lisp gère les définitions des fonctions au moyen de 2 propriétés naturelles associées à chacun des symboles :
- le FTYPE, le type de la fonction.
- la FVAL, la valeur de la fonction.

Les fonctions de cette section sont indépendantes des packages. Pour accéder à une fonction dans un certain package voir les fonctions GETFN1 et GETFN.

(TYPEFN <symb>) [SUBR à 1 argument]

retourne le type de la fonction associée au symbole <symb> ou () si le symbole ne possède pas de définition de fonction. Le type d'une fonction peut être l'un des symboles suivants : EXPR, FEXPR, MACRO, DMACRO, SUBR0, SUBR1, SUBR2, SUBR3, NSUBR, FSUBR, MSUBR ou DMSUBR. Cette fonction permet de savoir si l'erreur ERRUDF va se déclencher si on utilise le symbole <symb> en tant que fonction. Du fait de la représentation des fonctions indéfinies il n'est pas possible de décrire cette fonction en Lisp.

```
ex : (TYPEFN 'COND)          -> FSUBR
     (TYPEFN 'foofoo)        -> ()
        ; si foofoo n'a pas de valeur en tant que fonction
```

(VALFN <symb>) [SUBR à 1 argument]

retourne la valeur associée au symbole <symb> en tant que fonction ou 0 si ce symbole ne possède pas de définition de fonction. Cette valeur est une adresse pour les fonctions de type SUBR ou FSUBR et une liste dans le cas des EXPR, FEXPR, MACRO et DMACRO.

(SETFN <symb> <ftype> <fval>) [SUBR à 3 arguments]

permet d'initialiser pour le symbole <symb>, le type de sa fonction associée avec l'argument <ftype> ainsi que la valeur de sa fonction associée avec l'argument <fval>. SETFN est utilisé comme fonction de définition quand le nom de la fonction à définir est calculé.
ATTENTION : SETFN ne réalise que très peu de tests de validité, quand c'est possible il est préférable d'utiliser les véritables fonctions de définition qui contrôlent la validité de la définition.

```
ex : (SETFN 'FOO 'EXPR '((x) (+ x x)))    -> FOO
     (TYPEFN 'FOO)                        -> EXPR
     (VALFN 'FOO)                         -> ((x) (+ x x))
```

(RESETFN <symb> <ftype> <fval>) [SUBR à 3 arguments]

permet de changer pour le symbole <symb>, le type de sa fonction associée avec l'argument <ftype> ainsi que la valeur de sa fonction associée avec l'argument <fval> en s'assurant la compatibilité avec une définition précédente. Si elle n'est pas assurée le message suivant est imprimé :

*** RESETFN : fonction incompatible : <symb>*

RESETFN est utilisé par les fonctions de définition standard pour assurer la compatibilité des appels entre fonctions compilées et interprétées.

```
ex : (RESETFN 'FOO 'EXPR '((x) (+ x x)))    -> FOO
     (TYPEFN 'FOO)                          -> EXPR
     (VALFN 'FOO)                           -> ((x) (+ x x))
     (RESETFN 'FOO 'FEXPR '((x) 100))       ->
                    ** RESETFN : fonction incompatible : FOO
```

(FINDFN <s>) [SUBR à 1 argument]

recherche dans toute l'OBLIST le symbole dont la valeur en tant que fonction est égale à <s>. FINDFN retourne ce symbole si il existe et () dans le cas contraire.

FINDFN peut être défini en Lisp de la manière suivante :

```
(DE FINDFN (s)
    (TAG trouve
         (MAPOBLIST
              (LAMBDA (symb)
                      (WHEN (EQ (VALFN symb) s)
                            (EXIT trouve symb))))))
```

```
ex : (DE FOO (x) (+ x 2))     -> FOO
     (VALFN 'FOO)             -> ((x) (+ x 2))
     (FINDFN (VALFN 'FOO))    -> FOO
```

(REMFN <symb>) [SUBR à 1 argument]

détruit la définition de fonction associée au symbole <symb>. REMFN retourne <symb> en valeur.

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

(MAKEDEF <symb> <ftyp> <fval>) [SUBR à 3 arguments]

fabrique la définition de la fonction de nom <symb> en utilisant les composants <ftyp> et <fval> sous la forme d'une définition (c'est-à-dire sous la forme de l'appel d'une des fonctions DE, DF, DM, DMD ou DS).

MAKEDEF peut être défini en Lisp de la manière suivante :

```
(DE MAKEDEF (x typefn valfn)
    (SELECTQ (TYPEFN x)
        ((SUBR0 SUBR1 SUBR2 SUBR3 NSUBR FSUBR MSUBR DMSUBR)
           (LIST 'DS x typefn valfn))
        (EXPR   (MCONS 'DE x valfn))
        (FEXPR  (MCONS 'DF x valfn))
        (MACRO  (MCONS 'DM x valfn))
        (DMACRO (MCONS 'DMD x valfn))
        (T ())))
```

```
ex : (MAKEDEF 'F 'EXPR '((x) (+ x x)))   -> (DE F (x) (+ x x))
```

(GETDEF <symb>) [SUBR à 1 argument]

permet de retourner la définition de la fonction associée au symbole <symb> sous la forme de sa définition (c'est-à-dire sous la forme de l'appel d'une des fonctions DE, DF, DM, DMD ou DS).

GETDEF peut être défini en Lisp de la manière suivante :

```
(DE GETDEF (x)
    (IF (SYMBOLP x)
        (MAKEDEF x (TYPEFN x) (VALFN x))
        (ERROR 'GETDEF 'ERRSYM x)))

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

(REVERT <symb>) [SUBR à 1 argument]

permet de retrouver l'ancienne définition associée au symbole <symb>, s'il en avait une précédemment sauvée par une fonction de définition statique (voir ces fonctions).

REVERT peut être défini en Lisp de la manière suivante :

```
(DE REVERT (symb)
    (LET ((oldef (GETPROP symb '#:SYSTEM:PREVIOUS-DEF)))
         (WHEN oldef (EVAL oldef))))
```

(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 ((ftype (TYPEFN at2)) (fval (VALFN at2)))
         (IF ftype
             (SETFN at1 ftype fval)
             (ERROR 'SYNONYM 'ERRUDF at2))))

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

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

cette fonction est la forme FSUBR (c'est-à-dire qui n'évalue pas ses arguments) de la fonction précédente.

SYNONYMQ peut être défini en Lisp de la manière suivante :

```
(DF SYNONYMQ (sym1 sym2)
    (SYNONYM sym1 sym2))
```

ou bien

```
(DMD SYNONYMQ (sym1 sym2)
     (LIST 'SYNONYM sym1 sym2))

ex : (SYNONYMQ FOO BAR)  est équivalent à :
     (SYNONYM 'FOO 'BAR)
```

3.11.5 L'accès aux champs spéciaux des symboles

(OBJVAL <symb> <s>) [SUBR à 1 ou 2 arguments]

permet de lire le champ O-VAL du symbole <symb>. Si le 2ème argument <s> est fourni, il devient la nouvelle valeur de ce champ. Cette fonction ne peut pas être décrite facilement en Lisp.

```
ex : (OBJVAL 'GEE)          -> ()
     (OBJVAL 'GEE 'HAUGH)   -> HAUGH
     (OBJVAL 'GEE)          -> HAUGH
```

(PACKAGECELL <symb> <pkgc>) [SUBR à 1 ou 2 arguments]

si le 2ème argument n'est pas fourni cette fonction permet de lire le champ package (PKGC) du symbole <symb>. Si le 2ème argument <pkgc> est fourni, il devient le nouveau package du symbole. Cette fonction ne peut pas être décrite facilement en Lisp.

```
ex : (DEFVAR x '#:sator:arepo:tenet:opera:rotas)          -> x
     (PACKAGECELL x)              -> #:sator:arepo:tenet:opera
     (PACKAGECELL (PACKAGECELL x))      -> #:sator:arepo:tenet
     (PACKAGECELL (PACKACELL (PACKAGECELL x)) -> #:sator:arepo
     (PACKAGECELL x '#:en:to:pan)              -> #:en:to:pan
     x                                  -> #:en:to:pan:rotas
```

(GETFN1 <pkgc> <symb>) [SUBR à 2 arguments]

retourne le symbole **#:pkgc:sym** si le symbole <symb> possède une définition de fonction dans le package <pkgc>.

GETFN1 peut être défini en Lisp de la manière suivante :

```
(DE GETFN1 (pkgc symb)
    (LET ((nom (SYMBOL pkgc symb)))
         (IF (TYPEFN nom)
             nom
             ())))
```

```
ex : (GETFN1 () 'CAR)       -> CAR
     (DE #:FOO:BAR ())      -> #:FOO:BAR
     (GETFN1 'FOO 'BAR)     -> #:FOO:BAR
     (GETFN1 'GEE 'BAR)     -> ()
```

(GETFN <pkgc> <symb> <lastpkgc>) [SUBR à 2 ou 3 arguments]

cherche si un symbole de nom <symb> possède une définition de fonction dans le package <pkgc> puis, s'il n'en existe pas, dans ses packages pères, jusqu'au package de nom <lastpkgc> exclu ou bien jusqu'en haut de la hiérarchie des packages, c'est-à-dire jusqu'au package global (), si <lastpkgc> n'est pas spécifié. GETFN retourne le nom de cette fonction si elle existe (qui peut servir de 1er argument aux fonctions APPLY ou FUNCALL) ou () si elle n'existe pas. Cette fonction est utilisée à l'intérieur de l'interprète pour gérer les interruptions programmables, le lancement des #-macros et la gestion du terminal virtuel.

GETFN peut être défini en Lisp de la manière suivante :

```
(DE GETFN (pkgc symb . lastpkgc)
    (LET ((nom (SYMBOL pkgc symb)))
         (COND ((TYPEFN nom) nom)
               ((NULL pkgc) ())
               ((AND (CONSP lastpkgc)
                     (EQ (PACKAGECELL pkgc)
                         (CAR lastpkgc)))
                ())
               (T (GETFN (PACKAGECELL pkgc)
                         symb lastpkgc)))))
```

```
ex : ; définissons quelques fonctions :"
     (DE FOO ())                          -> FOO
     (DE #:BAR:FOO ())                    -> #:BAR:FOO
     (DE #:BAR:GEE:BUZ:FOO ())            -> #:BAR:GEE:BUZ:FOO

     (GETFN '#:BAR:GEE:BUZ 'FOO)          -> #:BAR:GEE:BUZ:FOO
     (GETFN '#:BAR:GEE 'FOO)              -> #:BAR:FOO
     (GETFN 'BAR 'FOO)                    -> #:BAR:FOO
     (GETFN () 'FOO)                      -> FOO
     (GETFN '#:POTOP:TERAZ 'FOO)          -> FOO
     (GETFN '#:BAR:GEE 'FOO ())           -> #:BAR:FOO
     (GETFN 'BAR 'FOO ())                 -> #:BAR:FOO
     (GETFN 'GEE 'FOO ())                 -> ()
     (GETFN '#:BAR:GEE:BUZ 'FOO 'BAR)     -> #:BAR:GEE:BUZ:FOO
     (GETFN '#:BAR:GEE 'FOO 'BAR)         -> ()
```

3.11.6 Les fonctions de création des symboles

Les symboles sont gérés au moyen d'une table de hachage commune pour tous les packages. La représentation externe d'un symbole hors du package courant est :

#:package:symbol

voir les fonctions d'entrées/sorties.

(SYMBOL <pkgc> <strg>) [SUBR à 2 arguments]

crée un nouveau symbole dans le package <pkgc> et de nom <strg>. Si l'argument <pkgc> vaut (), le package racine est utilisé. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SYMBOL 'foo "bar")        -> #:foo:bar
     (SYMBOL () "fuu")          -> fuu
```

(CONCAT <str1> ... <strN>) [SUBR à N arguments]

fabrique un nouveau symbole dont le P-NAME est construit en concaténant toutes les chaînes <str1> ... <strN>.

CONCAT peut être défini en Lisp de la manière suivante :

```
(DE CONCAT lpname
    (SYMBOL () (MAPCAN 'PNAME lpname)))

ex : (CONCAT 'foo (1+ 5) () 'bar)     ->   FOO6BAR
     (CONCAT "Foo" nil '#"Bar" -2)    ->   FooBar-2
```

(GENSYM) [SUBR à 0 argument]

retourne à chaque appel un nouveau symbole de type Gxxx dans lequel G est une chaîne (conservée dans la variable système #:SYSTEM:GENSYM-STRING) et xxx est un nombre incrémenté de 1 à chaque appel (ce nombre est le compteur de GENSYM et est conservé dans la variable système #:SYSTEM:GENSYM-COUNTER); G et xxx valent "g" et 100 au départ de l'interprète.

GENSYM peut être défini en Lisp de la manière suivante :

```
(DEFVAR #:SYSTEM:GENSYM-STRING "g")
(DEFVAR #:SYSTEM:GENSYM-COUNTER 100)

(DE GENSYM ()
    (CONCAT #:SYSTEM:GENSYM-STRING
            (INCR #:SYSTEM:GENSYM-COUNTER)))

ex : (GENSYM)                        ->  g101
     (GENSYM)                        ->  g102
     (GENSYM)                        ->  g103
     (LET ((#:SYSTEM:GENSYM-STRING "Etiq")) (GENSYM))
                                     ->  Etiq104
```

3.11.7 Les fonctions de gestion des symboles

(OBLIST <pkgc> <symb>) [SUBR à 0, 1 ou 2 arguments]

si aucun argument n'est fourni, 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 prédéfinies. Cette liste étant très longue (souvent de l'ordre de 2000 symboles) il est conseillé d'utiliser les fonctions MAPOBLIST, MAPCOBLIST ou MAPLOBLIST pour accéder successivement à tous ces symboles. Les deux arguments optionnels permettent de filtrer l'OBLIST. Si le premier argument est fourni <pkgc>, ce doit être un symbole qui désigne le package dont on veut extraire les symboles. Si le deuxième argument est fourni <symb>, ce doit être un symbole qui désigne le nom des symboles à extraire (qui peuvent être dans des packages différents). Un appel de type **(OBLIST <pkgc> <symb>)** est donc un moyen sûr de tester la présence d'un symbole dans un certain package sans le créer s'il n'y existait pas. Il n'est pas possible de définir simplement cette fonction en Lisp car il faut accéder aux propriétés naturelles de type A-LINK et P-NAME de chacun des symboles, ce qui n'est réalisable qu'avec les fonctions LOC et MEMORY.

```
ex : (OBLIST)  ->

     (ASSOC SIN CADADR FUNCALL TYPECN EQN TRACEND PLIST
     #:SYSTEM:VAR VAR NOM SYMBOLP GET ADDADR LOAD LETVQ
```

```
CURREAD FALSE PLENGTH  CDDADR FILEOUT PAIRLIS EX*
                       .....
                       .....
                       .....

LIBLOAD #:SYSTEM:LENGTH-FLOAT QUO INBUF UNTILEXIT XDEF)
(OBLIST 'SHARP)  ->
(#:SHARP:" #:SHARP:VALUE #:SHARP:$ #:SHARP:% #:SHARP:(
#:SHARP:+ #:SHARP:- #:SHARP:. #:SHARP:/ #:SHARP::
#:SHARP:[ #:SHARP:\ #:SHARP:^ #:SHARP:|)
```

(LHOBLIST <strg>) [SUBR à 1 argument]

retourne la liste de tous les symboles qui contiennent <strg> comme sous-chaîne dans leur nom. Cette fonction permet donc de filtrer les symboles de l'OBLIST; très utile pour retrouver l'orthographe exacte d'un nom de fonction.

LHOBLIST peut être défini en Lisp de la manière suivante :

```
(DE LHOBLIST (pname)
    (MAPLOBLIST (LAMBDA (symb) (INDEX pname symb 0))))
```

```
ex : (LHOBLIST "apc")      ->
        (MAPC MAPCON MAPCAN MAPCAR MAPCOBLIST)
     (LHOBLIST "string")   ->
        (STRING DUPLSTRING MAKESTRING FILLSTRING EQSTRING
        PRINSTRING SPANSTRING READSTRING BLTSTRING SUBSTRING
        #:SYSTEM:GENSYM-STRING SCANSTRING STRINGP)
```

(BOBLIST <n>) [SUBR à 0 ou 1 argument]

si l'argument <n> n'est pas présent, retourne la liste des *buckets* de la table globale de hachage des symboles du système. Si l'argument est présent, BOBLIST retourne le bucket de numéro donné. Cette fonction ne peut pas être décrite simplement en Lisp.

; pour obtenir la taille (le nombre de buckets) de la table de hachage :

```
(LENGTH (BOBLIST))
```

; pour retourner la liste de toutes les tailles des buckets :

```
(MAPCAR 'LENGTH (BOBLIST))
```

; pour retourner la taille du plus grand bucket :

```
(APPLY 'MAX (MAPCAR 'LENGTH (BOBLIST)))
```

; pour retourner tout le bucket dans lequel se trouve un symbole :

```
(BOBLIST (HASH <symb>))
```

(REMOB <symb>) [SUBR à 1 argument]

enlève toutes les valeurs des différentes propriétés naturelles du symbole <symb> de telle sorte que le symbole <symb> pourra être détruit à la prochaine récupération mémoire si aucun autre pointeur sur ce symbole n'existe.

REMOB peut être défini en Lisp de la manière suivante :

```
(DE REMOB (symb)
    (MAKUNBOUND symb)
    (PLIST symb ())
    (REMFN symb)
    (OBJVAL symb ())
    symb)
```

Pour récupérer tous les symboles d'un package faites :

```
(MAPC 'REMOB (OBLIST 'pkgc))
```

Enfin les fonctions MAPOBLIST, MAPCOBLIST et MAPLOBLIST sont décrites dans la section des fonctions d'application.

3.12 Les Fonctions sur les Chaînes de Caractères

Une chaîne de caractères est une collection de caractères accessibles par leur index (numéro). Les index des chaînes de caractères débutent à 0. Chaque chaîne de caractères possède également un type symbolique qui lui est propre. Par défaut ce type est STRING mais il peut être changé au moyen de la fonction TYPESTRING.

La représentation externe d'une chaîne de caractères de type STRING est la suivante :

```
"xxxxxxxxx"
```

La représentation externe d'une chaîne de caractères d'un autre type est la suivante :

```
#:type:"xxxxxxxxx"
```

Pour toutes ces fonctions l'argument <strg> doit être de type chaîne de caractères. S'il ne l'est pas il est automatiquement converti en utilisant la fonction STRING (sauf pour les fonctions SLEN, SREF et SSET).
Si la conversion n'est pas possible, l'erreur ERRNSA se déclenche dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas une chaîne : <s>      ou bien
** <fn> : string expected : <s>
```

dans lequel l'objet incriminé <s> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur.

L'accès à un caractère de la chaîne en dehors des limites du vecteur déclenche l'erreur ERROOB dont le libellé par défaut est :

```
** <fn> : argument hors limites : <s>      ou bien
** <fn> : out of bounds : <s>
```

dans lequel l'indice incriminé <s> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur.

Le prédicat de test du type chaîne de caractères, STRINGP, est décrit dans la section des prédicats de base.

Pour la description en Lisp de ces fonctions nous utiliserons la fonction de base PNAME qui permet de passer de la représentation interne d'un P-NAME à sa

représentation sous forme de liste de codes internes. Ces descriptions ne reflètent pas la véritable implantation de ces fonctions qui n'utilisent évidemment aucune cellule de liste.

3.12.1 Les fonctions de manipulation de base

Ces fonctions sont les primitives de base sur les chaînes de caractères. Elles permettent à priori de définir toutes les autres fonctions sur les chaînes de caractères.

(SLEN <strg>) [SUBR à 1 argument]

retourne la longueur de la chaîne de caractères <strg>. Le test de type n'a plus lieu après compilation en mode "ouvert", SLEN se compilant au moyen d'une seule instruction LLM3 pour des raisons d'efficacité. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SLEN "abc")              -> 3
     (SLEN "")                 -> 0
     (SLEN t)                  ->
                              ** SLEN : ERRNSA : t
```

(SREF <strg> <n>) [SUBR à 2 arguments]

retourne le <n>ième caractère de la chaîne <strg>. Si <n> dépasse les bornes de la chaîne <strg>, l'erreur ERROOB se déclenche. Ce test de débordement ainsi que le test de type n'a cependant plus lieu après compilation en mode "ouvert", SREF se compilant au moyen d'une seule instruction LLM3 pour des raisons d'efficacité. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ x "abcdef")         -> "abcdef"
     (SREF x 0)                -> #/a
     (SREF x 5)                -> #/f
     (SREF t 1)                ->
                              ** SREF : ERRSTR : t
     (SREF x -2)               ->
                              ** SREF : ERROOB : -2
     (SREF x 6)                ->
                              ** SREF : ERROOB : 6
```

(SSET <strg> <n> <cn>) [SUBR à 3 arguments]

modifie la valeur du <n>ième caractère de la chaîne <strg>. au moyen du caractère <cn>. Si <n> dépasse les bornes de la chaîne <strg> l'erreur ERROOB se déclenche. Le test de débordement ainsi que le test de type n'est cependant plus réalisé après compilation en mode "ouvert", SSET se compilant au moyen d'une seule instruction LLM3 pour des raisons d'efficacité. SSET retourne la nouvelle valeur du caractère de la chaîne (i.e. <cn>). Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ x "abcdef")         -> "abcdef"
     (SSET x 1 #/y)            -> #/y
     (SSET x 3 #/z)            -> #/z
     x                         -> "ayczef"
     (SSET x -3)               ->
```

```
                                    ** SSET : ERROOB : -3
(SSET x 6)                          ->
                                    ** SSET : ERROOB : 6
```

(TYPESTRING <strg> <symb>) [SUBR à 1 ou 2 arguments]

Le deuxième argument, s'il est fourni et s'il est symbolique, devient le nouveau type de la chaîne de caractères <strg. TYPESTRING retourne, après modification eventuelle, le type de la chaîne <strg>. La fonction TYPE-OF sur une chaîne retourne la valeur de TYPESTRING appliquée à cette chaîne. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ s "abc")                 -> "abc"
     (TYPESTRING s)                 -> STRING
     (TYPE-OF s)                    -> STRING
     (TYPESTRING s 'foo)            -> FOO
     (TYPESTRING v)                 -> FOO
     s                              -> #:foo"abc"
     (TYPE-OF s)                    -> FOO
     (TYPESTRING t)                 ->
                                 ** TYPESTRING : ERRNSA : t
     (TYPESTRING v "bar")           ->
                                ** TYPESTRING : ERRSYMB : bar
```

(EXCHSTRING <strg1> <strg2>) [SUBR à 2 arguments]

Cette fonction, très obscure, permet d'échanger physiquement les valeurs et le type des chaînes <strg1> et <strg2>. EXCHSTRING retourne la chaîne <strg1> modifiée. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ v "abc" w "de")          -> "de"
     (TYPESTRING v 'foo)            -> foo
     (SETQ y v z w)                 -> "de"
     (EXCHSTRING v w)               -> "de"
     v                              -> #:foo:"abc"
     w                              -> "de"
     y                              -> #:foo:"abc"
     z                              -> "de"
```

3.12.2 Les conversions des chaînes de caractères

(STRING <s>) [SUBR à 1 argument]

convertit l'argument <s> en une chaîne de caractères. La suite des caractères de cette chaîne est déterminée par les règles :
- si <s> est la liste vide (c'est-à-dire ()), cette suite de caractères est vide.
- si <s> est un symbole, il s'agit de la suite de caractères de son nom externe (P-NAME).
- si <s> est un nombre, il s'agit de la suite de caractères de sa représentation externe dans la base de conversion de sortie courante (OBASE).
- si <s> est une chaîne de caractères, il s'agit de cette suite de caractères.
- si <s> est une liste, elle est supposée être une liste de codes internes dont la taille ne doit pas excéder 255.

Si l'une de ces conditions n'est pas respectée, l'erreur ERRNSA se déclenche.

```
ex : (STRING ())              -> ""
     (STRING '||)             -> ""
     (STRING '|Foo|)          -> "Foo"
     (STRING "Bar")           -> "Bar"
     (STRING -345)            -> "-345"
     (STRING 2.3)             -> "2.3"
     (STRING '#"AbC")         -> "AbC"
     (STRING (MAKELIST 5 #/X))    -> "XXXXX"
```

(PNAME <strg>) [SUBR à 1 argument]

retourne la chaîne <strg> sous la forme d'un liste de codes internes. Cette fonction, dépendante des représentations internes Lisp, ne peut pas être décrite simplement.

```
ex : (PNAME ())           -> ()
     (PNAME nil)          -> ()
     (PNAME 'nil)         -> (110 105 108)
     (PNAME 'foobar)      -> (102 111 111 98 97 114)
     (PNAME -123)         -> (45 49 50 51)
     (PNAME "abcdef")     -> (97 98 99 100 101 102)
     (PNAME '#"abc")      -> (97 98 99)
```

(PLENGTH <strg>) [SUBR à 1 argument]
(SLENGTH <strg>) [SUBR à 1 argument]

retourne en valeur le nombre de caractères de la chaîne <strg>. Il n'y a aucune différence entre les fonctions PLENGTH et SLENGTH.

PLENGTH peut être défini en Lisp de la manière suivante :

```
(DE PLENGTH (pname)
    (LENGTH (PNAME pname)))
```

```
ex : (PLENGTH ())         -> 0
     (PLENGTH nil)        -> 0
     (PLENGTH 'nil)       -> 3
     (PLENGTH 'foobar)    -> 6
     (PLENGTH -123)       -> 4
     (PLENGTH "Sobota")   -> 6
     (PLENGTH '#"Dzien") -> 5
```

(HASH <strg>) [SUBR à 1 argument]

retourne la clef de l'algorithme interne de hachage de la chaîne <strg>. Les performances de l'algorithme de hachage peuvent être mesurées en utilisant la fonction BOBLIST. La description qui suit n'est donnée qu'à titre explicatif, l'implantation réelle de cette fonction est beaucoup plus performante.

HASH peut être défini en Lisp de la manière suivante :

```
(DE HASH (strg)
    (REM (LOGAND (HASHAUX (PNAME strg)) #$7FFF)
         (LENGTH (BOBLIST))))

(DE HASHAUX (pn)
    (IF (<= (LENGTH pn) 6)
        (HASHCOUNT (LENGTH pn) pn (LENGTH pn))
```

```
            (HASHCOUNT 6
                       (LASTN 6 pn)
                       (HASHCOUNT 6
                                  (FIRSTN 6 pn)
                                  (LENGTH pn)))))

(DE HASHCOUNT (count pn val)
    (SETQ pn (REVERSE pn))
    (REPEAT count
            (SETQ val (ADD (LOGSHIFT val 1) (NEXTL pn))))
    val)

ex : (HASH ())                    -> 0
     (HASH 'nil)                  -> 250
     (HASH 'foobar)               -> 35
     (HASH -123)                  -> 26
     (HASH "abcdef")              -> 178
     (HASH "galamantdelareine")   -> 100
     (HASH '#"abcd")              -> 238
```

3.12.3 Les comparaisons des chaînes de caractères

(EQSTRING <str1> <str2>) [SUBR à 2 arguments]

teste si les 2 arguments de type chaîne de caractères sont égaux. S'ils le sont EQSTRING retourne <str1> sinon EQSTRING retourne ().

EQSTRING peut être défini en Lisp de la manière suivante :

```
(DE EQSTRING (str1 str2)
    (AND (= (SLENGTH str1) (SLENGTH str2))
         (EQ (TYPESTRING str1) (TYPESTRING str2))
         (EQUAL (PNAME str1) (PNAME str2))))

ex : (EQSTRING "foo" "bar")              -> ()
     (EQSTRING "foo" "foo")              -> "foo"
     (EQSTRING "Foo" "foo")              -> ()
     (EQSTRING (string (1+ 11)) (catenate 1 2))   -> "12"
```

(ALPHALESSP <str1> <str2>) [SUBR à 2 arguments]

retourne T si le P-NAME <str1> est inférieur ou égal (lexicographiquement) au P-NAME <str2>, sinon retourne (). Cette fonction est utilisée pour réaliser des tris alphabétiques (voir la fonction SORTL).

ALPHALESSP peut être défini en Lisp de la manière suivante :

```
(DE ALPHALESSP (A1 A2)
    (LETN ALPHAL1 ((L1 (PNAME A1)) (L2 (PNAME A2)))
          (COND ((NULL L1) T)
                ((NULL L2) ())
                ((= (CAR L1) (CAR L2))
                    (ALPHAL1 (CDR L1) (CDR L2)))
                (T (IF (< (CAR L1) (CAR L2)) T ()))))))

ex : (ALPHALESSP 'A 'A)        -> T
     (ALPHALESSP 'B 'A)        -> ()
```

```
(ALPHALESSP 'A 'B)        -> T
(ALPHALESSP 'ZZZ 'ZZZZ)   -> T
```

3.12.4 Les fonctions de création de chaînes de caractères

(CATENATE <str1> ... <strN>) [SUBR à N arguments]

fabrique une nouvelle chaîne résultant de la concaténation des différentes chaînes <str1> ... <strN>. Cette fonction doit être distinguée de la fonction CONCAT qui, elle, crée un symbole.

CATENATE peut être défini en Lisp de la manière suivante :

```
(DE CATENATE l
    (STRING (MAPCAN 'PNAME l)))
```

```
ex : (CATENATE "foo" () 'bar (1+ 10) '#"()")   -> "foobar11()"
     (CATENATE -34 0 12)                       -> "-34012"
     (CATENATE "" || () nil)                   -> ""
```

(MAKESTRING <n> <cn>) [SUBR à 2 arguments]

retourne une chaîne de <n> caractères tous égaux au code interne <cn>.

MAKESTRING peut être défini en Lisp de la manière suivante :

```
(DE MAKESTRING (n cn)
    (STRING (MAKELIST n cn)))
```

```
ex : (MAKESTRING 0 #/a)          -> ""
     (MAKESTRING 4 #/a)          -> "aaaa"
     (MAKESTRING -1 #/a)         -> ""
```

(SUBSTRING <strg> <n1> <n2>) [SUBR à 2 ou 3 arguments]

retourne une copie de la sous-chaîne de <strg> à partir de la position <n1> (comptée à partir de 0) composée de <n2> caractères. Si ce dernier argument n'est pas fourni la sous-chaîne retournée se termine à la fin de la chaîne argument <strg>.

SUBSTRING peut être défini en Lisp de la manière suivante :

```
(DE SUBSTRING (strg n1 . n2)
    (STRING (FIRSTN (- (IF (CONSP n2)
                           (CAR n2)
                           (PLENGTH strg))
                       n1)
                    (NTHCDR n1 (PNAME strg)))))
```

```
ex : (SUBSTRING "abcde" 0 3)        -> "abc"
     (SUBSTRING "abcde" 1 2)        -> "bc"
     (SUBSTRING "abcde" 2)          -> "cde"
     (SUBSTRING "abcde" 9 2)        -> ""
```

(DUPLSTRING <n> <strg>) [SUBR à 2 arguments]

fabrique une chaîne résultant de la duplication <n> fois de la chaîne <strg>.

DUPLSTRING peut être défini en Lisp de la manière suivante :

```
(DE DUPLSTRING (n strg)
    (IF (<= n 0)
        ""
        (CATENATE strg (DUPLSTRING (1- n) strg))))
```

```
ex : (DUPLSTRING 3 "ab")      -> "abababab"
     (DUPLSTRING 3 "")        -> ""
     (DUPLSTRING 1 "abc")     -> "abc"
     (DUPLSTRING 0 "abc")     -> ""
```

3.12.5 Les fonctions d'accès aux caractères des chaînes

Ces fonctions permettent de manipuler individuellement les caractères des chaînes.

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

retourne en valeur la position du caractère de code interne <cn> dans la chaîne <strg>. La position du premier caractère est 0. Si le code <cn> n'existe pas dans la chaîne, CHRPOS retourne (). CHRPOS peut donc être utilisé comme prédicat testant la présence d'un caractère dans une chaîne.

CHRPOS peut être défini en Lisp de la manière suivante :

```
(DE CHRPOS (c pname)
    (LETN CHRP ((lcn (PNAME pname)) (n 0))
          (COND ((NULL lcn) ())
                ((EQ (CAR lcn) c) n)
                (T (CHRP (CDR lcn) (1+ n))))))
```

```
ex : (CHRPOS #/Y "OTYoty")                -> 2
     (CHRPOS #/N "OTY")                   -> ()
     (CHRPOS #/D '#"0123456789ABCDEF")    -> 13
```

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

retourne le code interne du <n>ième caractère de la chaîne <strg> et s'il n'existe pas retourne la valeur (). La position (le rang) du premier caractère est 0 (et non 1).

CHRNTH peut être défini en Lisp de la manière suivante :

```
(DE CHRNTH (n string)
    (NTH n (PNAME string)))
```

```
ex : (CHRNTH 0 "FOO")                  -> 70  c'est-à-dire #/F
     (CHRNTH 10 "0123456789ABCDEF")    -> 65  c'est-à-dire #/A
```

(CHRSET <n> <strg> <val>) [SUBR à 3 arguments]

permet de modifier le <n>ième caractère de la chaîne <strg> avec la valeur <val>. La position (le rang) du premier caractère est 0. Cette fonction de modification physique ne peut pas se décrire en Lisp.

```
ex : (SETQ x "abc")              -> "abc"
```

```
(CHRSET 0 x #/A)          -> #/A
x                         -> "Abc"
```

3.12.6 Les fonctions de modification physique des chaînes

Ces fonctions de modification physique peuvent être décrites en Lisp au moyen de la fonction CHRSET.

(BLTSTRING <str1> <n1> <str2> <n2> <n3>) [SUBR à 4 ou 5 args]

transfère dans la chaîne <str1> à partir de la position <n1> (comptée à partir de 0) <n3> caractères de la chaîne <str2> à partir de la position <n2>. Si l'argument <n3> n'est pas spécifié, toute la chaîne <strg2> à partir de <n2> sera transférée. La chaîne <str1> n'est jamais étendue. BLTSTRING retourne la chaîne <str1> modifiée. <str1> et <str2> peuvent être la même chaîne physique. Il est donc possible d'écraser le contenu d'une chaîne par elle-même dans n'importe quelle position.

BLTSTRING peut être défini en Lisp de la manière suivante :

```
(DE BLTSTRING (str1 n1 str2 n2 n3)
    ; ne traite pas le cas d'une chaîne partagée
    (REPEAT (MIN (IF (CONSP n3)
                     (CAR n3)
                     (- (SLENGTH str2) n2))
                 (- (SLENGTH str1) n1))
            (CHRSET n1 str1 (CHRNTH n2 str2))
            (INCR n1)
            (INCR n2))
    str1)
```

```
ex : (BLTSTRING "foobar" 1 "xyz" 2 1)    -> "fzobar"
     (BLTSTRING "foobar" 1 "xyz" 0)      -> "fxyzar"
     (BLTSTRING "foobar" 1 "toto" 0 6)   -> "ftotor"
     (BLTSTRING "foobar" 3 "toto" 0 8)   -> "footot"
     (SETQ strt "abcdefghij")            -> "abcdefghij"
     (BLTSTRING strt 1 strt 3 4)         -> "adefgfghij"
     (BLTSTRING strt 6 strt 0 2)         -> "adefgfadij"
     (BLTSTRING strt 0 strt 4 4)         -> "gfadgfadij"
```

(FILLSTRING <strg> <n1> <cn> <n2>) [SUBR à 3 ou 4 arguments]

modifie la chaîne <strg> à partir de la position <n1> (comptée à partir de 0) avec <n2> caractères de code interne <cn>. Si l'argument <n2> n'est pas spécifié ou si la somme <n1+n2> est plus grande que la longueur de la chaîne, la modification ne portera que jusqu'à la fin de la chaîne. La chaîne <strg> n'est jamais étendue. FILLSTRING retourne la chaîne <strg> modifiée.

FILLSTRING peut être défini en Lisp de la manière suivante :

```
(DE FILLSTRING (strg n1 cn . n2)
    (REPEAT (IF (CONSP n2)
                (MIN (CAR n2) (- (SLENGTH strg) n1)
                (- (SLENGTH strg) n1))
            (CHRSET n1 strg cn) (INCR n1)))
```

```
ex : (FILLSTRING "foobar" 1 #/X 2)    -> "fXXbar"
     (FILLSTRING "foobar" 0 #/Y 3)    -> "YYYbar"
     (FILLSTRING "foobar" 2 #/X)      -> "foXXXX"
     (FILLSTRING "foobar" 2 #/X 10)   -> "foXXXX"
```

3.12.7 Les fonctions de recherche sur les chaînes de caractères

(INDEX <str1> <str2> <n>) [SUBR à 3 arguments]

cherche si l'argument <str1> est un sous-ensemble de l'argument <str2> et ce à partir de la position <n>. Si c'est le cas, INDEX retourne le rang (compté à partir de 0) du sous-ensemble de <str2> égal à <str1>, dans le cas contraire INDEX retourne ().

INDEX peut être défini en Lisp de la manière suivante :

```
(DE INDEX (str1 str2 . n)
    (LETN INXAUX1 ((p1 (PNAME str1))
                   (p2 (NTHCDR (OR (CAR n) 0)
                               (PNAME str2)))
                   (n n))
          (COND ((< (LENGTH p2) (LENGTH p1)) ())
                ((<> (CAR p1) (CAR p2))
                   (INXAUX1 p1 (CDR p2) (1+ n)))
                (T (LETN INXAUX2 ((pp1 (CDR p1))
                                  (pp2 (CDR p2)))
                         (COND ((NULL pp1) n)
                               ((= (CAR pp1) (CAR pp2))
                                  (INXAUX2 (CDR pp1)
                                           (CDR pp2)))
                               (T (INXAUX1 p1 (CDR p2)
                                           (1+ n)))))))))
```

```
ex : (INDEX "foo" "foobar")          -> 0
     (INDEX "bar" "foobar")          -> 3
     (INDEX "foo" "xfoobar")         -> 1
     (INDEX "foo" "xfoobar" 2)       -> ()
     (INDEX "foo" "xfoobar" 1)       -> 1
     (INDEX "foo" "" 0)              -> ()
     (INDEX "" "foo" 0)              -> 0
```

(SCANSTRING <str1> <str2> <n>) [SUBR à 2 ou 3 arguments]

recherche dans la chaîne <str1> à partir de la position <n> (comptée à partir de 0 (valeur par défaut de cet argument)) un caractère faisant partie de la chaîne <str2>. SCANSTRING retourne le rang (compté à partir de 0) du premier caractère de <str1> qui existe dans <str2>, ou bien () si aucun caractère de <str1> n'existe dans <str2>.

```
ex : (SCANSTRING "abcd" "sbe")       -> 1
     (SCANSTRING "abcd" "efg")       -> ()
     (SCANSTRING "abcd" ".a" 1)      -> ()
     (SCANSTRING "abc" "defghc")     -> 2
     (SCANSTRING "foo" "")           -> ()
```

(SPANSTRING <str1> <str2> <n>) [SUBR à 2 ou 3 arguments]

recherche dans la chaîne <str1> à partir de la position <n> (comptée à partir de 0 (valeur par défaut de cet argument)) un caractère ne faisant pas partie de la chaîne <str2>. SPANSTRING retourne le rang (compté à partir de 0) du premier caractère de <str1> qui n'existe pas dans <str2>, ou bien () si tous les caractères de <str1> existent dans <str2>.

```
ex : (SPANSTRING "abcb" "ab")          -> 2
     (SPANSTRING "abcd" "aabbccdd")    -> ()
     (SPANSTRING "abcd" "bcd" 1)       -> ()
     (SPANSTRING "foo" "")             -> 0
```

3.13 Les Fonctions sur les Caractères

Ces fonctions vont manipuler des codes internes des caractères dénotés : <cn>. Le codage utilisé par Le_Lisp est en général le code ASCII.

(ASCII <cn>) [SUBR à 1 argument]

retourne le caractère de code ASCII <cn> (modulo 256). Un caractère étant un objet Lisp dont le P-NAME n'a qu'1 caractère..DebLL ex : (ASCII 67) -> C
(1+ (ASCII 49)) -> 2

(CASCII <ch>) [SUBR à 1 argument]

retourne le code ASCII du caractère <ch>.

```
ex : (CASCII 'c)  -> 99
     (CASCII 1)   -> 49
```

(UPPERCASE <cn>) [SUBR à 1 argument]

si <cn> représente le code interne d'un caractère minuscule, 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.

(ASCIIP <cn>) [SUBR à 1 argument]

teste si <cn> est bien un code interne, c'est-à-dire un petit entier dans l'intervalle [0 256[. Retourne le code si le test est vérifié et () dans le cas contraire.

(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 () dans le cas contraire.

(LETTERP <cn>) [SUBR à 1 argument]

teste si <cn> est le code interne d'une lettre. LETTERP retourne le code si le test est vérifié et () dans le cas contraire.

3.14 Les Fonctions sur les Vecteurs

Le_Lisp possède un dernier type d'objet, les vecteurs de S-expressions. Un vecteur est une collection d'objets Lisp accessibles par leur index (numéro). Les index des vecteurs débutent à 0. Chaque vecteur possède également un type symbolique qui lui est propre. Par défaut ce type est VECTOR mais il peut être changé au moyen de la fonction TYPEVECTOR.

La représentation externe des vecteurs de type VECTOR est la suivante :

```
#[s1 s2 ... sN]
```

La représentation externe d'un vecteur d'un autre type est la suivante :

```
#:type:#[s1 s2 ... sN]
```

Pour toutes les fonctions qui vont suivre, l'argument <vect> doit être de type vecteur sous peine de déclencher l'erreur ERRVEC dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas un vecteur : <s>     ou bien
** <fn> : vector expected : <s>
```

dans lequel l'objet incriminé <s> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur.

L'accès à un élément de vecteur en dehors des limites du vecteur déclenche l'erreur ERROOB dont le libellé par défaut est :

```
** <fn> : argument hors limites : <s>     ou bien
** <fn> : out of bounds : <s>
```

dans lequel l'indice incriminé <s> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur.

Le prédicat de test du type vecteur, VECTORP, est décrit dans la section des prédicats de base.

(MAKEVECTOR <n> <s>) [SUBR à 2 arguments]

fabrique un nouveau vecteur (de type VECTOR) de <n> éléments. Chacun d'eux est initialisé avec la valeur <s>. Si <n> n'est pas un nombre entier positif, les erreurs ERRNIA et ERROOB se déclenchent. MAKEVECTOR retourne le nouveau vecteur ainsi créé. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (MAKEVECTOR 5 ())      -> #[() () () () ()]
     (MAKEVECTOR 3 '(a b)) -> #[(a b) (a b) (a b)]
     (MAKEVECTOR 0 'a)      -> #[]
```

```
(MAKEVECTOR -1 ())      ->
                        ** MAKEVECTOR : ERROOB : -1
(MAKEVECTOR t ())       ->
                        ** MAKEVECTOR : ERRNIA : t
```

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

fabrique un nouveau vecteur (de type VECTOR) de taille N (qui est le nombre d'arguments d'appel de la fonction VECTOR), initialisé avec les différentes valeurs <s1> ... <sN>. VECTOR retourne le nouveau vecteur ainsi créé.

VECTOR peut être défini en Lisp de la manière suivante :

```
(DE VECTOR l
    (LET ((v (MAKEVECTOR (LENGTH l) ())))
         (FOR (i 0 1 (1- (VLENGTH v)))
              (VSET v i (nextl l)))
         v))
```

```
ex :  (VECTOR 0 1 2 3 4)                  ->  #[0 1 2 3 4]
      (VECTOR)                            ->  #[0]
      (APPLY 'VECTOR '(a b c))            ->  #[a b c]
      (VECTOR 1 #[1 2] "Foo" 'A '(B C)) ->
                                  #[1 #[1 2] "Foo" A (B C)]
```

(VLENGTH <vect>) [SUBR à 1 argument]

retourne le nombre d'éléments du vecteur <vect>. Cette fonction ne peut pas être décrite simplement en Lisp. Le test de type n'a plus lieu après compilation en mode "ouvert", VLENGTH se compilant au moyen d'une seule instruction LLM3 pour des raisons d'efficacité.

```
ex : (VLENGTH #[])          ->  0
     (VLENGTH #[1 2 3])     ->  3
     (VLENGTH t)            ->
                               ** VLENGHT : ERRVEC: t
```

(VREF <vect> <n>) [SUBR à 2 arguments]

retourne le <n>ième élément du vecteur <vect>. Si <n> dépasse les bornes du vecteur <vect>, l'erreur ERROOB se déclenche. Ce test de débordement n'a cependant plus lieu après compilation en mode "ouvert", VREF se compilant au moyen d'une seule instruction LLM3 pour des raisons d'efficacité. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ x #[a b c d e f])    ->  #[a b c d e f]
     (VREF x 0)                 ->  a
     (VREF x 5)                 ->  f
     (VREF t 1)                 ->
                               ** VREF : ERRVEC : t
     (VREF x -2)                ->
                               ** VREF : ERROOB : -2
     (VREF x 6)                 ->
                               ** VREF : ERROOB : 6
```

(VSET <vect> <n> <e>) [SUBR à 3 arguments]

modifie la valeur du <n>ième élément du vecteur <vect> avec l'objet <e>. Si <n> dépasse les bornes du vecteur <vect> l'erreur ERROOB se déclenche. Le test de débordement n'est cependant plus réalisé après compilation en mode "ouvert", VSET se compilant au moyen d'une seule instruction LLM3 pour des raisons d'efficacité. VSET retourne la nouvelle valeur de l'élément, c'est-à-dire <e>. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ x #[a b c d e f])    -> #[a b c d e f]
     (VSET x 1 #[x y])          -> #[x y]
     (VSET x 3 '(h i))          -> (h i)
     x                          -> #[a #[x y] c (h i) e f]
     (VSET x -3)                ->
                              ** VSET : ERROOB : -3
     (VSET x 6)                 ->
                               ** VSET : ERROOB : 6
```

(TYPEVECTOR <vect> <symb>) [SUBR à 1 ou 2 arguments]

Le deuxième argument, s'il est fourni et s'il est symbolique, devient le nouveau type du vecteur <vect>. TYPEVECTOR retourne, après modification éventuelle, le type du vecteur <vect>. La fonction TYPE-OF sur un vecteur retourne la valeur de TYPEVECTOR appliquée à ce vecteur. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ v #[a b c])          -> #[a b c]
     (TYPEVECTOR v)             -> VECTOR
     (TYPE-OF v)                -> VECTOR
     (TYPEVECTOR v 'foo)        -> FOO
     (TYPEVECTOR v)             -> FOO
     v                          -> #:foo:#[a b c]
     (TYPE-OF v)                -> FOO
     (TYPEVECTOR t)             ->
                           ** TYPEVECTOR : ERRVEC : t
     (TYPEVECTOR v "bar")       ->
                         ** TYPEVECTOR : ERRSYMB : bar
```

(EQVECTOR <vect1> <vect2>) [SUBR à 2 arguments]

teste si les 2 arguments, de type vecteur, sont égaux, c'est-à-dire s'ils possèdent le même type, le même nombre d'éléments et si tous leurs éléments sont égaux (en utilisant le prédicat EQUAL).

EQVECTOR peut être défini en Lisp de la manière suivante :

```
(DE EQVECTOR (vect1 vect2)
    (COND ((NOT (VECTORP vect1))
           (ERROR 'EQVECTOR 'ERRVEC vect1))
          ((NOT (VECTORP vect2))
           (ERROR 'EQVECTOR 'ERRVEC vect2))
          ((AND (= (VLENGTH vect1) (VLENGTH vect2))
                (EQ (TYPEVECTOR vect1) (TYPEVECTOR vect2)))
           (TAG NO
                (FOR (i 0 1 (1- (VLENGTH vect1)))
                     (UNLESS (EQUAL (VREF vect1 i)
```

```
                                        (VREF vect2 i))
                                    (EXIT NO ()))
                            vect1))
                    (T ()))))

ex : (EQVECTOR #[1 2 3] #[1 2 3])       -> #[1 2 3]
     (EQVECTOR #[1 2 3] #[1 2])         -> ()
     (EQVECTOR #:foo:#[1 2] #[1 2])     -> ()
     (EQVECTOR #:foo:#[1] #:foo:#[1]) ->   #:foo:#[1]
```

(BLTVECTOR <vect1> <n1> <vect2> <n2> <n3>) [SUBR à 5 args]

transfère dans le vecteur <vect1> à partir de la position <n1> (comptée à partir de 0) <n3> éléments du vecteur <vect2> à partir de la position <n2>. Le vecteur <vect1> n'est jamais étendu. BLTVECTOR retourne le vecteur <vect1> modifié. Contrairement à BLTSTRING, BLTVECTOR ne permet pas d'écraser le contenu d'un vecteur par lui-même dans n'importe quelle position : le transfert a toujours lieu de la gauche vers la droite.

BLTVECTOR peut être défini en Lisp de la manière suivante :

```
(DE BLTVECTOR (vect1 n1 vect2 n2 n3)
    ; cette définition ne traite pas des cas limites.
    (REPEAT n3
            (VSET vect1 n1 (VREF vect2 n2))
            (INCR n1)
            (INCR n2))
    vect1)

ex :  (BLTVECTOR #[f o o b a r] 1 #[x y z] 2 1)
        -> #[f z o b a r]
      (BLTVECTOR #[f o o b a r] 1 #[t o t o] 0 6)
        -> #[f t o t o r]
      (BLTVECTOR #[f o o b a r] 3 #[t o t o t a] 0 8)
        -> #[f o o t o t]
```

(FILLVECTOR <vect> <n1> <e> <n2>) [SUBR à 3 ou 4 arguments]

modifie le vecteur <vect> à partir de la position <n1> (comptée à partir de 0) avec <n2> expressions quelconques <e>. Si l'argument <n2> n'est pas spécifié ou si la somme <n1+n2> est plus grande que la longueur du vecteur, la modification ne portera que jusqu'à la fin du vecteur. Le vecteur <vect> n'est jamais étendu. FILLVECTOR retourne le vecteur <vect> modifié.

FILLVECTOR peut être défini en Lisp de la manière suivante :

```
(DE FILLVECTOR (vect n1 cn . n2)
    (REPEAT (IF (CONSP n2)
                (MIN (CAR n2) (- (VLENGTH vect) n1)
                (- (VLENGTH vect) n1))
            (VSET vect n1 e) (INCR n1)))

ex : (FILLVECTOR #[f o o b a r] 1 'x 2)     -> #[f x x b a r]
     (FILLVECTOR #[f o o b a r] 0 'y 3)     -> #[y y y b a r]
     (FILLVECTOR #[f o o b a r] 2 'x)       -> #[f o x x x x]
     (FILLVECTOR #[f o o b a r] 2 'x 10)    -> #[f o x x x x]
```

(EXCHVECTOR <vect1> <vect2>) [SUBR à 2 arguments]

Cette fonction, très obscure, permet d'échanger physiquement les valeurs et le type des vecteurs <vect1> et <vect2>. EXCHVECTOR retourne le vecteur <vect1> modifié. Cette fonction ne peut pas être décrite simplement en Lisp.

```
ex : (SETQ v #[a b c] w #[d e])          ->  #[d e]
     (TYPEVECTOR v 'foo)                 ->  foo
     v                                   ->  #:foo:#[a b c]
     (SETQ y v z w)                      ->  #[d e]
     (EXCHVECTOR v w)                    ->  #[d e]
     v                                   ->  #[d e]
     w                                   ->  #:foo:#[a b c]
     y                                   ->  #[d e]
     z                                   ->  #:foo:#[a b c]
```

Enfin la fonction MAPVECTOR est décrite dans la section sur les fonctions d'application.

3.15 Les Fonctions sur les Tableaux

Il est très aisé de réaliser des tableaux multi-dimensionnels avec les fonctions de la section précédente.

(MAKEARRAY <n1> ... <nN> <s>) [SUBR à N arguments]

permet de construire un tableau de <n1>*<n2>* ... <nN-1>*<nN> éléments, tous initialisés à la valeur <s>.

MAKEARRAY peut être défini en Lisp de la manière suivante :

```
(DE MAKEARRAY (arg1 . args)
    (IF (CONSP args)
        (LET ((result (MAKEVECTOR arg1 ())))
             (FOR (i 0 1 (1- arg1))
                  (VSET result i (APPLY 'MAKEARRAY args)))
             result)
        arg1))
```

(AREF <array> <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur de l'élément du tableau <array> d'indice <n1> ... <nN>. Ces indices sont comptés à partir de 0.

AREF peut être défini en Lisp de la manière suivante :

```
(DMD AREF (inst . args)
     (COND ((NULL args) inst)
           ((ATOM args) (ERROR 'AREF 'ERRWNA args))
           (T `(VREF (AREF ,inst ,@(NREVERSE (CDR (REVERSE args))))
                     ,(CAR (LAST args))))))
```

(ASET <array> <n1> ... <nN> <e>) [SUBR à N arguments]

permet de modifier la valeur de l'élément du tableau <array> d'indice <n1> ... <nN> avec la valeur <e>. Ces indices sont comptés à partir de 0. ASET retourne <e> en valeur.

ASET peut être défini en Lisp de la manière suivante :

```
(DMD ASET (inst . args)
     (COND ((ATOM args) (ERROR 'AREF 'ERRWNA args))
           ((CONSP (CDR args))
              `(VSET (AREF ,inst
                           ,.(NREVERSE (CDDR (REVERSE args))))
                     ,(CADR (REVERSE args))
                     ,(CAR (LAST args))))
           (T inst)))
```

```
ex :  (SETQ x (MAKEARRAY 3 4 0))
          ->  #[#[0 0 0 0] #[0 0 0 0] #[0 0 0 0]]
      (ASET x 1 2 -1)  ->  -1
      (AREF x 1 2)     ->  -1
      x   ->  #[#[0 0 0 0] #[0 0 -1 0] #[0 0 0 0]]
```

CHAPITRE 4

Les Fonctions sur les Nombres

Le_Lisp possède plusieurs types de nombres :

- les nombres à précision fixe
  - entiers sur 16 bits
  - flottants sur 31, 32, 48 ou 64 bits, dépendant des systèmes.
- les nombres à précision arbitraire (sous forme de bibliothèques Le_Lisp).

et il existe cinq types d'arithmétique :
- l'arithmétique générique
- l'arithmétique entière
- l'arithmétique entière étendue
- l'arithmétique flottante
- l'arithmétique mixte

Les fonctions de ce chapitre ne connaissent que les nombres entiers sur 16 bits et les nombres flottants. Le chapitre 10 décrit l'arithmétique rationnelle et complexe qui utilise les types numériques étendus.

Toute division par 0 provoque l'erreur ERR0DV dont le libellé par défaut est :

```
** <fn> : division par 0.      ou bien
** <fn> : zero divide.
```

dans lequel le nom de la fonction qui a provoqué l'erreur <fn> est imprimé.

4.1 L'Arithmétique Générique

Les arguments de ces fonctions peuvent être des nombres entiers, flottants ou bien d'un type numérique étendu. Si l'un des arguments est flottant, le résultat de ces fonctions est en général un nombre flottant. Si tous les arguments sont de type entier, le résultat de ces fonctions est un nombre entier. Si le calcul entier déborde ou si un argument n'est ni un entier ni un flottant, ces fonctions appellent l'interruption GENARITH.

Du fait de sa généralité, cette arithmétique est à conseiller mais elle n'est pas très rapide. Si l'efficacité est essentielle, il est possible d'utiliser les arithmétiques spécialisées sur les entiers ou sur les flottants.

4.1.1 L'interruption GENARITH

GENARITH [Interruption arithmétique]

Cette interruption est appelée si un calcul ne peut être réalisé par une fonction de l'arithmétique générique. Cette interruption va invoquer une fonction de nom :

```
(OR (GETFN (TYPE-OF <arg1>) <fn> ())
    (GETFN #:SYS-PACKAGE:GENARITH <fn> ()))
```

Dans laquelle <fn> est le nom de la fonction générique qui ne peut pas réaliser le calcul et <arg1> la valeur du premier argument. Cette fonction est invoquée avec 1 ou 2 arguments (dépendant de la fonction appelée, voir la section 4.1.7). Si cette fonction n'existe pas l'erreur ERRGEN se produit dont le libellé est :

```
** <fn> : ne peut pas calculer : <larg>     ou bien
** <fn> : can't compute : <larg>
```

dans lequel <fn> est le nom de la fonction incriminée et <larg> la liste des arguments évalués de cet appel. Ce mécanisme permet d'interfacer facilement d'autres arithmétiques totalement programmées en Lisp.

#:SYS-PACKAGE:GENARITH [Variable]

cette variable contient le nom du package dans lequel doivent être recherchées les fonctions génériques s'il n'existe pas de fonction associée au type du premier argument. Par défaut cette valeur est GENARITH.

4.1.2 Les tests de type

Seule la première fonction NUMBERP est susceptible d'appeler l'interruption GENARITH.

(NUMBERP <s>) [SUBR à 1 argument]

teste si <s> est un nombre. NUMBERP retourne le nombre <s> si le test est vérifié et () dans le cas contraire.

```
ex : (NUMBERP 44)       -> 44
     (NUMBERP 3.14)     -> 3.14
```

(FIXP <s>) [SUBR à 1 argument]

retourne <s> si <s> est un nombre entier de précision fixe et () dans le cas contraire.

```
ex : (FIXP 120)       -> 120
     (FIXP 10.256)    -> ()
```

(FLOATP <s>) [SUBR à 1 argument]

retourne <s> si <s> est un nombre flottant de précision fixe et () dans le cas contraire.

```
ex : (FLOATP 120)       -> ()
     (FLOATP 10.256)    -> 10.256
```

4.1.3 Les conversions numériques

Bien que certaines fonctions numériques convertissent leurs arguments, il est possible de réaliser des conversions explicitement au moyen des 3 fonctions :

(TRUNCATE <n>) [SUBR à 1 argument]
(FIX <n>) [SUBR à 1 argument]

si l'argument <n> est un nombre entier, TRUNCATE le retourne. Si <n> est un nombre flottant <n> est converti en entier si c'est possible sinon provoque l'interruption GENARITH. Cette conversion réalise une troncature : le résultat est un entier de même signe que l'argument <n>, de valeur absolue égale au plus grand entier inférieur ou égal à la valeur absolue de l'argument. FIX est un ancien nom gardé par souci de compatibilité.

```
ex : (TRUNCATE 10.)          -> 10
     (TRUNCATE 10.4)         -> 10
     (TRUNCATE 10.6)         -> 10
     (TRUNCATE -10.)         -> -10
     (TRUNCATE -10.4)        -> -10
     (TRUNCATE -10.6)        -> -10
     (TRUNCATE 32766.2)      -> 32766
     (TRUNCATE 32768.)       ->
                                ** appel de GENARITH
```

(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 entier, FLOAT provoque l'interruption GENARITH.

```
ex : (FLOAT 123)    -> 123.
```

4.1.4 Les fonctions de l'arithmétique générique

(+ <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 : (+)                              -> 0
     (+ 8)                            -> 8
     (+ 5 6)                          -> 11
     (+ -5 -6 1)                      -> -10
     (+ 32000 32000)                  ->
                                   ** appel de GENARITH
     (+ 32000. 32000. 1)              -> 64001.
     (+ 100. 1000. 10000. 100000.)    -> 111100.
```

(1+ <n>) [SUBR à 1 argument]

est équivalent à (+ <n> 1).

(- <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 à droite de la soustraction, 0. Si cette fonction ne possède qu'un argument elle réalise une négation arithmétique.

```
ex : (-)                    -> 0
     (- 20)                 -> -20
     (- 123.)               -> -123.
     (- 6 2)                -> 4
     (- 12 -7)              -> 19
     (- 10000. 1000. 100)   -> 8900.
     (- #$8000 1 1)         ->
                               ** appel de GENARITH
```

(1- <n>) [SUBR à 1 argument]

est équivalent à (- <n> 1).

(ABS <n>) [SUBR à 1 argument]

retourne la valeur absolue de l'argument <n>, |<n>|.

```
ex : (ABS 10)       -> 10
     (ABS -10)      -> 10
     (ABS 10.23)    -> 10.23
     (ABS -10.23)   -> 10.23
```

; pour calcule le carré de x en conservant son signe,
; utilisez la méthode de Gérard Baudet.

```
     (* x (ABS x))
```

(* <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 : (*)            -> 1
     (* 5)          -> 5
     (* 10 4)       -> 40
     (* 2 -3)       -> -6
     (* -100 -100)  -> 10000
     (* 1000 1000)  ->
                              ** appel de GENARITH
     (* 2.2 3 2)    -> 13.2
```

(/ <n1> <n2>) [SUBR à 1 ou 2 arguments]
(// <n1> <n2>) [SUBR à 1 ou 2 arguments]

retourne la valeur de la division de <n1> par <n2>. Si un seul argument est fourni calcule l'inverse : 1 / <n1>. Si les 2 arguments sont des entiers et si le quotient n'est pas juste, il y a appel de l'interruption GENARITH qui doit, si

elle le peut, calculer un nombre rationnel; pour obtenir un résultat flottant il faut utiliser la fonction DIVIDE et pour obtenir un quotient entier approché il faut utiliser la fonction QUOTIENT. Si l'argument <n2> est égal à 0, une erreur se produit.

```
ex : (/ 40.)          ->  .025
     (/ 5)            ->
                              ** appel de GENARITH
     (/ 20 5)         ->  4
     (/ 21 5)         ->
                              ** appel de GENARITH
     (/ 40 -4)        ->  -10
     (/ 123.45 6.7)   ->  18.4254
     (/ 4 0)          ->
                              ** appel de GENARITH
```

(QUO <n1> <n2>) [SUBR à 2 arguments]
(QUOTIENT <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du quotient entier de : <n1> / <n2>. QUOTIENT retourne toujours une valeur entière approchée donnée par la formule :

Si N et D sont des nombres entiers, le quotient Q et le reste R vérifient la relation :

$$N = Q * D + R \quad avec \quad 0 <= R < |D|$$

Si l'argument <n2> est égal à 0, une erreur se produit. Il n'y a aucune différence entre les fonctions QUO et QUOTIENT.

```
ex : (QUOTIENT 17 7)          ->  2
     (QUOTIENT 17 -7)         ->  -2
     (QUOTIENT -17 7)         ->  -3
     (QUOTIENT -17 -7)        ->  3
     (QUOTIENT 12. 5)         ->  2.4
     (QUOTIENT 123.45 6.7)    ->  18
     (QUOTIENT 10 0)          ->
                               ** appel de GENARITH
```

(MODULO <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du reste de la division entière de <n1> et <n2>. Si l'un de ces arguments n'est pas un entier l'interruption GENARITH se déclenche. Le reste de la division entière (donné par la formule de la fonction précédente) est toujours positif.

```
ex : (MODULO 17 7)          ->  3
     (MODULO 17 -7)         ->  3
     (MODULO -17 7)         ->  4
     (MODULO -17 -7)        ->  4
     (MODULO -8 2)          ->  0
     (MODULO 123 0)         ->
                               ** appel de GENARITH
     (MODULO 12.4 2)        ->
                               ** appel de GENARITH
```

(QUOMOD <n1> <n2>) [SUBR à 2 arguments]
#:EX:MOD [Variable]

cette fonction combine les deux fonctions précédentes. Elle retourne la même valeur que QUOTIENT et charge dans la variable globale #:EX:MOD le résultat de l'application de MODULO sur <n1> et <n2>.

QUOMOD peut être défini en Lisp de la manière suivante :

```
(DE QUOMOD (n1 n2)
    ; l'implantation réelle est beaucoup plus efficace.
    (SETQ #:EX:MOD (MODULO n1 n2))
    (QUOTIENT n1 n2))
```

(MIN <n1> ... <nN>) [SUBR à 1 ou N arguments]

retourne la plus petite valeur de <n1> ... <nN>.

```
ex : (MIN 10)          -> 10
     (MIN 10 20)       -> 10
     (MIN -10 -20)     -> -20
     (MIN 1 3. 2 -7)   -> -7
     (MIN -2. 3 0 7)   -> -2
```

(MAX <n1> ... <nN>) [SUBR à 1 ou N arguments]

retourne la plus grande valeur de <n1> ... <nN>.

```
ex : (MAX 10)          -> 10
     (MAX 10 20)       -> 20
     (MAX -10 -20)     -> -10
     (MAX 1 3. 2 -7)   -> 3.
     (MAX -2 3. 0 7)   -> 7
```

4.1.5 Les prédicats de l'arithmétique générique

Ces fonctions possèdent en général N arguments qui peuvent être des nombres de n'importe quel type (entier, flottant ou numérique étendu). 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.

Toutes ces fonctions retournent le 1er argument si le test est vérifié et () dans le cas contraire.

En cas d'appel de l'interruption GENARITH, seule la fonction (<?>) doit être définie, toutes les autres étant définies en fonction de celle ci.

(<?> <n1> <n2>) [SUBR à 2 arguments]

retourne -1 si <n1> < <n2>, 0 si <n1> = <n2> et 1 si <n1> > <n2>.

```
ex : (<?> 100 100)      -> 0
     (<?> 100 200.)     -> -1
     (<?> 200. 100)     -> 1
```

(= <n1> <n2> ... <nN>) [SUBR à N arguments]

si <n1> = <n2> = ... = <nN> alors = retourne <n1>, sinon = retourne ().

```
ex : (= 10 10)        -> 10
     (= -3 -2)        -> ()
     (= 10 10.)       -> 10
     (= 10 10. 10)    -> 10
     (= 10 9 10)      -> ()
```

(<> <n1> <n2> ... <nN>) [SUBR à 2 arguments]
(/= <n1> <n2> ... <nN>) [SUBR à 2 arguments]

si <n1> est différent <n2> alors <> retourne <n1>, sinon <> retourne (). Cette fonction est binaire du fait de la non transitivité de l'opérateur <>. Il n'y a aucune différence entre les fonctions <> et /=.

```
ex : (<> 11 10)       -> 11
     (<> -3 -3)       -> ()
     (<> 10. 10)      -> ()
```

(>= <n1> <n2> ... <nN>) [SUBR à N arguments]

si <n1> >= <n2> ... >= <nN> alors >= retourne <n1> sinon >= retourne (). Ces fonctions peuvent être utilisées pour tester des intervalles.

```
ex : (>= 3 7)         -> ()
     (>= 7 7)         -> 7
     (>= 9. 9)        -> 9.
     (>= 10 11 11)    -> 10
     (>= 10 11 10)    -> ()
```

(> <n1> <n2> ... <nN>) [SUBR à N arguments]

si <n1> > <n2> ... > <nN> alors > retourne <n1> sinon > retourne ().

```
ex : (> 5 5)          -> ()
     (> 7 4)          -> 7
     (> 9. 9)         -> ()
     (> 10 10 11)     -> ()
     (> 10 11 12)     -> 10
```

(<= <n1> <n2> ... <nN>) [SUBR à N arguments]

si <n1> <= <n2> ... <= <nN> alors <= retourne <n1> sinon <= retourne ().

```
ex : (<= 5 5)         -> 5
     (<= 4 6)         -> 4
     (<= 9. 9)        -> 9.
     (<= 10 10 9)     -> 10
     (<= 10 9 10)     -> ()
```

(< <n1> <n2> ... <nN>) [SUBR à N arguments]

si <n1> < <n2> .. < <nN> alors < retourne <n1> sinon < retourne ().

```
ex : (< 5 5)          -> ()
     (< 4 5)          -> 4
     (< 9. 9)         -> ()
```

```
(< 8 9 10)    -> 8
(< 8 9 9)     -> ()
```

(ZEROP <n>) [SUBR à 1 argument]

est équivalent à (= n 0)

(PLUSP <n>) [SUBR à 1 argument]

est équivalent à (>= n 0)

(MINUSP <n>) [SUBR à 1 argument]

est équivalent à (< n 0)

4.1.6 Les fonctions circulaires et mathématiques

Toutes ces fonctions ont des arguments de type flottant ou entiers. S'il ne le sont pas l'interruption GENARITH se déclenche.

(SIN <n>) [SUBR à 1 argument]
(COS <n>) [SUBR à 1 argument]
(ASIN <n>) [SUBR à 1 argument]
(ACOS <n>) [SUBR à 1 argument]
(ATAN <n>) [SUBR à 1 argument]

```
ex : (SETQ pi 3.1415926535)      -> 3.1415926535
     (SETQ pi/2 (/ pi 2.))       -> 1.570795
     (SETQ pi/4 (/ pi 4.))       -> 0.7853975
     (SIN 0)                     -> 0
     (SIN pi/4)                  -> 0.707107
     (SIN pi/2)                  -> 1.
     (ASIN (SIN 1.2))            -> 1.2
     (COS 0)                     -> 1
     (COS pi/4)                  -> 0.707107
     (COS pi/2)                  -> 0
     (ACOS (COS 1.2))            -> 1.2
     (* 4 (ATAN 1))              -> 3.14159

     (DE FNT (x y)
         (* nf (+ (COS (/ x nd)) (COS (/ y nd)) 2)))

     (DE FRISE (nl nf nd)
         (LET ((x) (y 30))
              (REPEAT nl
                      (SETQ x -30 y (1- y))
                      (REPEAT 59
                         (PRINCN
                           (CHRNTH (FIX (FNT (INCR x) y))
                                   "0 1 2 3 4 5 6 7 8 9 ")))
                      (TERPRI))))

     (FRISE 13 3.5 5.5)        produira
```

```
55 44  33       33 44  5   66666   5  44 33       33  44 55
5 44  33  2222   33 44  55       55  44 33   2222  33  44 5
 44  3  22222222  33 44  555   555  44 33  22222222  3  44
44  3  22      22  33 44  5555555  44 33  22      22  3  44
4  3  22   11    22 33 44         44 33 22    11   22  3  4
  3  22  1111111  22 33  44     44  33 22  1111111  22  3
 3  22 111    111  22 33  4444444  33 22  111    111 22  3
33 22 11        11  2  33   444   33  2  11        11 22 33
3 22  1   0000   11 22  33       33  22 11   0000   1  22 3
  2  11  000000   1  22 333     333 22  1   000000  11  2
 22  1  00000000  11 22  333   333  22 11  00000000  1  22
 22 11  00000000  11  2  3333 3333  2  11  00000000  11 22
 22 11  00000000  11  2  3333 3333  2  11  00000000  11 22
```

(EXP <n>) [SUBR à 1 argument]
(LOG <n>) [SUBR à 1 argument]
(LOG10 <n>) [SUBR à 1 argument]
(POWER <n1> <n2>) [SUBR à 2 arguments]
(SQRT <n>) [SUBR à 1 argument]

```
ex : (POWER 10 0)                -> 1.
     (POWER 10 1)                -> 10.
     (POWER 10 10)               -> 1e+10
     (POWER 10 -1)               -> .1
     (POWER 10 -2)               -> .01
     (POWER 123.45 6.7)          -> 1.03036e+14
     (EXP 0)                     -> 1.
     (EXP 1)                     -> 2.718282
     (LOG 1)                     -> 0.
     (LOG 2.718282)              -> 1.
     (LOG (EXP 1))               -> 1.
     (LOG10 1)                   -> 0.
     (LOG10 10)                  -> 1.
     (LOG10 20)                  -> 1.30103
     (LOG10 100)                 -> 2.
     (SQRT 100)                  -> 10.
     (SQRT 1000000.)             -> 1000.
     (SETQ x 1.2)                -> 1.2
     (SQRT (+ (POWER (SIN x) 2) (POWER (COS x) 2)))   -> 1.
```

4.1.7 Les nouvelles arithmétiques génériques

Pour créer une nouvelle arithmétique générique, il suffit de définir les fonctions suivantes dans le package du type de l'arithmétique étendue (<ext> dans cette liste). Ces fonctions sont toutes unaires ou binaires, le système se chargeant du traitememt des arguments multiples.

```
(#:<ext>:NUMBERP <s>)
(#:<ext>:FLOAT <s>)
(#:<ext>:TRUNCATE <s>)
(#:<ext>:+ <s1> <s2>)
(#:<ext>:- <s1> <s2>)
```

```
(#:<ext>:0- <s>)
(#:<ext>:/ <s1> <s2>)
(#:<ext>:1/ <s1>)
(#:<ext>:QUOMOD <s1> <s2>)
(#:<ext>:ABS <n>)
(#:<ext>:<?> <s1> <s2>)
(#:<ext>:SIN <s>)
(#:<ext>:COS <s>)
(#:<ext>:ASIN <s>)
(#:<ext>:ACOS <s>)
(#:<ext>:ATAN <s>)
(#:<ext>:EXP <s>)
(#:<ext>:LOG <s>)
(#:<ext>:LOG10 <s>)
(#:<ext>:POWER <s>)
(#:<ext>:SQRT <s>)
```

et pour définir un nouveau type étendu

```
(#:<ext>:EVAL <n>)
(#:<ext>:PRIN <n>)
```

Un exemple de création d'une nouvelle arithmétique étendue est donné au chapitre 10, section 3.

4.2 L'Arithmétique Entière

Les fonctions de cette arithmétique utilisent des opérandes <n> qui doivent être absolument de type entier à précision fixe de 16 bits. Ces fonctions n'effectuent de contrôle de validité de type que dans l'interprète et ne testent jamais les débordements. Si les arguments ne sont pas des nombres entiers, ces fonctions livrent en général de bien étranges résultats dans les fonctions compilées et peuvent même provoquer des erreurs de la machine hôte. Toutes ces fonctions sont le plus souvent compilées au moyen d'une seule instruction LLM3.

Dans l'interprète, si l'un des arguments n'est pas un nombre entier, l'erreur ERRNIA se déclenche dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas un entier : <s>      ou bien
** <fn> : not a fix :  <s>
```

dans lequel le nom de la fonction <fn> qui a provoqué l'erreur est imprimé ainsi que l'argument défectueux <s>.

4.2.1 Les fonctions de l'arithmétique entière

(ADD1 <n>) [SUBR à 1 argument]

retourne la valeur : <n> + 1 .

```
ex : (ADD1 6)      -> 7
     (ADD1 -3)     -> -2
```

(SUB1 <n>) [SUBR à 1 argument]

retourne la valeur : <n> - 1

```
ex : (SUB1 7)      -> 6
     (SUB1 -3)     -> -4
```

(ADD <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la somme : <n1> + <n2>.

```
ex : (ADD 5 6)       -> 11
```

(SUB <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la différence : <n1> - <n2>.

```
ex : (SUB 6 2)         -> 4
```

(MUL <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du produit : <n1> * <n2>.

```
ex : (MUL 10 4)      -> 40
```

(DIV <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur entière du quotient de : <n1> / <n2>. DIV effectue une division entière sur des nombres entiers. Si l'argument <n2> est égal à 0, l'erreur ERR0DV se produit.

```
ex : (DIV 20 5)        -> 4
     (DIV 21 5)        -> 4
```

(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>)
Le résultat intermédiaire est rangé en double précision entière (dépendant de la machine hôte mais d'au moins 32 bits) pour éviter un débordement du calcul intermédiaire. Le résultat final est un entier, éventuellement tronqué sur 16 bits.

```
ex : (SCALE 1000 1000 1000)    -> 1000
     (SCALE -100 2000 -1000)   -> 200
     (SCALE 1000 1000 3000)    -> 333
```

4.2.2 Les comparaisons de l'arithmétique fixe entière

Ces fonctions possèdent toutes 2 arguments <n1> et <n2> qui doivent être des nombres de type entier. Elles n'effectuent les tests de validité de type que dans l'interprète. Si les arguments de ces fonctions ne sont pas des nombres entiers, leurs résultats ne sont pas significatifs dans les fonctions compilées. Toutes ces fonctions sont le plus souvent compilées au moyen d'une seule instruction LLM3.

Toutes ces fonctions retournent le 1er argument si le test est vérifié et () dans le cas contraire.

(EVENP <n>) [SUBR à 1 argument]

si l'argument <n> est pair, alors EVENP retourne <n> sinon EVENP retourne ().

(ODDP <n>) [SUBR à 1 argument]

si l'argument <n> est impair alors ODDP retourne <n> sinon ODDP retourne ().

(EQN <n1> <n2>) [SUBR à 2 arguments]

si <n1> = <n2> alors EQN retourne <n1>, sinon EQN retourne ().

```
ex : (EQN 10 10)      -> 10
     (EQN -3 3)       -> ()
```

(NEQN <n1> <n2>) [SUBR à 2 arguments]

si <n1> /= <n2> alors NEQN retourne <n1>, sinon NEQN retourne ().

```
ex : (NEQN 11 10)     -> 11
     (NEQN -3 -3)     -> ()
```

(GE <n1> <n2>) [SUBR à 2 arguments]

si <n1> >= <n2> alors GE retourne <n1> sinon GE retourne ().

```
ex : (GE 3 7)         -> ()
     (GE 7 7)         -> 7
```

(GT <n1> <n2>) [SUBR à 2 arguments]

si <n1> > <n2> alors GT retourne <n1> sinon GT retourne ().

```
ex : (GT 5 5)         -> ()
     (GT 7 4)         -> 7
```

(LE <n1> <n2>) [SUBR à 2 arguments]

si <n1> <= <n2> alors LE retourne <n1> sinon LE retourne ().

```
ex : (LE 5 5)         -> 5
     (LE 4 6)         -> 4
```

(LT <n1> <n2>) [SUBR à 2 arguments]

si <n1> < <n2> alors LT retourne <n1> sinon LT retourne ().

```
ex : (LT 5 5)      -> ()
     (LT 4 5)      -> 4
```

4.2.3 Les fonctions booléennes

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 de débordement numérique. Toutes ces fonctions sont compilées au moyen d'une seule instruction LLM3.

(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 hexadécimal #$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 <n> est une puissance de 2 évaluez :

```
(= <n> (LOGAND <n> (- <n>)))
```

(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]

décale logiquement la valeur <n1> de <n2> positions. Si <n2> est positif, le décalage s'effectue vers la gauche (c'est-à-dire vers les poids forts, réalisant ainsi une multiplication) et si <n2> est négatif, le décalage s'effectue vers la deoite (c'est-à-dire vers les poids faibles, réalisant ainsi une division).

LOGSHIFT peut être défini en Lisp de la manière suivante :

```
(DE LOGSHIFT (n nb)
    ; cette définition suppose n positif.
    (IF (= nb 0)
        n
        (IF (< nb 0)
```

```
                (LOGSHIFT (DIV n 2) (1+ nb))
                (LOGSHIFT (MUL n 2) (1- nb)))))
ex : (LOGSHIFT 1 0)    -> 1
     (LOGSHIFT 1 1)    -> 2
     (LOGSHIFT 1 3)    -> 8
     (LOGSHIFT 1 15)   -> #$8000
     (LOGSHIFT 8 -3)   -> 1
```

4.2.4 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 bits 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> le nombre de bits de ce champ.

(2 <n>)** **[SUBR à 1 argument]**

retourne la valeur 2 à la puissance <n>, c'est-à-dire le bit de rang <n> d'un mot.

2** peut être défini en Lisp de la manière suivante :

```
(DE 2** (n)
    (LOGSHIFT 1 n))
ex : (2** 0)    -> 1
     (2** 4)    -> 16
     (2** 15)   -> #$8000
```

(LOAD-BYTE <n> <np> <nl>) **[SUBR à 3 arguments]**

extrait du mot <n> le champ de bits spécifié par <np> et <nl>.

LOAD-BYTE peut être défini en Lisp de la manière suivante :

```
(DE LOAD-BYTE (n p l)
    (LOGSHIFT (MASK-FIELD n p l) (- p)))
ex : (LOAD-BYTE #$F0 2 4)        -> #$C
```

(MASK-FIELD <n> <np> <nl>) **[SUBR à 3 arguments]**

retourne le mot résultant de l'isolation du champ de bits <np> <nl>.

MASK-FIELD peut être défini en Lisp de la manière suivante :

```
(DE MASK-FIELD (n p l)
    (LOGAND (LOGSHIFT (1- (2** l)) p) n))
ex : (MASK-FIELD #$F0 2 4)      -> #$30
```

(DEPOSIT-BYTE <n1> <np> <nl> <n2>) **[SUBR à 4 arguments]**

retourne le mot <n1> dans lequel le champ de bits <np> <nl> a été remplacé par la valeur <n2>.

DEPOSIT-BYTE peut être défini en Lisp de la manière suivante :

```
(DE DEPOSIT-BYTE (n1 p l n2)
    (LET ((n  (MASK-FIELD (LOGSHIFT n2 p) p l))
          (nr (LOGAND n1 (1- (2** p))))
          (nl (LOGAND n1 (LOGNOT (1- (2** (+ p l))))))
         (+ n nl nr))))))
ex : (DEPOSIT-BYTE #$F0 2 4 #$3)    -> #$CC
```

(DEPOSIT-FIELD <n1> <np> <nl> <n2>) [SUBR à 4 arguments]

retourne le mot <n1> dans lequel le champ de bits <np> <nl> a été remplacé par le champ de bits correspondant du mot <n2>.

DEPOSIT-BYTE peut être défini en Lisp de la manière suivante :

```
(DE DEPOSIT-FIELD (n1 p l n2)
    (LET ((n  (MASK-FIELD n2 p l))
          (nr (LOGAND n1 (1- (2** p))))
          (nl (LOGAND n1 (LOGNOT (1- (2** (+ l p)))))))
         (+ n nl nr))))))
ex : (DEPOSIT-FIELD #$F0 2 4 #$C)    -> #$CC
```

(LOAD-BYTE-TEST <n> <np> <nl>) [SUBR à 3 arguments]

teste dans le mot <n> si le champ de bits <np> <nl> est égal à zéro. Si le test est vérifié, LOAD-BYTE-TEST retourne ce champ de bits sinon il retourne ().

LOAD-BYTE-TEST peut être défini en Lisp de la manière suivante :

```
(DE LOAD-BYTE-TEST (n p l)
    (<> (LOAD-BYTE n p l) 0))
(LOAD-BYTE-TEST #$F0 2 2)    -> ()
(LOAD-BYTE-TEST #$F0 2 4)    -> #$C
```

4.2.5 Les fonctions pseudo-aléatoires

(SRANDOM <n>) [SUBR à 0 ou 1 argument]

Initialise l'accumulateur interne du générateur pseudo-aléatoire avec le nombre <n>. Si l'argument n'est pas fourni, SRANDOM retourne l'accumulateur courant.

(RANDOM <n1> <n2>) [SUBR à 2 arguments]

retourne un nombre tiré au sort sur l'intervalle [<n1> <n2>[. RANDOM utilise la méthode de congruence linéaire décrite dans [Knuth vol 2]. *L'accumulateur interne* G, est calculé selon la formule

```
        G(n+1) = (A * G(n) + C) modulo M
Avec    M = 7*7*7*7*13,   A = (7 * 13) + 1, C = 2731 (premier).
```

On obtient donc une suite d'accumulateurs de période M (31213), parcourant le segment [0..M-1]. Le calcul du nombre aléatoire se fait alors simplement par la formule :

```
r = n1 + ((G * (n2 - n1)) / M),
```

qui permet d'éviter la mauvaise distribution des chiffres peu significatifs de l'accumulateur pris modulo 7 ou 13.

4.3 L'Arithmétique Entière Etendue

Pour pouvoir implanter les nombres à précision arbitraire, un ensemble de fonctions entières fixes étendues est disponible. Pour toutes ces fonctions les arguments de type <n> sont des entiers de 16 bits. Tous les calculs sont effectués sur des quantités de 16 bits non signées. Par souci d'efficacité, ces fonctions ne testent pas le type de leurs arguments ni ne provoquent d'erreur. L'extension des valeurs de retour, si cette valeur dépasse 16 bits, est toujours donnée dans la variable système #:EX:REGRET.

#:EX:REGRET [Variable]

contient toujours les poids forts des calculs des fonctions entières fixes étendues, représentés sous la forme d'un entier de 16 bits.

(EX+ <n1> <n2>) [SUBR à 2 arguments]

Retourne les poids faibles, sur 16 bits, de la somme :

```
<n1> + <n2> + #:ex:regret
```

La nouvelle retenue se retrouve dans la variable #:EX:REGRET.

```
ex : (SETQ #:EX:REGRET 0)          -> 0
     (EX+ 100 200)                 -> 300
     #:EX:REGRET                   -> 0
     (SETQ #:EX:REGRET 1)          -> 1
     (EX+ 100 200)                 -> 301
     #:EX:REGRET                   -> 0
     (SETQ #:EX:REGRET 0)          -> 0
     (EX+ #$FFFF #$FFFF)           -> -2
     #:EX:REGRET                   -> 1
     (SETQ #:EX:REGRET 1)          -> 1
     (EX+ #$FFFF #$FFFF)           -> -1
     #:EX:REGRET                   -> 1
```

(EX1+ <n>) [SUBR à 1 argument]

Retourne les poids faibles, sur 16 bits, de la somme :

```
<n> + 1 + #:ex:regret
```

donc équivalent à :

```
(EX+ <n> 1)
```

La nouvelle retenue se retrouve dans la variable #:EX:REGRET.

```
ex : (SETQ #:EX:REGRET 0)          -> 0
     (EX1+ 10)                     -> 11
     #:EX:REGRET                   -> 0
     (EX1+ #$FFFF)                 -> 0
```

```
#:EX:REGRET                    -> 1
```

(EX- <n>) [SUBR à 1 argument]

Retourne, sur 16 bits, la différence :

```
- <n> - 1
```

```
ex : (EX- -2)    -> 1
     (EX- -1)    -> 0
     (EX- 0)     -> -1
     (EX- 1)     -> -2
     (EX- 2)     -> -3
```

(EX* <n1> <n2> <n3>) [SUBR à 3 arguments]

Retourne les poids faibles, sur 16 bits, du calcul :

```
<n1> * <n2> + <n3> + #:ex:regret
```

Les poids forts se retrouvent dans la variable #:EX:REGRET.

```
ex : (SETQ #:EX:REGRET 0)    -> 0
     (EX* 100 100 10)        -> 10010
     #:EX:REGRET             -> 0
     (SETQ #:EX:REGRET 0)    -> 0
     (EX* -1 -1 0)           -> 1
     #:EX:REGRET             -> -2
     (SETQ #:EX:REGRET -1)   -> -1
     (EX* -1 -1 0)           -> 0
     #:EX:REGRET             -> -1
```

(EX/ <n1> <n2>) [SUBR à 2 arguments]

#:EX:REGRET contient les poids forts du dividende, <n1> les poids faibles de ce même dividende. EX/ retourne, sur 16 bits, le quotient de ce dividende par le diviseur <n2> tandis que le reste se retrouve dans #:EX:REGRET.

```
#:ex:regret|<n1> / <n2>     retourné en valeur et
#:ex:regret|<n1> rem <n2>   -> #:ex:regret
```

```
ex : (SETQ #:EX:REGRET 0)     -> 0
     (EX/ 100 5)              -> 20
     #:EX:REGRET              -> 0
     (SETQ #:EX:REGRET 1)     -> 1
     (EX/ 100 5)              -> 13127
     #:EX:REGRET              -> 1
     (SETQ #:EX:REGRET -2)    -> -2
     (EX/ 0 -1)               -> -2
     #:EX:REGRET              -> -2
     (SETQ #:EX:REGRET -3)    -> -3
     (EX/ 3 -1)               -> -2
     #:EX:REGRET              -> 1
```

(EX? <n1> <n2>) [SUBR à 2 arguments]

Retourne -1 si <n1> est plus petit strictement que <n2>, retourne 0 si <n1> égal <n2> et retourne 1 si <n1> est strictement plus grand que <n2>. Les comparaisons opèrent sur des quantités de 16 bits non signées.

```
ex : (EX? 0 0)     -> 0
     (EX? 2 3)     -> -1
     (EX? 3 2)     -> 1
     (EX? -1 -2)   -> 1
     (EX? -1 1)    -> 1
     (EX? -1 -1)   -> 0
     (EX? -2 -1)   -> -1
```

4.4 L'Arithmétique Flottante

Les fonctions de cette arithmétique utilisent des opérandes <n> qui doivent être obligatoirement de type flottant. Ces fonctions n'effectuent de contrôle de validité de type que dans l'interprète. Si les arguments ne sont pas des nombres flottants, ces fonctions livrent en général de bien étranges résultats dans les fonctions compilées et peuvent même provoquer des erreurs de la machine hôte. Toutes ces fonctions sont compilées au moyen d'une seule instruction LLM3.

Dans l'interprète, si l'un des arguments n'est pas un nombre flottant, l'erreur ERRNFA se déclenche dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas un flottant : <s>      ou bien
** <fn> : not a float :  <s>
```

dans lequel le nom de la fonction <fn> qui a provoqué l'erreur est imprimé ainsi que l'argument défectueux <s>.

4.4.1 Les fonctions de l'arithmétique flottante

(FADD <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la somme : <n1> + <n2>.

```
ex : (FADD 5. 6.)        -> 11.
```

(FSUB <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la différence : <n1> - <n2>.

```
ex : (FSUB 6. 2.)        -> 4.
```

(FMUL <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du produit : <n1> * <n2>.

```
ex : (FMUL 10. 4.)       -> 40.
```

(FDIV <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.

```
ex : (FDIV 11. 2.)      -> 5.5
     (FDIV 10. 2.)      -> 5.
```

4.4.2 Les comparaisons de l'arithmétique flottante

Ces fonctions possèdent toutes 2 arguments <n1> et <n2> qui doivent être des nombres flottants. Elles n'effectuent les tests de validité de type que dans l'interprète. Si les arguments de ces fonctions ne sont pas des nombres flottants, leurs résultats ne sont pas significatifs dans les fonctions compilées.

Toutes ces fonctions retournent le 1er argument si le test est vérifié et () dans le cas contraire.

(FEQN <n1> <n2>) [SUBR à 2 arguments]

si <n1> = <n2> alors FEQN retourne <n1>, sinon FEQN retourne ().

```
ex : (FEQN 10. 10.)     -> 10.
     (FEQN -3. 3.)      -> ()
```

(FNEQN <n1> <n2>) [SUBR à 2 arguments]

si <n1> # <n2> alors FNEQN retourne <n1>, sinon FNEQN retourne ().

```
ex : (FNEQN 11. 10.)    -> 11.
     (FNEQN -3. -3.)    -> ()
```

(FGE <n1> <n2>) [SUBR à 2 arguments]

si <n1> >= <n2> alors FGE retourne <n1> sinon FGE retourne ().

```
ex : (FGE 3. 7.)        -> ()
     (FGE 7. 7.)        -> 7.
```

(FGT <n1> <n2>) [SUBR à 2 arguments]

si <n1> > <n2> alors FGT retourne <n1> sinon FGT retourne ().

```
ex : (FGT 5. 5.)        -> ()
     (FGT 7. 4.)        -> 7.
```

(FLE <n1> <n2>) [SUBR à 2 arguments]

si <n1> <= <n2> alors FLE retourne <n1> sinon FLE retourne ().

```
ex : (FLE 5. 5.)        -> 5.
     (FLE 4. 6.)        -> 4.
```

(FLT <n1> <n2>) [SUBR à 2 arguments]

si <n1> < <n2> alors FLT retourne <n1> sinon FLT retourne ().

```
ex : (FLT 5. 5.)      -> ()
     (FLT 4. 5.)      -> 4.
```

4.5 L'Arithmétique Mixte

Les arguments de ces fonctions peuvent être des nombres entiers ou flottants. Si l'un des arguments est flottant ou *si un calcul entier déborde,* 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. Si un argument n'est pas un nombre entier ou un nombre flottant, l'erreur ERRNNA se déclenche dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas un nombre : <s>      ou bien
** <fn> : not a number :  <s>
```

dans lequel le nom de la fonction <fn> qui a provoqué l'erreur est imprimé ainsi que l'argument défectueux <s>.

Il n'y a pas de comparaisons spécifiques pour ce type d'arithmétique : les comparaisons génériques conviennent.

Cette arithmétique est donc une arithmétique flottante dans laquelle on peut utiliser des nombres entiers.

(PLUS <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur de la somme : <n1> + <n2> + ... + <nN>. Si aucun argument n'est fourni, PLUS retourne l'élément neutre de l'addition, 0.

```
ex : (PLUS)                              -> 0
     (PLUS 10)                           -> 10
     (PLUS 5 6)                          -> 11
     (PLUS -5 -6 1)                      -> -10
     (PLUS 100. 1000. 10000. 100000.)    -> 111100.
     (PLUS 32000 32000 1)                -> 64001.
```

(DIFFER <n1> ... <nN>) [SUBR à N arguments]
(DIFFERENCE <n1> ... <nN> [SUBR à N arguments]

retourne la valeur de la différence : <n1> - <n2> - ... - <nN>. Si aucun argument n'est fourni, DIFFERENCE retourne l'élément neutre à droite de la soustraction, 0. Si cette fonction ne possède qu'un argument elle réalise une négation arithmétique. Il n'y a aucune différence entre les fonctions DIFFER et DIFFERENCE.

```
ex : (DIFFERENCE)                    -> 0
     (DIFFERENCE 20.)                -> -20.
     (DIFFERENCE 6 2)                -> 4
     (DIFFERENCE 12 -7)              -> 19
     (DIFFERENCE 10000. 1000. 100)   -> 8900.
```

```
(DIFFERENCE #$8000 1 1)          ->  -32770
```

(TIMES <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur du produit : <n1> * <n2> * ... * <nN>. Si aucun argument n'est fourni, TIMES retourne l'élément neutre de la multiplication, 1.

```
ex : (TIMES)              ->  1
     (TIMES 5)            ->  5
     (TIMES 10 4)         ->  40
     (TIMES 2 -3)         ->  -6
     (TIMES -100 -100)    ->  10000
     (TIMES 2.2 3 2)      ->  13.2
     (TIMES 1000 1000)    ->  1.e+06
```

(DIVIDE <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du quotient de : <n1> / <n2>. Si les deux arguments sont des nombres entiers et si le quotient n'est pas juste, DIVIDE retourne une valeur flottante.
Si l'argument <n2> est égal à 0, une erreur se produit.

```
ex : (DIVIDE 10 5)          ->  2
     (DIVIDE 12 5)          ->  2.4
     (DIVIDE 12. 5)         ->  2.4
     (DIVIDE 40 -4)         ->  -10
     (DIVIDE 123.45 6.7)    ->  18.4254
```

CHAPITRE 5

Structures, Types Etendus et Programmation Objet

Matthieu Devin

Ce chapitre décrit d'abord les fonctions qui permettent de créer des objets structurés en Lisp. Ces objets sont analogues aux *records* de Pascal, Lisp permettant en plus de créer une hiérarchie de types de structures.

Il décrit ensuite la hiérarchie des types Lisp et les moyens de l'étendre : vecteurs typés, chaînes typées et doublets étiquetés. De par leur implantation sous forme de vecteurs typés, les structures fournissent le moyen le plus immédiat d'étendre la typologie Le_Lisp.

Nous expliquons ensuite comment utiliser les fonctions primitives de recherche dans l'oblist et les fonctions de typage pour implanter un langage orienté objet dans le style de SMALLTALK.

Cette partie du système Le_Lisp a bénéficié des expériences de Ceyx [Hullot 83] et d'Alcyone [Hullot 85] qui ont permis de dégager un noyau minimal permettant la construction d'extensions orientées objet.

5.1 Les Structures

La primitive DEFSTRUCT permet de définir de nouveaux types d'objets structurés (structures) analogues aux *records* de Pascal. La fonction NEW permet de créer des *instances* de ces types d'objets.

Les objets structurés sont formés d'un certain nombre de champs nommés. Chaque champ peut contenir un objet Lisp quelconque. Les champs ne sont pas typés. On peut consulter et modifier le contenu des champs d'un objet structuré par des *fonctions d'accès aux champs*, qui sont définies au moment de la définition des structures.

On peut définir une structure comme sous-structure d'une structure définie antérieurement. Dans ce cas la sous-structure hérite de tous les champs de la structure en supplément des champs déclarés lors de sa définition.

5.1.1 Définition d'une structure

(DEFSTRUCT <struct> <champ1> ... <champN>) [MACRO]

définit une structure de nom <struct>, qui est un symbole, comportant les champs <champ1> ... <champN>. <struct> devient un nouveau type d'objets Lisp (voir la fonction TYPE-OF).

<champ1> ... <champN> sont les noms des champs de la structure. Ce sont soit des symboles, soit des listes à deux éléments. Dans ce cas le premier élément de

la liste est le nom du champ. Le second élément est une forme Lisp qui permet de calculer la valeur initiale de ce champ lors de la création d'instances. Cette forme sera évaluée lors de la création de chaque instance de la structure.

Lors de la définition de la structure les fonctions d'accès en lecture et en écriture aux champs, ainsi qu'une fonction spécifique de création d'instances de la structures sont définies. Ces fonctions sont décrites plus bas.

Si <struct> a la forme #:<sur-structure>:<structure>, où <sur-structure> est le nom d'une structure déja définie dans le système, <struct> comportera tous les champs de <sur-structure> en plus des champs <champ1>...<champN>. En ce cas le type #:<sur-structure>:<structure> est un sous-type du type <sur-structure>.

5.1.2 Création d'une instance

(NEW <struct>) [SUBR à 1 argument]
(#:<struct>:MAKE) [SUBR à 0 argument]

Ces deux fonctions créent et retournent en valeur une nouvelle instance de la structure <struct>. La fonction #:<struct>:MAKE est la fonction de création spécifique à la structure <struct>. Elle est automatiquement définie par la fonction DEFSTRUCT. L'objet créé a le type <struct> (voir la fonction TYPE-OF).

Les champs de l'objet créé sont éventuellement initialisés avec les formes d'initialisation fournies lors de la définition de la structure. Ces formes sont évaluées au moment de la création de l'objet. L'ordre d'évaluation des formes d'initialisation n'est pas précisé. Les champs pour lesquels aucune forme d'initialisation n'a été fournie sont initialisés à ().

Si l'argument de la fonction NEW n'est pas le nom d'une structure définie dans le système, l'erreur ERRSTC est déclenchée, dont le libellé est :

```
** <fn> : l'argument n'est pas une structure : <s>     ou bien
** <fn> : not a structure : <s>
```

dans lequel <fn> est le nom de la fonction ayant provoqué l'erreur (ici NEW) et <s> l'argument fautif.

NEW peut être défini en Lisp de la manière suivante :

```
(DE NEW (type)
    (IFN (GETPROP type 'DEFSTRUCT)
         (ERROR 'NEW 'ERRSTC type)
         (APPLY (SYMBOL type 'MAKE) ())))
```

5.1.3 Accès aux champs

(#:<struct>:<champ> <o> [<e>]) [SUBR à 1 ou 2 arguments]

<struct> est le nom d'une structure, <champ> est le nom d'un champ de cette structure. Les fonctions #:<struct>:<champ> sont les fonctions d'accès aux champs des instances de la structure <struct>. Elles sont définies automatiquement par la fonction DEFSTRUCT.

Si l'objet <o> n'est pas une instance de la structure <struct> (voir la fonction TYPEP), l'erreur ERRSTC est déclenchée. Si un seul argument est fourni cette

fonction rend la valeur du champ <champ> de l'objet <o>. Si un second argument est fourni sa valeur est stockée dans le champ <champ> de l'objet <o> et cette valeur est rendue en résultat.

Si le compilateur Le_Lisp est présent au moment de la définition de la structure (c'est-à-dire si (FEATUREP 'COMPILER) est vrai), les fonctions d'accès aux champs sont définies comme des macros du compilateur (voir la fonction DEFMACRO-OPEN). La compilation des appels de ces fonctions engendre alors un code extrêmement efficace. Dans ce cas, après compilation, les fonctions d'accès aux champs n'effectuent aucun test de type sur leur premier argument.

5.1.4 Test de type

(STRUCTUREP <o>) [SUBR à 1 argument]

Ce prédicat est vrai si l'objet <o> est une instance d'une structure définie dans le système. (Voir aussi la fonction TYPEP).

ex : définition d'une structure HOMME à deux champs AGE et POIDS.
Le champ POIDS sera initialisé à la valeur 75.

```
? (defstruct homme
?            age
?            (poids 75))
= homme
?
```

Création d'une instance

```
? (setq homme1 (#:homme:make))
= #:homme:#[() 75]
```

Remplissage du champ AGE de l'objet homme par 40

```
? (#:homme:age homme1 40)
= 40
```

Extraction du champ POIDS

```
? (#:homme:poids homme1)
= 75
```

Fonction STRUCTUREP

```
? (structurep homme1)
= t
```

Exemple de sous-structure :

Extension de la structure HOMME à celle de CHERCHEUR

```
? (defstruct #:homme:chercheur
?            projet
?            articles)
= #:homme:chercheur
```

Création d'un chercheur et initialisation d'un champ

```
? (setq c1 (#:homme:chercheur:make))
= #:homme:chercheur:#[() 75 () ()]
? (#:homme:chercheur:articles c1 '("confUSA" "These"))
= (confUSA These)
```

Le chercheur hérite des champs de l'homme

```
? (#:homme:poids c1)
= 75
? (#:homme:age c1 26)
```

```
= 26
? (#:homme:chercheur:poids c1)
= 75
? (#:homme:chercheur:age c1)
= 26
```

5.1.5 Implantation des structures

Les instances d'une structure sont des vecteurs typés dont le type est le nom de la structure. Les valeurs des champs des objets sont stockées dans le vecteur.

L'espace mémoire occupé par une structure à N champs est N+4 pointeurs.

5.2 La Typologie Lisp

Les types des objets Le_Lisp sont représentés par des symboles. La hiérarchie des packages Lisp permet de hiérarchiser ces types : le symbole #:HOMME:CHERCHEUR représente le sous-type CHERCHEUR du type HOMME.

Il n'est pas nécessaire de *créer* explicitement de nouveaux types, la simple écriture du symbole #:FOO:BAR permet de parler du type FOO et de son sous-type BAR. Tout symbole Lisp peut donc être interprété comme un type.

En pratique les types intéressants sont ceux que peut retourner la fonction TYPE-OF : l'un des types prédéfinis du système ou bien un *type de l'utilisateur* (c'est-à-dire le type d'un vecteur typé, d'une chaîne typée, ou d'un doublet étiqueté).

Les types prédéfinis au lancement du système sont les types de base Lisp. Ils sont tous sous-type du type ||, qui est le type universel.
Les types de base Lisp sont :

- FIX : les petits entiers
- FLOAT : les nombres flottants
- SYMBOL : les symboles
- NULL : le marqueur de fin de liste ()
- STRING : les chaînes de caractères
- VECTOR : les vecteurs de pointeurs
- CONS : les cellules de listes

(TYPE-OF <s>) [SUBR à 1 argument]

Cette fonction permet de calculer le type d'un objet Lisp quelconque. Si l'argument est une chaîne ou un vecteur cette fonction rend le type de cet objet (fonctions TYPEVECTOR et TYPESTRING). Si l'argument est un doublet étiqueté cette fonction rend en valeur le CAR du doublet si celui ci est un atome ou une liste dont le CAR est un symbole. Sinon la fonction rend le type Lisp de l'objet (FIX, FLOAT, CONS etc..).
Dans le cas ou le CAR d'un doublet étiqueté n'est ni un symbole ni une liste dont le CAR est un symbole, le résultat de la fonction TYPE-OF n'est pas défini.

TYPE-OF peut être défini en Lisp de la manière suivante :

```
(DE TYPE-OF (s)
    (COND ((FIXP s) 'FIX)
          ((FLOATP s) 'FLOAT)
          ((NULL s) 'NULL)
          ((SYMBOLP s) 'SYMBOL)
          ((CONSP s)
           (IF (TCONSP s)
               (WHEN (OR (ATOM (CAR s)) (SYMBOLP (CAAR S)))
                 (CAR s))
               'CONS))
          ((VECTORP s) (TYPEVECTOR s))
          ((STRINGP s) (TYPESTRING s))
          (T ())))
```

```
ex : ? (TYPE-OF 12)
     = FIX
     ? (TYPE-OF 12.2)
     = FLOAT
     ? (TYPE-OF ())
     = NULL
     ? (TYPE-OF 'ASD)
     = SYMBOL
     ? (TYPE-OF "assasa")
     = STRING
     ? (TYPE-OF #[1 12 3])
     = VECTOR
     ? (TYPE-OF #:FOO:BAR:"Hello Mamma")
     = #:FOO:BAR
     ? (TYPE-OF #:HOMME:#[() 75])
     = HOMME
     ? (TYPE-OF '(1))
     = CONS
     ? (TYPE-OF '#(FOO . A))
     = FOO
```

(SUBTYPEP <type1> <type2>) [SUBR à 2 arguments]

<type1> et <type2> sont deux symboles représentant des types Lisp. Ce prédicat est vrai si <type1> est un sous-type de <type2>.

SUBTYPEP peut être défini en Lisp de la manière suivante :

```
(DE SUBTYPEP (type1 type2)
    (OR (EQ type1 type2)
        (IFN (PACKAGECELL type1)
             (NULL type2)
             (SUBTYPEP (PACKAGECELL type1) type2))))
```

```
ex : ? (SUBTYPEP '#:HOMME:CHERCHEUR 'HOMME)
     = t
     ? (SUBTYPEP 'SYMBOL 'SYMBOL)
     = t
     ? (SUBTYPEP '#:HOMME:CHERCHEUR '#:HOMME:TROUVEUR)
     = ()
```

```
? (SUBTYPEP 'FIX '||)
= t
```

(TYPEP <s> <type>) [SUBR à 2 arguments]

ce prédicat est vrai si le type de <s> est un sous-type du type <type>.

TYPEP peut être défini en Lisp de la manière suivante :

```
(DE TYPEP (s type)
    (SUBTYPEP (TYPE-OF s) type))
```

```
ex : ? (TYPEP (#:HOMME:CHERCHEUR:MAKE) 'HOMME)
     = t
     ? (TYPEP 'FOO 'SYMBOL)
     = t
     ? (TYPEP (#:HOMME:CHERCHEUR:MAKE) '#:HOMME:TROUVEUR)
     = ()
     ? (TYPEP 12 '||)
     = t
```

5.3 Programmation Orientée Objet

La hiérarchie des packages Le_Lisp est utilisée pour encoder la hiérarchie des types de données. Dans un langage orienté objet à la SMALLTALK (comme Ceyx ou Alcyone par exemple), on veut pouvoir attacher des *méthodes* aux types des objets.

En Le_Lisp ces méthodes sont des fonctions Lisp, définies dans les packages des types auxquels elles sont attachées. Ainsi la fonction #:HOMME:DECRIT est-elle considérée comme la méthode DECRIT attachée au type HOMME.

On désire que les sous-types d'un type donné *héritent* des méthodes attachées à ce type. Par exemple les objets de type #:HOMME:CHERCHEUR doivent hériter de la méthode DECRIT des objets de type HOMME. La fonction prédéfinie GETFN permet d'effectuer cette recherche de méthodes avec héritage.

On désire parfois qu'un type d'objets hérite des méthodes de plusieurs types Lisp : c'est l'héritage multiple. La fonction GETFN permet de rechercher une fonction dans une liste de packages en faisant une recherche en profondeur d'abord dans les packages supérieurs aux packages de cette liste.

On désire enfin rechercher une méthode associée à la combinaison de plusieurs types d'objets. Par exemple, pour implanter une arithmétique générique on a besoin de définir des méthodes d'addition permettant d'additionner n'importe quels types de nombres (FLOAT, FIX, BIGNUM, RATIONAL, etc.). En Le_Lisp les méthodes du produit des types T1 et T2 sont dans le *package produit* (T1 . T2). Une telle méthode se note #:(T1 . T2):ADD. La fonction prédéfinie GETFN2 permet de rechercher les méthodes des types produits.

5.3.1 Recherche des méthodes

Les fonction ci-dessous sont les fonctions prédéfinies de recherche de fonctions dans l'oblist. Grâce à une implantation particulièrement étudiée de l'oblist ces fonctions sont extrêmement efficaces.

(GETFN1 <fn> <pkgc>) [SUBR à 2 arguments]

Cette fonction recherche une fonction de nom <fn> dans le package <pkgc>. Si cette fonction existe elle est retournée sinon GETFN1 retourne ().

(GETFN <fn> <pkgc1> [<pkgc2>]) [SUBR à 2 ou 3 arguments]

Cherche une fonction de nom <fn> dans le package <pkgc1> ou l'un des packages qui lui sont supérieurs. Si l'argument <pkgc2> est fourni il est utilisé comme package d'arrêt exclus : si dans la remontée des packages pères de <pkgc1> on arrive à <pkgc2> la recherche est abandonnée. On abandonne toujours la recherche avant de chercher dans le package <pkgc2>. Si l'on trouve la fonction elle est retournée, sinon GETFN retourne ().

Si <pkgc1> est une liste de packages, la recherche a d'abord lieu dans le premier package de cette liste et dans tous ses pères, puis dans le second et ses pères, jusqu'à découverte d'une fonction ou épuisement de la liste. Cette recherche en *profondeur d'abord* a aussi lieu si l'un des packages pères de <pkgc1> est une liste. Ce méchanisme permet d'implanter un *héritage multiple* des méthodes. Dans ce cas le second argument permet d'arrêter la recherche pour chaque package de la liste.

GETFN peut être défini en Lisp de la manière suivante :

```
(DE GETFN (fn p1 . p2)
    (IF p2 (SETQ p2 (CAR p2))
           (SETQ p2 '0))
    (GETFN-AUX fn p1 p2))

(DE GETFN-AUX (fn p1 p2)
    (COND  ((EQ p1 p2) ()) ; arret sur p2
           ((EQ '|| p1)    ; la racine
            (GETFN1 fn p1))
          ((CONSP p1)      ; recherche multiple
            (OR (GETFN-AUX fn (CAR p1) p2)
                (GETFN-AUX fn (CDR p1) p2)))
           (T              ; le cas normal
            (OR (GETFN1 fn p1)
                (GETFN-AUX fn (PACKAGECELL p1) p2)))))
```

(GETFN2 <fn> <pkgc1> <pkgc2>) [SUBR à 3 arguments]

Cette fonction permet de rechercher une fonction de nom <fn> dans le package produit (<pkgc1> . <pkgc2>) ou bien dans l'un des packages produits supérieurs à ce package. La hiérarchie des packages produit est bâtie à partir de la hiérarchie des packages en utilisant l'ordre lexicographique. La remontée hiérarchique dans les packages produits est toujours arrêtée avant d'atteindre le package racine pour l'un des deux membres du produit.

Les packages supérieurs au package produit (#:A:B . #:C:D) sont, dans l'ordre :

```
(#:A:B . #:C:D) < (#:A:B . C) < (A . #:C:D) < (A . B)
```

5.3.2 Invocation des méthodes

La combinaison des fonctions de calcul de type et des fonctions de recherche de méthode permet d'implanter des fonctions de transmission de messages.

(SEND <symb> <o> <p1> ... <pP>) [SUBR à N arguments]
(SEND-ERROR <symb> <reste>) [SUBR à 2 arguments]

Cette fonction recherche et invoque la méthode <symb> associée au type de l'objet <o>. Cette méthode doit être une fonction Lisp à P+1 arguments. Elle est appelée avec comme arguments la liste formée de l'objet <o> et des P arguments <p1>...<pP>.

La méthode recherchée est une fonction Lisp de nom <symb> dans le package qui est le type de l'objet <o> ou bien dans l'un de ses packages pères à l'exclusion du package racine. Si aucune méthode n'est trouvée cette fonction appelle la fonction SEND-ERROR qui, par défaut, déclenche l'erreur ERRUDM dont le libellé est :

```
** <fn> : méthode indéfinie : <l>     ou bien
** <fn> : undefined method : <l>
```

dans lequel <l> est la liste des arguments fournis à SEND et <fn> le nom de la fonction ayant provoqué l'erreur (ici SEND).

SEND peut être défini en Lisp de la manière suivante :

```
(DE SEND (m o . larg)
    (LET ((fn (GETFN (TYPE-OF o) m ||)))
         (IF fn
             (APPLY fn o larg)
             (FUNCALL 'SEND-ERROR m (CONS o larg)))))
```

SEND-ERROR peut être défini en Lisp de la manière suivante :

```
(DE SEND-ERROR (m rest)
    (ERROR 'SEND 'ERRUDM (CONS m rest)))
```

Définition de la méthode HELLO des objets de type FIX

```
ex: ? (DE #:FIX:HELLO (n)
    ?     (REPEAT n (PRINT "Hello")))
    = #:FIX:HELLO
```

Recherche et invocation de cette méthode

```
? (SEND 'HELLO 3)
Hello
Hello
Hello
= ()
```

Définition de la méthode HELLO pour les objets de type HOMME

```
? (DE #:HOMME:HELLO (hom)
?     (PRINT "Je suis un homme")
?     (PRINT "je pese " (#:HOMME:POIDS hom) "kilos"))
= #:HOMME:HELLO
? (SEND 'HELLO (#:HOMME:MAKE))
Je suis un homme
je pese 75 kilos
```

```
= kilos
? (SEND 'HELLO (#:HOMME:CHERCHEUR:MAKE))
Je suis un homme
je pese 75 kilos
= kilos
```

(SEND-SUPER <type> <symb> <o> <p1> ... <pN>) [SUBR à N args]

Cette fonction a le même comportement que la fonction SEND à la différence que la méthode est recherchée à partir du package <type> exclus, et non à partir du type de l'objet <o>. <type> doit être un des packages pères du type de l'objet <o> sinon une erreur se déclenche.

SEND-SUPER est principalement utilisée à l'intérieur d'une méthode pour invoquer une méthode de même nom définie à un niveau supérieur dans la hiérarchie des types.

Dans l'exemple ci-dessous la méthode DECRIT est définie pour les objets de type HOMME. On redéfinit la méthode DECRIT pour les objets du sous-type #:HOMME:CHERCHEUR comme invoquant d'abord la méthode DECRIT au niveau juste supérieur à #:HOMME:CHERCHEUR dans la hiérarchie des types, puis comme effectuant un traitement spécifique aux objets de type #:HOMME:CHERCHEUR.

```
ex : ? (DE #:HOMME:DECRIT (h)
     ?     (PRINT "un homme de " (#:HOMME:AGE h)   "ans, qui pèse "
     ?            (#:HOMME:POIDS h) " kilos"))
     = #:HOMME:DECRIT
     ?
     ? (DE #:HOMME:CHERCHEUR:DECRIT (c)
     ?     (SEND-SUPER '#:HOMME:CHERCHEUR 'DECRIT c)
     ?     (PRINT "qui a écrit " (LENGTH (#:HOMME:CHERCHEUR:ARTICLE c))
     ?             "articles"))
     = #:HOMME:CHERCHEUR:DECRIT
     ?
     ? (SEND 'DECRIT (#:HOMME:CHERCHEUR:MAKE))
     un homme de () ans, qui pèse 75 kilos
     qui a écrit 0 articles
     = articles
```

(CSEND <fndef> <symb> <o> <p1> ... <pN>) [SUBR à N arguments]

Cette fonction est identique à la fonction SEND mais si la recherche de la méthode échoue, la fonction <fndef> est invoquée au lieu de provoquer une erreur, avec tous les arguments de CSEND sauf le premier.

Cette fonction est particulièrement utile pour redéfinir la recherche des méthodes. On peut par exemple définir des méthodes par défaut dans un package spécial, ou bien implanter un autre héritage multiple.

CSEND peut être défini en Lisp de la manière suivante :

```
(DE CSEND (default s o . largs)
    (LET ((fun (GETFN (TYPE-OF o) s '||)))
         (IF fun
             (APPLY fun o largs)
             (APPLY default s o largs))))
```

Utilisation du package * comme un défaut

```
ex : ? (DE DEFAULT (m . para)
     ?     (APPLY (GETFN '* m '||) para))
     = DEFAULT
     ?
     ? (DE #:*:DECRIT (o)
     ?     (PRINT "un objet inconnu"))
     = #:*:DECRIT
     ?
     ? (CSEND 'DEFAULT 'DECRIT 12)
     un objet inconnu
     = un objet inconnu
     ? (CSEND 'DEFAULT 'DECRIT (#:HOMME:MAKE))
     un homme de () ans, qui pese 75 kilos
     = kilos
```

(SEND2 <symb> <o1> <o2> <p1> ... <pN>) [SUBR à N arguments]

Cette fonction recherche et invoque la méthode de nom <symb> associée au type produit des types des objets <o1> et <o2>. Cette méthode est recherchée par la fonction GETFN2 (voir cette fonction). La fonction éventuellement trouvée est invoquée avec comme arguments <o1>, <o2> et les N arguments <p1> ... <pN>.

Définition de méthodes sur des types produits

```
ex : ? (de #:(a . a):foo (o1 o2)
     ?     'aa)
     = #:(a . a):foo
     ?
     ? (de #:(a . #:a:b):foo (o1 o2)
     ?     'ab)
     = #:(a . #:a:b):foo
     ?
     ? (de #:(#:a:b . a):foo (o1 o2)
     ?     'ba)
     = #:(#:a:b . a):foo
     ?
     ? (de #:(a . #:a:c):foo (o1 o2)
     ?     'ac)
     = #:(a . #:a:c):foo
     ?
```

Invocation de ces méthodes

(Les objets sont ici des doublets étiquetés)

```
? (send2 'foo '#(a) '#(a))
= aa
? (send2 'foo '#(a) '#(#:a:b))
= ab
? (send2 'foo '#(#:a:b) '#(a))
= ba
? (send2 'foo '#(#:a:b) '#(#:a:b))
= ba
? (send2 'foo '#(#:a:c) '#(#:a:b))
= ab
? (send2 'foo '#(#:a:b) '#(#:a:c))
```

```
= ba
? (send2 'foo '#(#:a:c) '#(#:a:c))
= ac
```

5.3.3 Les méthodes prédéfinies du système

Le_Lisp utilise de manière interne la fonction SEND pour l'impression et l'évaluation des objets Lisp.

L'impression des objets Lisp implantés sous forme de vecteurs typés, chaînes de caractères typés et doublets étiquetés provoque l'envoi du message PRIN à ces objets. Ce message doit provoquer l'impression de l'objet dans le tampon de sortie, sans terminaison de ligne. Si aucune méthode n'existe pour imprimer ces objets le format d'impression par défaut est utilisé.

Les méthodes d'impression sont utilisées par toutes les fonctions d'impression (PRINT, PRIN, PRINFLUSH, EXPLODE, etc.). Elles seront donc utilisées, par exemple, lors de l'impression des paramètres d'une fonction tracée, lors de l'édition des fonctions, ou lors de l'impression des messages d'erreur.

Attention dans ce dernier cas : si la méthode d'impression provoque une erreur il est fort probable que le traitement de cette erreur réutilise la même méthode d'impression et provoque derechef la même erreur. On risque ainsi d'obtenir la fameuse erreur bouclante "error in error handler". Il est donc conseillé de mettre au point les méthodes d'impression sous un autre nom et de ne les nommer PRIN que lorsqu'elles semblent correctes.

Les méthodes d'évaluation sont utilisées pour les vecteurs typés. L'évaluation d'un vecteur typé provoque la transmission du message EVAL à ce vecteur. Si aucune méthode EVAL n'existe pour ce vecteur l'évaluation du vecteur retourne le vecteur lui même. Sinon c'est la valeur rendue par la méthode EVAL du vecteur qui est rendue en valeur.

```
ex:  ; Les objets de type HOMME vont s'imprimer sous la forme
     ; #H(<age> <poids>)
     ; La #-macro H permet de relire cette notation.
     ? (DE #:HOMME:PRIN (homme)
     ?     (PRIN "#H(" (#:HOMME:AGE homme) " " (#:HOMME:POIDS homme) ")"))
     = #:HOMME:PRIN
     ?
     ? (NEW 'HOMME)
     = #H(() 75)
     ?
     ? (DEFSHARP |H| ()
     ?      (LET ((age-poids (READ))
     ?            (homme (NEW 'HOMME)))
     ?           (#:HOMME:AGE (CAR age-poids))
     ?           (#:HOMME:POIDS (CADR age-poids))
     ?           homme))
     = #:SHARP:H
     ?
     ? #H(23 75)
     = #H(23 75)
```

CHAPITRE 6

Les Entrées/Sorties

Les entrées/sorties de base s'effectuent au niveau du caractère, de la ligne ou de la S-expression Lisp, sur des flux séquentiels.

6.1 Introduction

Cette introduction présente les principaux concepts utilisés pour la description des fonctions d'entrée/sorties.

6.1.1 Les caractères

Les caractères peuvent être représentés de deux façons :
- sous la forme d'un nombre entier représentant le code interne de ce caractère ‹cn›.
- sous la forme d'un atome (symbole, chaîne, ou nombre entier) ‹ch›. Si l'atome est un symbole ou une chaîne, le premier caractère du symbole ou de la chaîne est utilisé. Si c'est un nombre le premier chiffre de la représentation du nombre dans la base de sortie courante (voir la fonction OBASE) est utilisé.

Le_Lisp utilise l'une ou l'autre de ces représentations. Les fonctions qui utilisent la première représentation sont en général suffixées par CN et celles utilisant la seconde par CH. Le code interne utilisé dépend des systèmes. C'est en général le code ASCII. Les codes internes sont compris entre 0 (inclus) et 256 (exclus).

Il est toujours plus efficace et plus économique en mémoire d'utiliser la représentation sous forme de codes internes. L'utilisation des #-macros #/, #^ et #\ rend de plus cette représentation presque aussi lisible que la représentation sous forme d'atomes mono-caractères.

6.1.2 Les suites de caractères

Les suites de caractères, peuvent être représentées sous trois formes :
- chaînes de caractères ‹string›.
- listes de codes internes ‹lcn›.
- listes d'atomes mono-caractères ‹lch›.

Les fonctions qui utilisent la première représentation sont suffixées par STRING. Elles sont en général plus efficaces et plus économiques en place mémoire.

6.1.3 Les lignes

Usuellement les fonctions de lecture et d'impression sont utilisées pour manipuler des S-expressions. Dans ce cas le format est libre et la fin de ligne physique importe peu. Il est cependant parfois nécessaire d'effectuer des lectures ou des impressions caractère par caractère ou ligne par ligne. Les marques de *fin de ligne* sont alors importantes.

Lorsque l'on lit caractère par caractère, au moyen des fonctions READCN/READCH ou PEEKCN/PEEKCH, la fin de ligne est matérialisée par les deux caractères consécutifs #\CR et #\LF. Ceci est valable aussi bien sur un terminal que sur un fichier.
Lorsque l'on lit ligne par ligne, au moyen des fonctions READSTRING ou READLINE, la fin de ligne physique n'est jamais matérialisée dans le résultat obtenu.

Lors des impressions, des marques de fin de ligne sont insérées explicitement par l'utilisateur (fonctions PRINT, TERPRI et TYNEWLINE), et automatiquement par le système lorsque l'on atteint la marge droite du tampon de sortie. Les marques insérées dépendent bien sûr des systèmes et du type du terminal utilisé. Elles ne correspondent pas nécessairement à des caractères supplémentaires (sortie sur BITMAP, sortie sur des fichiers à enregistrements).
On peut positionner la marge droite plus loin que la fin du tampon. Dans ce cas le système n'ajoute pas de marque de fin de ligne automatiquement (voir la fonction RMARGIN).

6.1.4 Les canaux

Toutes les entrées/sorties s'effectuent sur des flux séquentiels, connectés à des terminaux ou des fichiers. Le_Lisp connaît un flux d'entrée et un flux de sortie courants, positionnés par les deux fonctions INCHAN et OUTCHAN. Toutes les lectures ont lieu sur le flux d'entrée courant, toutes les écritures sur le flux de sortie courant.

6.1.5 Les interruptions programmables des entrées/sorties

Les fonctions d'entrées/sorties les plus utilisées (READ, PRINT, etc.) lisent et écrivent des caractères dans des tampons; elles ne font pas d'accès réel au terminal ou sur disque.

Des fonctions de plus bas niveau permettent de gérer les tampons d'entrée et de sortie. Ce sont les interruptions programmables BOL, EOL, et FLUSH. Elles sont aisément redéfinissables par l'utilisateur par la mécanique de gestion des interruptions programmables. Ceci qui permet d'étendre le système d'entrées/sorties : lecture à partir d'une liste de chaînes de caractères, écriture dans plusieurs fichiers simultanément, etc. Ces fonctions très puissantes sont néanmoins rarement utilisées explicitement.

6.2 Les Fonctions d'Entrée de Base

(READ) [SUBR à 0 argument]

lit la S-expression suivante de type quelconque (atome ou liste) sur le flux d'entrée courant et la retourne en valeur. READ est la principale fonction de lecture et permet de lire n'importe quel objet Lisp quelque soit sa taille. Son fonctionnement détaillé est donné dans la section suivante.

Si une erreur de syntaxe est détectée, l'erreur Lisp ERRSXT est déclenchée dans laquelle le nom de la fonction qui a provoqué l'erreur est imprimé ainsi que le message décrivant le type de l'erreur :

```
** <fn> : erreur de syntaxe : <msg>
** <fn> : syntax error : <msg>
```

Voici la liste des messages :

1 : liste de IMPLODE trop courte
2 : chaîne de caractères trop longue
3 : symbole trop long
4 : unité syntaxique débutant par) ou .
5 : symbole spécial trop long
6 : mauvaise utilisation des packages
7 : mauvaise construction pointée : . <s>)
8 : la liste du REREAD ne contient pas que des caractères
9 : la liste du IMPLODE ne contient pas que des caractères
10 : mauvaise valeur de splice-macro
11 : fin de fichier durant une lecture
12 : mauvaise utilisation du backquote

Le plus souvent cette erreur provient d'une mauvaise utilisation des paires pointées ou d'un P-NAME trop long (de plus de 128 caractères), dû à l'oubli du caractère | ou ".

(STRATOM <n> <strg> <i>) [SUBR à 3 arguments]

réalise une lecture partielle dans la chaîne de caractères <strg> sur <n> caractères. STRATOM retourne un symbole ou un nombre (entier ou flotant), si l'indicateur <i> est égal à () et retourne toujours un symbole si l'indicateur <i> est différent de (). Cette fonction permet d'atteindre la partie interne du lecteur Le_Lisp qui réalise les conversions numériques et la recherche des symboles dans l'OBLIST et ne peut donc pas être décrite simplement en Le_Lisp. Elle est principalement utilisée pour écrire d'autres lecteurs en Le_Lisp.

```
ex : (STRATOM 3 "abcdef" ())      -> abc
     (STRATOM 3 "01234" ())       -> 12
     (STRATOM 2 "01234" t)        -> |012|
     (STRATOM 4 " () " ())        -> | () |
     (STRATOM 5 "00012.34" ())    -> 12
     (STRATOM 6 "00012.34" ())    -> 12.
     (STRATOM 7 "00012.34" ())    -> 12.3
```

(READSTRING) [SUBR à 0 argument]

lit la ligne suivante du flux d'entrée et retourne celle-ci sous la forme d'une chaîne de caractères. Cette chaîne ne contient pas les éventuels délimiteurs de fin de ligne. READSTRING permet de réaliser des lectures portables au niveau de la ligne. Elle est utilisée pour réaliser à peu de frais les interfaces entre programme et utilisateur.

ATTENTION : cette fonction n'effectue aucune conversion de caractères majuscules en caractères minuscules.

READSTRING peut être défini en Lisp de la manière suivante :

```
(DE READSTRING ()
    (LET ((l) (c))
         (WHILE (NEQ (SETQ c (READCN)) #\CR) ; lit jusqu'à #\CR
                (NEWL l c))
         (READCN)                          ; saute #\LF
         (STRING (NREVERSE  l))))
```

(READLINE) [SUBR à 0 argument]

comme dans la fonction précédente lit la ligne suivante du flux d'entrée, mais retourne celle-ci sous forme d'une liste de codes internes. Cette liste ne contient pas les éventuels délimiteurs de fin de ligne. Cette fonction est plus chère en ressource mémoire que la fonction précédente, mais fournit parfois une interface plus pratique.

READLINE peut être défini en Lisp de la manière suivante :

```
(DE READLINE ()
    (EXPLODE (READSTRING)))

ex : ? (MAPCAR 'ASCII (READLINE))
                              ; appelle la lecture d'une ligne
  Le gai rossignol            ; ligne terminée par Return
  = (L e   g a i   r o s s i g n o l )
                              ; liste des caractères lus

; pour fabriquer une suite de mots à partir d'une ligne
; qui ne contient pas de caractères spéciaux ( ) . [ ]

? (IMPLODE (APPEND '#"(" (READLINE) '#")")))
Le merle moqueur              ; ligne terminée par Return
= (LE MERLE MOQUEUR)          ; liste des atomes lus
```

(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 (c'est-à-dire d'un nombre). Cette fonction n'effectue pas de conversion automatique des caractères majuscules en caractères minuscules. Elle ne peut pas être simplement décrite en Lisp.

(READCH) [SUBR à 0 argument]

est identique à la fonction précédente, mais retourne un atome mono-caractère.

READCH peut être défini en Lisp de la manière suivante :

```
(DE READCH ()
    (ASCII (READCN)))
```

(PEEKCN) [SUBR à 0 argument]

retourne en valeur le caractère suivant du flux d'entrée d'une manière identique à la fonction READCN, toutefois, ce caractère n'est pas véritablement lu mais seulement *consulté*. Il en résulte que des appels successifs de la fonction PEEKCN retournent toujours le même résultat. PEEKCN est utilisé pour tester le caractère suivant du flux d'entrée avant de le lire véritablement.

PEEKCN peut être défini en Lisp de la manière suivante :

```
(DE PEEKCN ()
    (LET ((cn (READCN)))
         (REREAD (LIST cn))))
```

(PEEKCH) [SUBR à 0 argument]

est identique à la fonction précédente, mais retourne un atome mono-caractère.

PEEKCH peut être défini en Lisp de la manière suivante :

```
(DE PEEKCH ()
    (ASCII (PEEKCN)))
```

(REREAD ‹lcn›) [SUBR à 1 argument]

rajoute la liste des caractères ‹lcn› en tête du flux d'entrée. Ces caractères seront lus au prochain appel d'une des fonctions de lecture vues précédemment (READ, READSTRING, READCN ...) avant de continuer à lire sur le véritable flux d'entrée. Cette fonction est particulièrement puissante si elle est utilisée à l'intérieur même d'un READ ou d'un macro-caractère. REREAD retourne la liste des caractères qui ont été ajoutés en tête du flux d'entrée.

```
(PROGN (REREAD (LIST #/' #/a #\SP)) (READ))   -> (QUOTE A)
```

Pour imprimer la liste des caractères à relire

```
(PRINT (REREAD ()))
```

(READ-DELIMITED-LIST ‹cn›) [SUBR à 1 argument]

lit une liste de S-expressions jusqu'à l'occurrence d'un caractère de code interne ‹cn›. Le type du caractère ‹cn› doit être CMACRO, CSPLICE ou CMSYMB. Cette fonction est particulièrement utile à l'intérieur d'un macro caractère pour lire un ensemble de S-expression.

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

```
(DE READ-DELIMITED-LIST (cn)
    (LET ((RES ()))
         (UNTIL (EQ (VALIDCN) cn)
                (NEWL RES (READ)))
         (READCN)          ; lit le caractère cn
         (NREVERSE RES)))

(DE VALIDCN ()
    (SELECTQ (TYPECN (PEEKCN))
```

```
                ((CSEP CECOM)
                      (READCN)
                      (VALIDCN))
                (CBCOM
                      (READCN)
                      (UNTIL (EQ (TYPECN (READCN)) 'CECOM))
                      (VALIDCN))
                (T (PEEKCN))))
ex : ; Après les deux définitions de macro-caractères
     (DMC |[| ()
          (CONS 'LIST (READ-DELIMITED-LIST #/])))
     (DMC |]| ()
          (ERROR '|]| 'ERRSXT "doit apparaître après un ["))
     [1 2 3 4]          est lu comme    (list 1 2 3 4)
     ['PRINT [A B C] 'FOO]     " "      (list 'print (list a b c) 'foo)
     [1 2 ; les commentaires sont bien ignorés
        3 ; là aussi
        ]                      " "      (list 1 2 3)
```

(TEREAD) [SUBR à 0 argument]

vide tous les caractères en attente dans le tampon d'entrée et dans les tampons internes du lecteur sur le canal d'entrée courant. Après un appel à cette fonction l'interruption programmable BOL sera immédiatement déclenchée à la première lecture. Cette fonction est particulièrement utile pour supprimer les caractères séparateurs présents dans les tampons internes du lecteur juste après une lecture par la fonction READ. Cette fonction ne peut être décrite en Lisp en raison de l'existence de tampons internes au lecteur.

```
ex : ? (DE FOO ()
     ?      (PRINT "read->" (READ))
     ?      (PRINT "readstring->" (READSTRING))
     ?      T)
     = FOO
     ? (FOO)
     ? hello
     read->hello
     readstring->
     = t
     ? (DE BAR ()
     ?      (PRINT "read->" (READ))
     ?      (TEREAD)
     ?      (PRINT "readstring->" (READSTRING))
     ?      T)
     = BAR
     ? (BAR)
     ? hello
     read->hello
     ? hello world
     readstring->hello world
     = t
```

6.2.1 A l'intérieur de READ

La variable et la fonction suivante ne sont significatives qu'à l'intérieur d'un READ. Elles ne seront utilisées que dans un macro-caractère ou bien au cours du traitement des interruptions programmables de lecture (BOL et EOF).

#:SYSTEM:IN-READ-FLAG [Variable]

cette variable interne, qui est positionnée par la fonction READ, permet de savoir si le lecteur Lisp se trouve au milieu d'un READ (c'est-à-dire si un objet Lisp de n'importe quel type a commencé à être lu).

Cette variable est par exemple utilisée dans la fontion EOF pour tester si une fin de fichier apparaît durant la lecture d'un objet Lisp (voir la fonction EOF).

(CURREAD) [SUBR à 0 argument]

retourne la liste en cours de lecture ou bien () si le lecteur n'est pas en train de lire une liste. La liste retournée contient un élément supplémentaire en tête, toujours égal au symbole CURREAD.

```
ex : (DMS |%| ()
          ; juste pour comprendre CURREAD
          (PRINT (CURREAD))
          ())
```

La lecture de :

```
'(A B % C (% D %) % E %)
```

provoquera à la lecture les impressions

```
(CURREAD A B)
(CURREAD)
(CURREAD D)
(CURREAD A B C (D))
(CURREAD A B C (D) E)
```

redéfinition de % comme opérateur binaire infixe

```
(DMS |%| ()
     (IF (CONSP (CDR (CURREAD)))
         (RPLACD (CURREAD) (CONS '|%| (CDR (CURREAD))))
         (ERROR '|%| 'ERRSXT ()))
     ())
```

la lecture de :

```
'(A % (B % C))
```

produira :

```
(% A (% B C))
```

6.3 L'utilisation du Terminal en Entrée

A chaque demande de ligne sur le terminal, le système imprime une *chaîne d'invite* qui peut être modifiée au moyen de la fonction PROMPT et qui par défaut est la chaîne "? ". Il lit ensuite une ligne au terminal. L'utilisateur termine la ligne en frappant l'un des deux caractères #\CR ou #\LF. C'est la fonction BOL couramment active (voir les interruptions programmables des entrées/sorties) qui gère le dialogue avec l'utilisateur.

Le_Lisp fait l'écho de chaque caractère au fur et à mesure de la lecture. Les caractères de contrôle ASCII (c'est-à-dire les caractères entrés en pressant simultanément la touche CTRL et le caractère) sont imprimés *à la DEC* (c'est-à-dire ^ suivi du caractère).

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 : efface toute la ligne entrée

Un puissant éditeur de lignes, EDLIN est fourni avec le système. Il permet de gérer un historique des commandes et de compléter les noms des symboles partiellement tapés par l'utilisateur. On lance cet éditeur par la fonction EDLIN (voir le chapitre 17). La commande ESC-? donne la documentation en ligne d'EDLIN.

Les variables globales système #:SYSTEM:REAL-TERMINAL-FLAG et #:SYSTEM:LINE-MODE-FLAG permettent de supprimer l'écho et l'éditeur de ligne minimum.

#:SYSTEM:REAL-TERMINAL-FLAG [Variable]

si cet indicateur est positionné (l'option par défaut dépend des systèmes), l'éditeur de ligne et l'écho des caractères est activé. S'il ne l'est pas, l'éditeur est inactivé et le système suppose qu'un autre processeur réalise l'écho et l'édition de ligne.

#:SYSTEM:LINE-MODE-FLAG [Variable]

si cet indicateur est positionné (l'option par défaut dépend des systèmes) l'éditeur de ligne suppose en plus que la lecture peut uniquement se faire ligne à ligne. Dans ce cas le comportement des fonctions TYI et TYS est imprévisible, le système utilise la fonction TYINSTRING pour lire une ligne au terminal.

(PROMPT <strg>) [SUBR à 0 ou 1 argument]

permet de connaître ou de modifier la *chaîne d'invite* imprimée par le système quand celui-ci est en attente de caractères au terminal. Cette fonction peut être utilisée avec la structure de contrôle WITH pour changer la chaîne d'invite localement. Par défaut cette chaîne vaut: "? ".

```
ex : ? ()          ; la chaîne d'invite normale de lecture
     = ()
     ? (prompt "> ")
     = >
     > ()          ; elle est changée
     = ()
     > (with ((prompt "allo>"))   ; avec la chaîne d'invite "allo>"
     >       (toplevel))          ; un coup de toplevel
```

```
allo>(cons 1 2)
= (1 . 2)
= ()
>  ; retour à la chaîne "> "
```

6.4 La Lecture Standard

La lecture des S-expressions Lisp par la fonction READ s'effectue en format *libre* : chaque élément syntaxique peut être encadré d'un ou plusieurs délimiteurs (espaces, fins de lignes, commentaires).

Le lecteur Lisp est paramétré par une *table de lecture* qui indique quels sont les caractères spéciaux (délimiteurs de listes, de chaînes et de commentaires, macro-caractères, etc.). Les autres caractères sont dits caractères de type PNAME, ou caractères normaux.

6.4.1 La lecture des symboles

Un symbole est une suite de caractères de type PNAME qui ne représente pas un nombre (entier ou flottant).

Lorsqu'un symbole doit contenir des caractères spéciaux, il faut encadrer tous les caractères du symbole avec le caractère délimiteur de symboles (la barre de valeur absolue | par défaut).

A la lecture, un symbole ne peut pas dépasser 128 caractères. On peut néanmoins créer des symboles de grande longueur en utilisant la fonction CONCAT.

Les packages des symboles sont spécifiés de trois manières, qui sont décrites plus bas :
- la #-macro #:
- le macro-caractère :
- un caractère de type délimiteur de package

#:SYSTEM:READ-CASE-FLAG [Variable]

cette variable permet de contrôler la conversion automatique des caractères majuscules en leur équivalent minuscule. Si la valeur de cette variable est (), et c'est l'option par défaut, la conversion est automatique, si elle est différente de () aucune conversion n'est réalisée.

```
ex : car            correspond au symbole    car
     LONGTRESLONGATOME  "       "     "      longtreslongatome
     |Foo Bar|        "       "     "      Foo Bar
     #:Foo:Bar        "       "     "      #:foo:bar
     #:foo:|Bar()|:gee "      "     "      #:foo:Bar():gee
```

Mais si on lit les symboles de la manière suivante :

```
(LET ((#:SYSTEM:READ-CASE-FLAG T))
     (READ))

CdR            correspond au symbole       CdR
|Ach|              "       "     "         Ach
```

6.4.2 La lecture des chaînes de caractères

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. S'il y a une fin de ligne au milieu d'une chaîne de caractères celle ci contiendra les deux caractères CR et LF. A la lecture, une chaîne ne peut actuellement dépasser 256 caractères. La fonction MAKESTRING permet de créer des chaînes plus longues. Il n'y a jamais de transcodage des caractères majuscules en caractères minuscules durant la lecture des chaînes.

On peut préciser le type d'une chaîne de caractères typée au moyen de la #-macro #:.

```
ex :  "hello"                contient les caractères          hello
      "avec ""guillemets"""      "     "       "       avec "guillemets"
      #:schiz:"parano"       est une chaine typée de type     schiz
                             quicontient les caractères       parano
```

6.4.3 La lecture des nombres entiers et rationnels

Les nombres entiers sont représentés par une suite de chiffres de longueur quelconque, éventuellement précédée d'un signe, + ou -. Les nombres rationnels sont représentés par un nombre entier immédiatement suivi du caractère / immédiatement suivi d'un second nombre entier.

Les caractères représentant des chiffres dépendent de la base de lecture. Voir la fonction IBASE. Par défaut la base de lecture est 10, les chiffres sont donc les chiffres usuels.

La construction des nombres à partir de leur représentation textuelle utilise l'arithmétique générique. Ceci veut dire que si le nombre fourni au système n'est pas un petit entier 16 bits, la valeur lue dépend de l'arithmétique générique utilisée. En standard l'artihmétique générique convertit en nombres flottants les nombres entiers trop grands et les nombres rationnels qui ne se réduisent pas à un nombre entier.

```
ex :  12          correspond à l'entier          12
      -1              "       "     "            -1
      +0              "       "     "             0
      12/4            "       "     "             3
      12/-4           "       "     "            -3
```

Avec l'arithmétique générique standard

```
       32767      correspond à l'entier        32767
      -32767          "       "     "        -32767
       32768      correspond au flottant       32768.
      -32769          "       "     "        32769.
      -32768      correspond à l'innomable     #$8000
       12/5       correspond au flottant        2.4
      -12/5           "       "     "         -2.4
       12/-5          "       "     "         -2.4
      -12/-5          "       "     "          2.4
```

Avec l'arithmétique rationnelle : (cf Chapitre 10)

```
      30424324324  correspond à l'entier        30424324324
      12/5         correspond au rationnel      12/5
      45574512/56     "       "     "          5696814/7
```

(IBASE <n>) [SUBR à 0 ou 1 argument]

permet de consulter si l'argument <n> n'est pas fourni, ou de modifier la valeur de la base de conversion des nombres en entrée. La base affecte la lecture des nombres entiers et rationnels uniquement. La lecture des nombres flottants a toujours lieu en base 10. Cette base doit être comprise entre 2 et 36. Cette fonction est la contrepartie de la fonction OBASE.

```
? (IBASE)
= 10
? (IBASE 16)
= 16
? FFF           ; n'est plus un symbole
= 4095
? (IBASE A)     ; pour repasser en base 10
= 10
? (IBASE 36)    ; gag!
= 36
? IBASE         ; est un nombre! ainsi que tous les autres symboles
= 23902         ; ne contenant que des lettres ou des chiffres.
?               ; Comment remettre la base d'entrée à 10 ?!?
? (|ibase| a)   ; élémentaire mon cher
= 10
```

Il existe d'autres manières de lire des nombres en particulier grâce aux #-MACRO #$, #% et #<base>r qui permettent de décrire des nombres dans n'importe quelle base.

6.4.4 La lecture des nombres flottants

Les nombres flottants sont représentés par une suite de chiffres décimaux (la partie entière), optionnellement suivie d'un point et d'une autre suite de chiffres décimaux (la partie fractionnaire), optionnellement suivie du caractère E ou e suivi d'un nombre entier signé (la partie exposant). La partie entière ou la partie fractionnaire peuvent manquer mais pas les deux en même temps. La partie exposant peut manquer.

```
ex :    1.50          correspond au nombre     1.50
        1e+0                                   1.
        10e+1                                  100
        -10e+1                                 -1.
        0e+1                                   0
        .45                                    .45
        12.34e-3                               0.1234

        e-            est un symbole!
```

6.4.5 La lecture des listes

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 à (A B C . D)
```

```
((A) . (B))          "      "   ((A) B)
```

A la fin 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) (+ n 2))))))    ; est lu sans erreur
```

6.4.6 La lecture des vecteurs

Les vecteurs sont représentés par les deux caractères #[suivis des éléments du vecteur séparés par des blancs, suivis du caractère]. La lecture des vecteurs s'effectue au moyen de la #-macro #[. On peut préciser le type d'un vecteur en utilisant la #-macro #:.

```
ex : #[a b c]            est un vecteur à trois éléments a, b , c
     #[]                 est un vecteur vide
     #:type:#[12 23 40]  est un vecteur typé
```

6.4.7 La lecture des commentaires

Il est possible d'insérer des commentaires, qui sont considérés comme des délimiteurs au cours de la lecture. Un commentaire est une suite de caractères quelconques précédée d'un caractère de type début de commentaires 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 caractère fin de commentaires le caractère Return. Par défaut tout commentaire s'arrêtera donc en fin de ligne.

```
(DE FOO (N              ; le nombre
         L)             ; la liste
    (IF (= N 0)         ; plus rien à faire
        ...
```

Il est également possible d'écrire des commentaires multilignes au moyen de la #-macro |. Ces commentaires multilignes peuvent s'imbriquer.

```
(DE FOO #|le nom|# (n)
     #|  ce commentaire
         tient sur plusieurs
         lignes et contient #|
                    d'autres commentaires |#
         qui s'imbriquent à merveille |#
     (+ n n))
```

6.4.8 Les types des caractères

L'analyseur lexical Le_Lisp (c'est-à-dire la fonction READ) utilise une *table de lecture* pour effectuer commodément son analyse. Cette table associe un type symbolique 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 :

CNULL [type de caractère]

Tous les caractères de ce type sont complètement ignorés à la lecture (ex : le caractère *Avance-bande* ou NULL ...).

CBCOM [type de caractère]

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 ayant ce type, le caractère point virgule ;.

CECOM [type de caractère]

Ce type de caractère sert à indiquer la fin d'un commentaire. Par défaut il n'existe qu'un seul caractère ayant ce type, le caractère #\CR.

CQUOTE [type de caractère]

Ce type de caractère est utilisé pour *quoter* n'importe quel autre caractère. Ceci consiste à lui donner implicitement le type CPNAME (le type des caractères normaux). Par défaut il n'existe pas de caractère de ce type, mais il est d'usage d'utiliser l'un des caractères slash **/** ou backslash **** en cas de besoin.

CLPAR [type de caractère]

Ce type de caractère (la parenthèse ouvrante (par défaut) sert de caractère de début de liste.

CRPAR [type de caractère]

Ce type de caractère (la parenthèse fermante) par défaut) sert de caractère de fin de liste.

CDOT [type de caractère]

Ce type de caractère (le point . par défaut) sert à écrire les paires pointées s'il apparaît encadré de caractères de type CSEP.

CSEP [type de caractère]

Définit un caractère séparateur normal (ex : l'espace, la tabulation ...).

CPKGC [type de caractère]

Ce type de caractère fait office de caractère délimiteur de package. Par défaut il n'existe pas de caractère de ce type. Il est toutefois d'usage de prendre le caractère deux-points (:). L'utilisation de : comme caractère délimiteur de package entraîne les lectures suivantes, qui sont plus lisibles que l'utilisation de la #-macro #:.

```
? (TYPECN #/: 'cpkgc)
= cpkgc
? 'foo:bar
= #:foo:bar
? 'gee:muu:wizz
= #:gee:muu:wizz
? (SETQ #:sys-package:colon 'mon:package:prive)
= #:mon:package:prive
```

```
? ':bizarre
= #:mon:package:prive:bizarre
```

CSPLICE [type de caractère]

Ce type indique que le caractère est un splice-macro. Une fonction est associée au symbole 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). L'exemple classique d'un tel caractère est le macro caractère dièse #.

CMACRO [type de caractère]

Ce type indique que le caractère est un macro-caractère. Une fonction est associée au symbole 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).

CSTRING [type de caractère]

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 guillemet " .

CPNAME [type de caractère]

Définit un caractère normal pouvant être utilisé pour construire un P-NAME.

CSYMB [type de caractère]

Ce type de caractère indique le délimiteur de symboles spéciaux. Par défaut il n'y a qu'un seul caractère de ce type, le caractère valeur absolue.

CMSYMB [type de caractère]

Ce type indique que le caractère doit être lu comme un symbole mono-caractère et qu'il n'a pas besoin de délimiteur de symbole.

(TYPECN <cn> <symb>) [SUBR à 1 ou 2 arguments]

permet de connaître (si <symb> n'est pas fourni) ou de modifier (si <symb> est fourni) le type du caractère <cn>. Cette fonction retourne le type courant du caractère <cn> après modification éventuelle.

```
ex :  ; après la redéfinition syntaxique des
      ; caractères < et >,
         (TYPECN #/< 'CLPAR)    -> CLPAR
         (TYPECN #/> 'CRPAR)    -> CRPAR
      ; l'entrée :
         <CONS '(A> '<B)>
      ; sera lue :
         (CONS '(A) '(B))
```

(TYPECH <ch> <symb>) [SUBR à 1 ou 2 arguments]

est identique à la fonction précédente, mais le caractère est spécifié sous la forme d'un symbole mono-caractère.

Voici un programme qui imprime la table des types

```
(DE PRINTABLECH ()
    ; imprime la table des types des caractères
    (LET ((#:SYSTEM:PRINT-CASE-FLAG T)
          (I -1) (J 63)
          (AL (MAPCOBLIST (LAMBDA (X)
                (WHEN (GETPROP X '#:SHARP:VALUE)
                      (LIST (CONS (GETPROP X '#:SHARP:VALUE)
                                  X)))))))
         (REPEAT 64
                 (PRINTABLECH1 (INCR I) 8)
                 (PRINTABLECH1 (INCR J) 38)
                 (TERPRI))))

(DE PRINTABLECH1 (VAL POS)
    ; imprime le type d'un caractère
    (OUTPOS POS)
    (WITH ((OBASE 10)) (PRIN VAL) (OUTPOS (+ POS 4)))
    (WITH ((OBASE 16)) (PRIN "#$" VAL) (OUTPOS (+ POS 10)))
    (COND ((ASSOC VAL AL) (PRIN "#\" (CASSOC VAL AL)))
          ((< VAL 32) (PRIN "#^") (PRINCN (+ VAL 64)))
          (T (PRIN "#/") (PRINCN VAL)))
    (IF (>= (OUTPOS) (+ POS 17))
        (PRIN " ")
        (OUTPOS (+ POS 17)))
    (PRIN (TYPECN VAL)))
```

Voici l'impression de la table des types

```
  0   #$0   #\NULL CNULL      64  #$40  #/@   CPNAME
  1   #$1   #^A    CMACRO     65  #$41  #/A   CPNAME
  2   #$2   #^B    CPNAME     66  #$42  #/B   CPNAME
  3   #$3   #^C    CPNAME     67  #$43  #/C   CPNAME
  4   #$4   #^D    CPNAME     68  #$44  #/D   CPNAME
  5   #$5   #^E    CMACRO     69  #$45  #/E   CPNAME
  6   #$6   #^F    CMACRO     70  #$46  #/F   CPNAME
  7   #$7   #\BELL CPNAME     71  #$47  #/G   CPNAME
  8   #$8   #\BS   CSEP       72  #$48  #/H   CPNAME
  9   #$9   #\TAB  CSEP       73  #$49  #/I   CPNAME
 10   #$A   #\LF   CECOM      74  #$4A  #/J   CPNAME
 11   #$B   #^K    CSEP       75  #$4B  #/K   CPNAME
 12   #$C   #^L    CMACRO     76  #$4C  #/L   CPNAME
 13   #$D   #\CR   CECOM      77  #$4D  #/M   CPNAME
 14   #$E   #^N    CPNAME     78  #$4E  #/N   CPNAME
 15   #$F   #^O    CPNAME     79  #$4F  #/O   CPNAME
 16   #$10  #^P    CMACRO     80  #$50  #/P   CPNAME
 17   #$11  #^Q    CPNAME     81  #$51  #/Q   CPNAME
 18   #$12  #^R    CPNAME     82  #$52  #/R   CPNAME
 19   #$13  #^S    CPNAME     83  #$53  #/S   CPNAME
```

```
20  #$14  #^T     CPNAME      84  #$54  #/T     CPNAME
21  #$15  #^U     CPNAME      85  #$55  #/U     CPNAME
22  #$16  #^V     CPNAME      86  #$56  #/V     CPNAME
23  #$17  #^W     CPNAME      87  #$57  #/W     CPNAME
24  #$18  #^X     CPNAME      88  #$58  #/X     CPNAME
25  #$19  #^Y     CPNAME      89  #$59  #/Y     CPNAME
26  #$1A  #^Z     CPNAME      90  #$5A  #/Z     CPNAME
27  #$1B  #\ESC   CPNAME      91  #$5B  #/[     CMACRO
28  #$1C  #^\     CPNAME      92  #$5C  #/\     CPNAME
29  #$1D  #^]     CPNAME      93  #$5D  #/]     CMACRO
30  #$1E  #^^     CPNAME      94  #$5E  #/^     CMACRO
31  #$1F  #^_     CPNAME      95  #$5F  #/_     CPNAME
32  #$20  #\SP    CSEP        96  #$60  #/`     CMACRO
33  #$21  #/!     CMACRO      97  #$61  #/a     CPNAME
34  #$22  #/"     CSTRING     98  #$62  #/b     CPNAME
35  #$23  #/#     CSPLICE     99  #$63  #/c     CPNAME
36  #$24  #/$     CPNAME      100 #$64  #/d     CPNAME
37  #$25  #/%     CPNAME      101 #$65  #/e     CPNAME
38  #$26  #/&     CPNAME      102 #$66  #/f     CPNAME
39  #$27  #/'     CMACRO      103 #$67  #/g     CPNAME
40  #$28  #/(     CLPAR       104 #$68  #/h     CPNAME
41  #$29  #/)     CRPAR       105 #$69  #/i     CPNAME
42  #$2A  #/*     CPNAME      106 #$6A  #/j     CPNAME
43  #$2B  #/+     CPNAME      107 #$6B  #/k     CPNAME
44  #$2C  #/,     CMACRO      108 #$6C  #/l     CPNAME
45  #$2D  #/-     CPNAME      109 #$6D  #/m     CPNAME
46  #$2E  #/.     CDOT        110 #$6E  #/n     CPNAME
47  #$2F  #//     CPNAME      111 #$6F  #/o     CPNAME
48  #$30  #/0     CPNAME      112 #$70  #/p     CPNAME
49  #$31  #/1     CPNAME      113 #$71  #/q     CPNAME
50  #$32  #/2     CPNAME      114 #$72  #/r     CPNAME
51  #$33  #/3     CPNAME      115 #$73  #/s     CPNAME
52  #$34  #/4     CPNAME      116 #$74  #/t     CPNAME
53  #$35  #/5     CPNAME      117 #$75  #/u     CPNAME
54  #$36  #/6     CPNAME      118 #$76  #/v     CPNAME
55  #$37  #/7     CPNAME      119 #$77  #/w     CPNAME
56  #$38  #/8     CPNAME      120 #$78  #/x     CPNAME
57  #$39  #/9     CPNAME      121 #$79  #/y     CPNAME
58  #$3A  #/:     CMACRO      122 #$7A  #/z     CPNAME
59  #$3B  #/;     CBCOM       123 #$7B  #/{     CPNAME
60  #$3C  #/<     CPNAME      124 #$7C  #/|     CSYMB
61  #$3D  #/=     CPNAME      125 #$7D  #/}     CPNAME
62  #$3E  #/>     CPNAME      126 #$7E  #/~     CPNAME
63  #$3F  #/?     CPNAME      127 #$7F  #\DEL   CNULL
128 ....  255     CPNAME
```

6.4.9 Les macro-caractères

Un macro-caractère est un caractère quelconque auquel a été associée une fonction qui est lancée automatiquement à la lecture de ce caractère dans le flux d'entrée. Il existe deux types de macro-caractères : CMACRO et CSPLICE. Dans le premier type, la valeur retournée par la fonction associée à ce macro-caractère remplace le macro-caractère lu. Dans le deuxième type la valeur retournée par la fonction associée à ce macro-caractère doit être une liste (qui peut être vide). Cette liste sera ajoutée à l'expression en cours de lecture au moyen de la fonction NCONC. Si la valeur retournée par cette fonction n'est pas une liste l'erreur de syntaxe numéro 10 se déclenche. Tout caractère peut être utilisé comme macro-caractère.

(DMC <ch> () <s1> ... <sN>) [FSUBR]

sert à définir un macro-caractère du premier type. L'argument <ch> doit être un symbole mono-caractère. DMC associe à ce caractère une fonction sans liste de paramètres (cette liste vide est obligatoire à cette position) 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 être défini en Lisp de la manière suivante :

```
(DF DMC L
    ; fabrique la fonction associée
    (APPLY 'DE L))
    ; force le nouveau type de ce caractère
    (TYPECH (CAR L) 'CMACRO)
    ; et retourne en valeur le caractère.
    (CAR L))
```

(DMS <ch> () <s1> ... <sN>) [FSUBR]

sert à définir un macro-caractère du deuxième type. L'argument <ch> doit être un symbole mono-caractère. DMS associe à ce caractère une fonction sans liste de paramètres (cette liste vide est obligatoire à cette position) et un corps de fonction <s1> ... <sN>. DMS possède la même syntaxe que les autres fonctions de définition (DE, DMD, etc.) et retourne <ch> en valeur.

DMS peut être défini en Lisp de la manière suivante :

```
(DF DMS L
    ; fabrique la fonction associée
    (APPLY 'DE L))
    ; force le nouveau type de ce caractère
    (TYPECH (CAR L) 'CSPLICE)
    ; et retourne en valeur le caractère.
    (CAR L))
```

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]
    ; le caractère redevient normal
    (TYPECH ch 'CPNAME)
    ; destruction de la fonction
```

```
(REMFN ch)
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 ou DMS.

Il existe 13 macro-caractères prédéfinis :
- le macro-caractère apostrophe '
- le macro-caractère accent grave `
- le macro-caractère virgule ,
- le macro-caractère deux points :
- le macro-caractère flêxe ^
- le macro-caractère crochet ouvrant [
- le macro-caractère dièse #
- le macro-caractère clam !
- le macro-caractère charger un fichier ^L
- le macro-caractère charger un module ^A
- le macro-caractère éditer un fichier ^E
- le macro-caractère éditer une fonction ^F
- le macro-caractère paragrapher ^P

6.4.9.1 Les macro-caractères de base : ' ^ [

' apostrophe (quote) [Macro caractère]

L'apostrophe (quote) ' est le plus connu et le plus utilisé des macro-caractères. Placé devant une <S-expression> quelconque, il retourne la liste (QUOTE <S-expression>).

' peut être défini en Lisp de la manière suivante :

```
(DMC |'| ()
     (LIST 'QUOTE (READ)))
ex : '(A B)    est lu comme       (QUOTE (A B))
     ''A              "           (QUOTE (QUOTE A))
```

^ flêxe [Macro caractère]

ce macro-caractère permet de lire un symbole mono-caractère dont le P-NAME est un caractère contrôle.

^ peut être défini en Lisp de la manière suivante :

```
(DMC |^| ()
      (ASCII (LOGAND (READCN) 31)))
```

pour lire un symbole mono caractère de code ASCII 004

```
ex : ^D
```

[début de grand nombre [Macro caractère]

ce macro-caractère n'est définie qu'avec les bibliothèques des nombres à précision arbitraire et servent à décrire des nombres ne tenant pas sur une ligne ou utilisant des notations spécialisées.
ATTENTION : ce macro caractère ne doit pas être confondu avec l'écriture des vecteurs de S-expressions : #[...]

6.4.9.2 Le macro caractère accent grave (backquote)

` accent grave (backquote) [Macro caractère]
, ,. ,@ virgule [Macro caractère]

ce macro caractère sert à synthétiser des programmes qui construisent des S-expressions Lisp (en général des listes). Les S-expressions Lisp synthétisées par ce macro-caractère utiliseront les fonctions QUOTE, CONS, MCONS, LIST, APPEND et NCONC. Tout ce qui suit le caractère accent grave est considéré comme le modèle de la liste à construire. Les éléments variables de ce modèle seront indiqués par une virgule placée devant l'expression à calculer. Tous les autres éléments du modèle sont considérés comme des constantes. Il existe 3 types d'éléments variables :
- les éléments simples préfixés par une virgule
- les segments de liste qui seront ajoutés à la liste au moyen de la fonction APPEND, préfixés par une virgule suivie du caractère "@"
- les segments de liste qui seront ajoutés à la liste au moyen de la fonction NCONC, préfixés par une virgule suivie du caractère point "."

BACKQUOTE peut être défini en Lisp de la manière suivante :

```
(DMC |,| ()
     ; pour prévenir bien des erreurs
     (ERROR '|,| 'errsxt "en dehors d'un `"))

(DMC |`| ()
     (FLET ((|,| ()
             (COND
                ((EQ (PEEKCN) #/@)
                  (READCN)
                  (CONS '|,@| (READ)))
                ((EQ (PEEKCN) #/.)
                  (READCN)
                  (CONS '|,.| (READ)))
                (T (CONS '|,| (READ))))))
           (BACKQUOTIFY (READ))))

(DE BACKCONSTANT (X)
   ; teste si l'objet x est une constante
   (OR (NULL X)
       (AND (CONSP X)
            (EQ (CAR X) 'QUOTE)
            (NULL (CDDR X)))))

(DE BACKQUOTIFY (X)
    ; construction de l'expression "backquotée"
    (COND ((NULL X) ())
          ((ATOM X) (LIST 'QUOTE X))
```

```
                ((EQ (CAR X) '|,|) (CDR X))
                ((AND (CONSP (CAR X)) (EQ (CAAR X) '|,@|))
                   ; simplification de l'APPEND
                   (LET ((A (CDAR X))
                         (D (BACKQUOTIFY (CDR X))))
                        (COND
                           ; (append x ()) --> x
                           ((NULL D) A)
                           ; (app x (app . l)) --> (app x . l)
                           ((AND (CONSP D) (EQ (CAR D) 'APPEND))
                               (MCONS 'APPEND A (CDR D)))
                           ; rien à faire
                           (T (LIST 'APPEND A D)))))
                ((AND (CONSP X)
                      (CONSP (CAR X))
                      (EQ (CAAR X) '|,.|))
                   (IF (CDR X)
                       (LIST 'NCONC
                             (CDAR X)
                             (BACKQUOTIFY (CDR X)))
                       (CDAR X)))
                (T (LET ((A (BACKQUOTIFY (CAR X)))
                         (D (BACKQUOTIFY (CDR X))))
                        ; simplification des CONS
                        (COND
                          ((NULL D)
                              ; (cons x ()) --> (list x) ou '(x)
                              (IF (BACKCONSTANT A)
                                  (LIST 'QUOTE (LIST (CADR A)))
                                  (LIST 'LIST A)))
                          ((AND (BACKCONSTANT A)
                                (BACKCONSTANT D))
                              ; (cons 'x  'y) --> '(x . y)
                              (LIST 'QUOTE
                                    (CONS (CADR A) (CADR D))))
                          ((AND (CONSP D) (EQ (CAR D) 'CONS))
                              ; (cons x (cons y z))
                              ;  --> (mcons x y z)
                              (LIST 'MCONS A (CADR D) (CADDR D)))
                          ((AND (CONSP D) (EQ (CAR D) 'LIST))
                              ; (cons x (list y1 ... yn))
                              ;  --> (list x y1 ... yn)
                              (CONS 'LIST (CONS A (CDR D))))
                          ((AND (CONSP D) (EQ (CAR D) 'MCONS))
                              ; (cons x (mcons y1 ... yn))
                              ; --> (mcons x y1 ... yn)
                              (MCONS 'MCONS A (CDR D)))
                          (T  ; le cas général
                              (LIST 'CONS A D)))))))
```

```
ex :   `(A B C)     est lu comme  '(A B C)
       `(A ,B C)          ""       (LIST 'A B 'C)
       `(,X . Y)          ""       (CONS X 'Y)
```

```
`(A B . ,C)          ""      (MCONS 'A 'B C)
`(A B ,@C D E)       ""      (MCONS 'A 'B (APPEND C '(D E)))
`(X ,@Y ,@Z V)       ""      (CONS 'X (APPEND Y Z '(V))
`(X ,.Y ,Z)          ""      (CONS 'X (NCONC Y (LIST Z)))
```

6.4.9.3 Le macro caractère dièse (sharp)

 # dièse (sharp) [Macro caractère]

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 s'explique ainsi : à l'occurrence d'un caractère # dans le flux d'entrée, le lecteur Lisp lit le caractère suivant au moyen de la fonction READCH. Ce caractère lu, appelé sélecteur de #-macro, doit posséder une définition de fonction dans un package dont le nom se trouve dans la variable #:SYS-PACKAGE:SHARP. Cette fonction est lancée sans argument. La valeur retournée par cet appel doit être une liste (qui peut être vide) qui sera insérée au moyen de la fonction NCONC à l'objet en cours de lecture.

Certains sélecteurs de macro-caractères acceptent des paramètres numériques. Dans ce cas la fonction associée au sélecteur reçoit ce paramètre en argument. Les paramètres sont fournis entre le caractère # et le sélecteur. Ils sont toujours exprimés en base 10.
La lecture suivante invoque le sélecteur r (lecture en base quelconque) avec le paramètre 36.

```
ex :  ? #36rFOO
      = 20328
```

#:SYS-PACKAGE:SHARP [Variable]

cette variable contient le nom du package dans lequel doivent être recherchées les fonctions associées aux différents sélecteurs de #-macros. Par défaut cette valeur est SHARP

Voici décrit en Lisp le fonctionnement de ce macro-caractère :

```
(DEFVAR #:SYS-PACKAGE:SHARP 'SHARP)

(DMS |#| ()
     (LET ((c (READCN))
           argnum)
          (WHEN (MEMQ c '#"0123456789")
                (SETQ argnum (SUB c #/0))
                (WHILE (MEMQ (SETQ c (READCN)) '#"0123456789")
                       (SETQ argum (ADD (SUB c #/0)
                                        (MUL argnum 10)))))
          (LET ((f (GETFN #:SYS-PACKAGE:SHARP (ASCII c) ())))
               (IF f
                   (APPLY f (WHEN argnum (LIST argnum)))
                   (ERROR '|#| 'errudf c)))))
```

Il est possible de définir de nouvelles #-macros au moyen de la fonction suivante.

(DEFSHARP <ch> <larg> <s1> ... <sN>) [FSUBR]

définit une nouvelle #-macro. <ch> est le nouveau caractère sélecteur. La liste <larg> est la liste de paramètres de la fonction associée au sélecteur. Cette liste est vide si le sélecteur n'accepte pas de paramètre numérique. Sinon c'est une liste de paramètre usuelle, et la fonction associée au sélecteur recevra le paramètre numérique en argument. DEFSHARP possède donc la même syntaxe que les fonctions de définition (DE, DMD, etc.)

DEFSHARP peut être défini en Lisp de la manière suivante :

```
(DF DEFSHARP (ch . f)
    (SETFN (SYMBOL #:SYS-PACKAGE:SHARP ch) 'EXPR f)
    ch)
```

Il existe un certain nombre de sélecteurs prédéfinis :

#/ [#-Macro]

retourne le code interne du caractère suivant lu sur le flux d'entrée. C'est le meilleur moyen connu à ce jour pour entrer des codes internes dans un programme. #/A remplace avantageusement 65 pour décrire le code interne du caractère A.

```
ex : #/A      est lu comme      65
     #/a          " "           97
```

#\ [#-Macro]

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-liste de ce symbole sous l'indicateur #:SHARP:VALUE. Cette macro est souvent utilisée pour définir des codes internes de caractères non imprimables. Si aucune valeur se trouve sur la P-Liste, #:SHARP:VALUE provoque l'erreur variable indéfinie.

```
ex : #\CR     est lu comme      13
     #\ESC        " "           27
     #\SP         " "           32
```

pour avoir la liste des valeurs actuelles :

```
(MAPLOBLIST (LAMBDA (x) (GETPROP x '#:SHARP:VALUE)))
```

#^ [#-Macro]

retourne le code interne du caractère suivant lu sur le flux d'entrée considéré comme un caractère contrôle, c'est ASCII c'est-à-dire avec les bits 6 et 7 mis à 0.

```
ex : #^C      est lu comme     3
     #^Z          " "          21
```

#$ [#-Macro]

lit le nombre suivant en base seize.

```
ex : #$FFFF       est lu comme     -1
     #$1000           " "          4096
```

#% [#-Macro]

lit le nombre suivant en base deux.

```
ex : #%1100001    est lu comme     97
     #%110            " "          5
```

#<base>r [#-Macro]

lit le nombre suivant dans la base <base>. <Base> est un paramètre numérique accepté par la #-macro r.

```
ex : #8r177       est lu comme     127
     #3r221           " "           25
```

| [#-Macro]

permet de décrire une valeur numérique de 16 bits au moyen de deux valeurs de 8 bits, le caractère valeur absolue séparant les octets. Ce caractère ne peut apparaître qu'après utilisation d'un des sélecteurs suivants : /, \, ^, $ ou %.

```
ex : #^X|^A      est lu comme     6145
     #%10|/Z          " "          1114
     #\CR|\LF         " "          3338
```

#'

placée devant une S-expression, cette macro retourne la liste :
(FUNCTION <s-expression>).

```
ex : #'FOO        est lu comme    (FUNCTION FOO)
```

Il s'agit donc d'une abréviation pour les appels de la fonction FUNCTION.

#" [#-Macro]

retourne la liste des codes internes des caractères encadrés du caractère guillemet. Pour inclure le caractère guillemet il faut le doubler.

```
ex : #"Foo Bar"  est lu comme la liste     (70 111 111 32 66 97 114)
     #"F""B"           " "                 (70 34 66)
```

#: [#-Macro]

permet d'entrer le nom d'un symbole et de son ou ses packages. Avec ce sélecteur on peut définir des symboles de packages quelconques (symboles ou listes). Il permet aussi de décrire le type d'un vecteur ou d'une chaîne typée. Cette écriture est également utilisée par l'imprimeur qui utilise l'option #:SYSTEM:PRINT-PACKAGE-FLAG.

```
ex : #:pkgc:symb        est lue comme le symbole <symb>
                        dans le package <pkgc>
     #:(a . b):foo      le symbole foo dans le package (a . b)
     #:bar:#[e1 e2]     le vecteur [e1 e2] de type bar
```

#. [#-Macro]

Evalue l'expression suivante du flux d'entrée. Cette valeur est retournée par la #-macro. Cette #-macro permet d'appeler l'évaluateur lui-même durant une lecture.

```
ex : #.(1+ 4)     est lu comme     5
```

#([#-Macro]

lit une liste dont la première cellule est étiquetée.

```
ex : (TCONSP '#(a . b))    ->  #(a . b)
```

#[[#-Macro]

lit un vecteur de S-expressions.

```
ex : (VECTORP (MAKEVECTOR 2 4))            ->  #[4 4]
     (EQUAL (MAKEVECTOR 2 'A) #[A A]))     ->  T
```

#+ [#-Macro]

Evalue l'expression suivante du flux d'entrée. Si cette valeur est vraie, la #-macro retourne en valeur l'expression suivante du flux d'entrée sinon elle retourne () mais saute quand même l'expression suivante. Ce macro-caractère permet de réaliser une lecture conditionnelle.

```
ex : #+ GC-ON  (DE GC-ALARM () )
     ; si la variable GC-ON est differente de (), retourne
     ; la liste (DE GC-ALARM () )  sinon retourne ().
```

#- [#-Macro]

le contraire de la macro précedente. Si la valeur de l'expression suivante du flux d'entrée est fausse (égale à ()) l'expression qui suit encore est retournée en valeur sinon elle est sautée et le macro-caractère retourne ().

#| [#-Macro]

permet de débuter un commentaire multiligne. Celui-ci se terminera à l'occurrence de la séquence de caractères : |#. Il peut y avoir des commentaires multilignes à l'intérieur d'un commentaire multiligne.

Voici l'équivalent Le_Lisp des #-macro standard :

```
(DEFSHARP |.| ()
    (LIST (EVAL (READ))))

(DEFSHARP + ()
    (IF (EVAL (READ))
        (LIST (READ))
        (READ)
        ()))

(DEFSHARP - ()
    (IFN (EVAL (READ))
         (LIST (READ))
         (READ)
         ()))

(DEFSHARP |'| ()
```

```
    (LIST (LIST 'FUNCTION (READ))))
(DEFSHARP |/| ()
    (SHARP-LOWBYTE (READCN)))
(DEFSHARP |\  ()
    (LET ((l (READ)))
         (LET ((n (GETPROP l 'SHARP:VALUE)))
              (IF n
                  (SHARP-LOWBYTE n)
                  (ERROR 'SHARP:VALUE 'ERRUDV l)))))
(MAPC
   (LAMBDA (x y) (PUTPROP x y 'SHARP:VALUE))
   '(NULL BELL BS TAB LF RETURN CR   ESC SP DEL RUBOUT)
   '(0    7    8  9   10 13     13  27  32 127 127))
(DEFSHARP |^| ()
    (SHARP-LOWBYTE (LOGAND 31 (READCN))))
(DE SHARP-LOWBYTE (n)
    (IF (NEQ (PEEKCN) 124) ; le caractère valeur absolue
        (LIST n)
        (READCN)           ; saute la barre
        (LIST
          (LET ((n1 (LOGSHIFT n 8)) (n2 (READCN)))
             (SELECTQ n2
              (47 ; i.e le slash
                (LOGOR n1 (READCN)))
              (94 ; i.e. chapeau
                (LOGOR n1 (LOGAND 31 (READCN))))
              (92 ; i.e le backslash
                (LET ((l (READ)))
                     (LET ((n (GETPROP l 'SHARP:VALUE)))
                          (IF n
                              (LOGOR n1 n)
                              (ERROR 'SHARP:VALUE 'ERRUDV l)))))
              (T  ; tous les autres
                 (+ n1 (READCN)))))))))
(DEFSHARP |"| ()
   (LET ((l) (i))
        (UNTILEXIT EOC
           (IF (= (SETQ i (READCN)) #/")
               (IF (= (PEEKCN) #/")
                   (NEWL l (READCN))
                   (EXIT EOC (LIST (NREVERSE l))))
               (NEWL l i)))))))
(DEFSHARP |$| ()
    (WITH ((IBASE 16))
          (LET ((r (READ)))
               (IF (FIXP r)
                   (LIST r)
                   (ERROR '|#$| 'ERRSXT r)))))
(DEFSHARP |%| ()
```

```
    (WITH ((IBASE 2))
          (LET ((r (READ)))
               (IF (FIXP r)
                   (LIST r)
                   (ERROR '|#%| 'ERRSXT r)))))
(DEFSHARP |r| (base)
    (WITH ((IBASE base))
          (LET ((r (READ)))
               (IF (FIXP r)
                   (LIST r)
                   (ERROR '|#r| 'ERRSXT r)))))
(DEFSHARP |||| ()
    (SKIP-SHARP-COMMENT)
    ())
(DE SKIP-SHARP-COMMENT ()
    (UNTILEXIT EOC
       (SELECTQ (READCN)
            (#/# (WHEN (EQ (PEEKCN) #/|)
                       (READCN)
                       (SKIP-SHARP-COMMENT)))
            (#/| (WHEN (EQ (PEEKCN) #/#)
                       (READCN)
                       (EXIT EOC)))
            (T ()))))
(DEFSHARP |:| ()
    (WITH ((TYPECN #/: 'CPKGC)
           (TYPECN #/# 'CPNAME))
          (LIST (READ))))
(DEFSHARP |(| ()
    (REREAD '(#/())
    (LET ((1 (READ))) (LIST (TCONSMK 1))))
(DEFSHARP |[| ()
    (LIST (APPLY 'VECTOR (READ-DELIMITED-LIST #/]))))
```

6.4.9.4 Le macro caractère deux points (colon)

Bien qu'il soit toujours possible de décrire un symbole et son package en utilisant la notation absolue #:PKGC:SYMB, il est souvent utile de disposer d'une abréviation. C'est ce que fournit le macro-caractère deux-points qui placé devant un symbole le créera automatiquement dans un package dont le nom se trouve dans la variable #:SYS-PACKAGE:COLON.

#:SYS-PACKAGE:COLON [Variable]

cette variable contient le nom du package à utiliser avec le macro-caractère deux-points en tête d'un symbole. Par défaut cette valeur est USER. Cette variable décrit aussi le package à utiliser lors de la lecture d'un symbole commencant par un caractère de type CPKGC.

```
ex : (DEFVAR #:SYS-PACKAGE:COLON 'LOCAL)    -> LOCAL
     :FOO        sera lu comme              #:LOCAL:FOO
```

```
(DEFVAR #:SYS-PACKAGE:COLON
        '#:PK1:PK2)                    -> #:PK1:PK2
:FOO:BAR     sera lu comme             #:PK1:PK2:FOO:BAR

(TYPECN #// 'CPKGC)                    / est délimiteur de packages
/FOO         sera lu comme             #:PK1:PK2:FOO
/FOO/BAR          "  "                 #:PK1:PK2:FOO:BAR
```

6.4.9.5 Les macro-caractères du terminal ! ^L ^A ^E ^F ^P

Ces macro-caractères sont utilisés pour entrer des commandes rapidement en interaction sur le terminal.

^L charger (load) [Macro caractère]

placé devant un symbole, fabrique l'appel :
(LIBLOADFILE <symbole> T)
Il permet donc de charger un fichier Lisp interactivement d'une manière extrêmement concise. ^L retourne le nom du fichier effectivement chargé. Le suffixe #:SYSTEM:LELISP-EXTENSION n'est pas rajouté en fin du nom de fichier s'il est déja présent.

^L peut être défini en Lisp de la manière suivante :

```
(DMC ^L ()
     ; ^L  :  pour charger un fichier
     (LIST 'LIBLOADFILE (READSTRING) T))
```

```
ex : ^Lfoo      est lu comme      (LIBLOADFILE "FOO" T)
```

^A charger un module [Macro caractère]

A le même rôle que ^L mais charge des modules compilés avec le compilateur complice.

^A peut être défini en Lisp de la manière suivante :

```
(DMC ^A ()
   ; Pour charger un module.
   (LIST 'LOADMODULE (READSTRING)))
```

^E éditer un fichier (edit) [Macro caractère]

permet d'appeler l'éditeur PEPE sur un fichier ou sur tout ce qui est imprimé par l'évaluation d'une expression quelconque.

```
ex : ^Ebar     est lu comme    (PEPE bar)
     ^E(PRETTY foo)           (PEPE (PRETTY foo))
     ^E^Pfoo bar              (PEPE (PRETTY foo bar))
```

pour appeler PEPE sur la liste triée des noms des fonctions globales

```
^E(SORTL (MAPLOBLIST (LAMBDA (x)
                       (AND (NULL (PACKAGECELL x))
                            (TYPEFN x)))))
```

^F éditer une fonction [Macro caractère]

Permet d'appeler l'éditeur du système hôte sur le fichier dans lequel une fonction a été définie, et de recharger ce fichier en sortie de l'éditeur. Le nom de l'éditeur du système hôte est donné par la valeur de la variable globale #:SYSTEM:EDITOR. La propriété #:SYSTEM:LOADED-FROM-FILE, positionnée par les fonctions de définitions de fonctions, indique quel fichier doit être édité.

Si la fonction BAR a été chargée à partir du fichier FOO.LL

ex : ^Fbar

est lu comme :

```
(PROGN (COMLINE (CATENATE #:SYSTEM:EDITOR
                          " foo.ll"))
       (LOAD "foo.ll" t))
```

Mais si elle n'a pas été chargée à partir d'un fichier,
on crée un fichier <file> temporaire de nom (GENSYM) et

^Fbar est lu comme

est lu comme :

```
(PROGN (PRETTYF <file> bar)
       (COMLINE (CATENATE #:SYSTEM:EDITOR " <file>"))
       (LOAD <file> t))
```

^P paragrapher (pretty) [Macro caractère]

permet d'appeler le paragrapheur de fonction sur une ou plusieurs fonctions dont les noms suivent.

^P peut être défini en Lisp de la manière suivante :

```
(DMC ^P ()
     (CONS 'PRETTY
           (IMPLODE (PNAME (CATENATE "("
                                     (READSTRING)
                                     ")")))))
```

```
ex : ^Pfoo   est lu comme   (PRETTY foo)
     ^Pfoo bar              (PRETTY foo bar)
```

! clam (bang) [Macro caractère]

utilisé devant une ligne en utilisation interactive du système, fabrique l'appel : (COMLINE <reste de la ligne>) Il permet donc de fournir une ligne de commande au système d'exploitation local d'une manière extrêmement concise.

ex : !ls -ls est lu comme (COMLINE "ls -ls")

6.5 Les Fonctions de Sortie de Base

Toutes les fonctions qui suivent éditent leurs arguments dans le tampon de sortie. Les expressions ne sont effectivement imprimées que lorsque le tampon est vidé.
Ceci peut arriver à différents moments :

1- lorsque les caractères atteignent la marge droite du tampon
2- lorsque les caractères atteignent la fin du tampon
3- en utilisant les fonctions qui vident le tampon après édition (PRINT, PRINFLUSH, TERPRI).

Dans le premier cas l'interruption programmable EOL est déclenchée, une fin de ligne est donc insérée. Dans le second cas c'est l'interruption FLUSH qui est déclenchée, le tampon est donc simplement vidé. Ceci ne peut arriver que si la marge droite est plus loin que la fin du tampon de sortie (voir la fonction RMARGIN)

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

Edite dans le tampon de sortie les différentes S-expressions <s1> ... <sN> et retourne la valeur de <sN>. Cette fonction ne peut pas être décrite simplement en Lisp. Les expressions ne seront effectivement écrites que lorsque le tampon sera vidé. L'appel sans argument, (PRIN), n'a absolument aucun effet.

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

Cette fonction est identique à la fonction PRIN mais vide immédiatement le tampon de sortie après l'édition des S-expressions et commence un nouvelle ligne (en déclenchant l'interruption programmable EOL). L'appel sans argument, (PRINT), vide le tampon et commence un nouvelle ligne.

PRINT peut être défini en Lisp de la manière suivante :

```
(DMD PRINT 1
     `(PROG1 (PRIN ,@1) (ITSOFT 'EOL ())))

ex : ? (PRINT (1+ 9) (CDR '(A B C)))    ; forme à évaluer
       10(B C)                          ; exécution
     = (B C)                            ; valeur retournée
     ? (PROGN (PRIN 123) (PRINT 'FOO)) ; forme à évaluer
       123FOO                           ; exécution
     = FOO                              ; valeur retournée
     ? (PROGN (REPEAT 12 (PRIN 'A))
              (PRINT))                  ; forme à évaluer
       AAAAAAAAAAAA                     ; exécution
     = ()                               ; valeur retournée
```

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

Cette fonction est identique à la fonction PRIN mais vide immédiatement le tampon de sortie après l'édition des S-expressions (en déclenchant l'interruption programmable FLUSH). Cette fonction ne commence pas de nouvelle ligne.
L'appel sans arguments, (PRINFLUSH), vide le tampon de sortie.

PRINFLUSH peut être défini en Lisp de la manière suivante :

```
(DMD PRINFLUSH 1
     `(PROG1 (PRIN ,@1) (ITSOFT 'FLUSH ())))

ex : ? (DE QUAM (msg)
     ?     (PRINFLUSH msg) ; édite le msg dans le tampon vide le tampon
```

```
?     (READ))          ; lit la réponse.
= QUAM

? (QUAM "Nb de satellites de VEGA")
  Nb de satellites de VEGA ? xxx   ; xxx est la réponse entrée
= xxx

; pour réaliser plusieurs impressions et vider le tampon à la fin
(DE PRIN-AND-FLUSH (exp n)
    (REPEAT n (PRIN exp))
    (PRINFLUSH))
```

(TERPRI <n>) [SUBR à 0 ou 1 argument]

saute <n> lignes sur le canal de sortie courant en appelant <n> fois la fonction PRINT. L'argument <n> peut ne pas être fourni, sa valeur par défaut est 1. L'appel (TERPRI) est donc équivalent à (TERPRI 1) ou à (PRINT). TERPRI retourne toujours T en valeur.

TERPRI peut être défini en Lisp de la manière suivante :

```
(DE TERPRI n
    (REPEAT (IF n (CAR n) 1)
            (PRINT)))
```

(PRINCN <cn> <n>) [SUBR à 1 ou 2 arguments]

Edite <n> fois le caractère de code interne <cn> dans le flux de sortie. Si <n> est omis, le code <cn> n'est édité qu'une fois. PRINCN retourne le code interne <cn> en valeur.

```
ex :  (LET ((cn #/A))
           (REPEAT 10 (PRINCN (INCR cn))))
; produira
      BCDEFGHIJK

     (DE PYR (n1 n2)
         (WHEN (> n1 0)
               (PRINCN #\SP n1)
               (PRINCN #/* (1+ (* n2 2)))
               (PRINT)
               (PYR (1- n1) (1+ n2))))

     (PYR 4 0)

; produira
         *
        ***
       *****
      *******
```

(PRINCH <ch> <n>) [SUBR à 1 ou 2 arguments]

est identique à la fonction précédente, mais le caractère <ch> est spécifié sous la forme d'un symbole mono-caractère.

```
ex : ; édite  10 caractères + dans le tampon
     (PRINCH '|+| 10)
```

PRINCH peut être défini en Lisp de la manière suivante :

```
(DE PRINCH (ch)
    (PRINCN (CASCII ch)))
```

6.6 Le Contrôle des Fonctions de Sortie

6.6.1 Les limitations d'impression

Pour pouvoir limiter les impressions des structures très longues et éviter de faire boucler l'impression en cas de listes circulaires ou partagées, trois nouvelles fonctions ont été introduites.

(PRINTLENGTH <n>) [SUBR à 0 ou 1 argument]

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) le nombre maximal d'éléments de liste qui est imprimé lors d'un PRINT ou d'un PRIN. Par défaut la valeur de ce nombre est de 5000. Si une liste contient un nombre d'éléments plus élevé, le reste de ses éléments ne sera pas imprimé et trois points de suspension seront imprimés. Cette fonction permet de terminer l'impression des listes circulaires sur les CDR. Pour désactiver ce type de limitation d'impression il faut appeler cette fonction avec un argument égal à 0. PRINTLENGTH retourne en valeur la longueur maximale courante d'impression.

```
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 maximale d'impression lors d'un PRINT ou d'un PRIN. Par défaut la valeur de ce nombre est de 100. Cette valeur représente le nombre maximal de parenthèses ouvrantes non encore refermées. 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. Pour désactiver ce type de limitation d'impression il faut appeler cette fonction avec un argument égal à 0. PRINTLEVEL retourne en valeur la profondeur maximale courante d'impression.

```
ex : (PRINTLEVEL 3) -> 3
     '(DE FOO (l)
          (IF (NULL (CDR l))
              l
              (FOO (CDR l))))
     -> (DE FOO (l) (IF (NULL &) l (FOO &)))

; impression des listes circulaires ;

? (PRINTLEVEL 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 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 ...
?
```

(PRINTLINE <n>) [SUBR à 0 ou 1 argument]

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) le nombre maximal de lignes qui vont être imprimées lors d'un PRINT ou un PRIN. Par défaut la valeur de ce nombre est 2000. Si une expression nécessite plus de lignes, la dernière ligne apparaîtra avec des points de suspension. Pour désactiver ce type de limitation d'impression il faut appeler cette fonction avec un argument égal à 0. PRINTLINE retourne en valeur le nombre maximal de lignes autorisé pour une édition.

Il existe une bibliothèque permettant d'imprimer des programmes Le_Lisp d'une manière beaucoup plus lisible (voir le chapitre 8) ainsi qu'une bibliothèque d'impression des listes circulaires ou partagées (voir le chapitre 9).

6.7 L'Edition Standard

Les différents objets Lisp sont édités dans le tampon de sortie en respectant l'unique règle suivante : la représentation externe d'un objet atomique ne peut être éditée à cheval sur deux lignes; ceci pour les symboles, les chaînes de caractères et les nombres.

L'impression de la fonction (QUOTE <s>) s'effectue '<s> (donc en restituant le macro-caractère prédéfini d'entrée) pour des raisons de lisibilité.

Le nombre *innommable* en complément à 2, "-0", s'écrit toujours :

```
? #$8000
= #$8000
? (LOGSHIFT 1 15)
= #$8000
```

Enfin tout pointeur qui n'est pas un pointeur Le_Lisp s'écrit toujours :

```
   #<>
ex : ? (VAG '(-1 . -1))
   = #<>
```

Trois variables système et deux fonctions permettent de régler certaines modalités d'impression.

#:SYSTEM:PRINT-FOR-READ [Variable]

contient l'indicateur d'impression des symboles et des chaînes de caractères. Si #:SYSTEM:PRINT-FOR-READ vaut T, les symboles contenant des caractères autres que des caractères de type CPNAME sont encadrés du caractère | à l'impression, et les chaînes de caractères sont encadrées du caractère ". Les | et " sont de plus doublés à l'intérieur de ces objets. Enfin le système imprime un espace entre chacun des arguments des fonction PRINT, PRIN et PRINFLUSH.
Ainsi, quand cet indicateur est positionné, l'impression produite peut être relue par la fonction READ.

```
ex : ? (SETQ a '(foobar |bar foo| "avec ""guillemet"""))
     = (foobar bar foo avec "guillemet")
     ? (SETQ #:system:print-for-read t)
     = t
     ? a
     = (foobar |bar foo| "avec ""guillemet""")
```

#:SYSTEM:PRINT-CASE-FLAG [Variable]

contient l'indicateur de casse en sortie. Si #:SYSTEM:PRINT-CASE-FLAG vaut T, les symboles sont imprimés en majuscules. Si #:SYSTEM:PRINT-CASE-FLAG vaut (), et c'est l'option initiale, les symboles sont imprimés comme ils ont été entrés. Notez qu'en général l'indicateur de lecture #:SYSTEM:READ-CASE-FLAG vaut () et que tous les symboles sont alors lus en minuscules.

```
ex : ? ; Différencions majuscules et minuscules à la lecture
     ? (SETQ #:SYSTEM:READ-CASE-FLAG t)
     = t
     ? (SETQ a '(foobar FOOBAR FooBar))
     = (foobar FOOBAR FooBar)
     ? (SETQ #:SYSTEM:PRINT-CASE-FLAG t)
     = T
     ? a
     = (FOOBAR FOOBAR FOOBAR)
     ? (SETQ #:SYSTEM:PRINT-CASE-FLAG ())
     = ()
     ? a
     = (foobar FOOBAR Foobar)
```

#:SYSTEM:PRINT-PACKAGE-FLAG [Variable]

contient l'indicateur d'impression des packages. Cet indicateur peut prendre trois valeurs :
- () : les packages ne sont pas imprimés
- T (défaut) : les packages sont imprimés en notation absolue #:pkg:symb.
- 0 : si la valeur du package est égale à la valeur de la variable #:SYS-PACKAGE:COLON, les packages sont imprimés en mode raccourci en utilisant la syntaxe du macro-caractère ":".

```
ex : ? (SETQ #:SYSTEM:PRINT-PACKAGE-FLAG T)
     = T
     ? (SETQ #:SYS-PACKAGE:COLON 'USER)
     = USER
```

```
? '#:FOO:BAR
= #:FOO:BAR
? ':FOO
= #:USER:FOO
? (SETQ #:SYSTEM:PRINT-PACKAGE:FLAG ())
= ()
? '#:FOO:BAR
= BAR
? ':FOO
= FOO
? (SETQ #:SYSTEM:PRINT-PACKAGE:FLAG 0)
= 0
? '#:FOO:BAR
= #:FOO:BAR
? ':FOO
= :FOO
```

(OBASE <n>) [SUBR à 0 ou 1 argument]

permet de consulter si l'argument <n> n'est pas fourni, ou de modifier la valeur de la base de conversion des nombres en sortie. Cette base doit être comprise entre 2 et 36. Cette fonction est la contrepartie de la fonction IBASE.

```
ex : ? 100
     = 100
     ? (OBASE 16)
     ; OBASE retourne toujours 10, voyez-vous pourquoi ?
     = 10
     ? 100
     = 64
     ? (OBASE 17)    ; et pourquoi pas ...
     = 10
     ? (IBASE 13)    ; conversion base 13 en base 17!
     = D
     ? 23AB
     = 107A
     ? 34AB
     = 18AD
```

(PTYPE <symb> <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 <symb>. Ce P-TYPE est principalement utilisé par le paragrapheur d'expressions Lisp (voir le chapitre 8) pour y stocker le format à utiliser pour éditer ce symbole en tant que fonction. PTYPE retourne la valeur courante du P-TYPE du symbole <symb>.

6.8 Les Entrées/Sorties sur des Listes

(EXPLODE <s>) [SUBR à 1 argument]

retourne la liste de tous les codes internes de la représentation externe de l'expression <s>, qui peut être de type quelconque. EXPLODE retourne la liste des codes qui seraient imprimés 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)
```

(EXPLODECH <s>) [SUBR à 1 argument]

est équivalente à la fonction précédente mais retourne une liste d'atomes (symboles mono-caractères pour les lettres et les caractères spéciaux, nombres pour les chiffres).

EXPLODECH peut être défini en Lisp de la manière suivante :

```
(DE EXPLODECH (s)
    (MAPCAR 'ASCII (EXPLODE s)))
```

```
ex : (EXPLODECH -120)        -> (- 1 2 0)
     (EXPLODECH '(CAR '(A B)))
         -> (|(| c a r | | |'| |(| a | | b |)| | |)|))
```

(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 des codes internes <ln>. IMPLODE est l'inverse de EXPLODE et permet de fabriquer de nouveaux objets Lisp au moyen de leur représentation externe d'une manière analogue à 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))      -> -237
     (IMPLODE (EXPLODE '(A B)))    -> (A B)
```

(IMPLODECH <s>) [SUBR à 1 argument]

est équivalente à la fonction précédente mais utilise une liste d'atomes (symboles mono-caractère pour les lettres et les caractères spéciaux, nombres pour les chiffres).

IMPLODECH peut être défini en Lisp de la manière suivante :

```
(DE IMPLODECH (s)
    (IMPLODE (MAPCAR 'CASCII s)))
```

```
ex : (IMPLODECH '(- 1 2 3))              -> -123
     (IMPLODECH (EXPLODECH '(a b)))      -> (a b)
```

6.9 Gestion des Tampons d'Entrée/Sortie

Toutes les fonctions de lecture et d'impression décrites jusqu'à présent travaillent au travers de tampons. Les fonctions de lecture lisent des caractères dans le tampon d'entrée, les fonction d'impression écrivent des caractères dans le tampon de sortie.

Ces tampons sont remplis et vidés par le système au moyen des trois interruptions programmables BOL (Beginning-Of-Line), EOL (End-Of-Line) et FLUSH.

Il y a effectivement un tampon différent pour chaque canal d'entrée et de sortie ce qui permet de réaliser des entrées/sorties sur plusieurs canaux parallèlement. Les chaînes de caractères des tampons sont allouées automatiquement par le système lors de la création de nouveaux canaux. Leur longueur, qui est de 256 caractères, ne peut être modifiée.

6.9.1 Le tampon d'entrée

6.9.1.1 Introduction

Une chaîne de caractères, INBUF, représente le contenu du tampon d'entrée courant. Deux index dans la chaîne, INMAX et INPOS, représentent la fin du tampon et la position de lecture courante. INMAX est l'index du prochain caractère à lire dans la chaîne.

Si pendant une lecture INMAX rejoint INPOS, ceci signifie que tous les caractères disponibles dans le tampon ont été lus. L'interruption programmable BOL est alors déclenchée. La fonction invoquée par cette interruption doit remplir le tampon d'entrée et positionner INMAX pour indiquer combien de caractères ont été chargés dans le tampon.

6.9.1.2 Manipulation du tampon d'entrée

Toutes ces fonctions manipulent le tampon du canal d'entrée courant, sélectionné par la fonction INCHAN.

(INBUF <n> <cn>) [SUBR à 0, 1 ou 2 arguments]

rend en valeur la chaîne de caractères du tampon d'entrée courant. Les fonctions de manipulation de chaînes permettent de manipuler facilement le contenu de ce tampon. Voici, par exemple, une manière particulièrement efficace d'implanter la fonction READSTRING :

```
(DE READSTRING ()
    (WHEN (= (INPOS) (INMAX))
          (ITSOFT 'BOL ()))
    (PROG1 (SUBSTRING (INBUF) 0 (SUB (INMAX) 2))
           (INPOS (INMAX))))
```

INBUF accepte deux arguments optionnels qui permettent de consulter et de modifier directement le contenu du tampon. On a les équivalences :

```
(INBUF n)          ==      (SREF (INBUF) n)
(INBUF n cn)       ==      (SSET (INBUF) n cn)
```

(INMAX <n>) [SUBR à 0 ou 1 argument]

permet de spécifier, si <n> est fourni, le nombre maximal de caractères disponibles dans le tampon d'entrée. Si <n> n'est pas fourni, INMAX retourne le nombre actuel de caractères du tampon d'entrée. Cette fonction est principalement utilisée à l'intérieur d'une fonction BOL, pour spécifier le nombre de caractères chargés dans le tampon.

(INPOS <n>) [SUBR à 0 ou 1 argument]

permet de spécifier, si <n> est fourni, la position actuelle de lecture sur le tampon d'entrée. Si <n> n'est pas fourni, INPOS retourne la position actuelle de lecture sur le tampon d'entrée. Cette fonction est très rarement utilisée.

6.9.1.3 L'interruption programmable début de ligne

Quand il n'y a plus de caractères à lire dans le tampon d'entrée (c'est-à-dire lorsqu'on a lu les INMAX caractères du tampon), le système déclenche une interruption programmable de nom BOL. La fonction associée à cette interruption remplit le tampon d'entrée avec les caractères suivants lus sur le canal INCHAN, y ajoute éventuellement les marqueurs de fin de ligne #\CR et #\LF, et positionne l'index INMAX. Il est, bien sûr, possible de prendre le contrôle de cette interruption en positionnant la variable #:SYS-PACKAGE:ITSOFT. Il faut alors remplir soi-même le tampon et positionner l'index INMAX. L'index INPOS est toujours remis à zéro par le système après le traitement de l'interruption BOL; il est donc inutile de le positionner soi même.

BOL [Interruption programmable]

cette interruption programmable est déclenchée par les fonctions READ, READCN, PEEKCN, READSTRING et les fonctions associées lorsque le tampon d'entrée est vide (INPOS = INMAX).

(BOL) [SUBR à 0 argument]

met dans le tampon d'entrée la ligne suivante lue sur le terminal ou sur un fichier et positionne INMAX. C'est cette fonction qui permet d'éditer la ligne tapée par l'utilisateur (effacement par BS ou DEL, annulation par ^X ou ^U) et qui fait l'écho des caractères de contrôle *à la DEC.*

La fonction BOL utilise les fonctions du terminal virtuel (TYI, TYCN, etc.) pour lire les caractères et en faire éventuellement l'écho. Il suffit donc, en général, de définir un nouveau type de terminal virtuel pour rediriger les lectures vers un nouveau flux (fenêtres, voies séries, etc.). Il peut être intéressant de consulter le terminal virtuel de type #:TTY:WINDOW qui décrit une telle redirection (fichier VIRBITMAP de la bibliothèque).

Nous donnons ci-dessous une description partielle de la fonction BOL en Lisp. Nous n'y décrivons ni la lecture sur les fichiers, ni l'éditeur de ligne, ni l'écho des caractères de contrôle. L'utilisateur intéressé consultera le source de l'éditeur de ligne EDLIN de la bibliothèque standard.

```
(DE BOL ()
    (IF (FIXP (INCHAN))
        (...lecture sur fichier...)   ; ne peux pas être décrit en Lisp
```

```
(LET ((max 0))
     (TYSTRING (PROMPT) (SLEN (PROMPT)))
     (COND
          (#:SYSTEM:LINE-MODE-FLAG
           (SETQ max (TYINSTRING (INBUF))))
          ((NOT #:SYSTEM:REAL-TERMINAL-FLAG)
           (UNTIL (MEMQ (SSET (INBUF) max (TYI)) '(#\CR #\LF))
                  (INCR max)))
          (T
             (LET ((inbuf (INBUF))
                   (char 0))
                  (UNTIL (MEMQ (SETQ char (TYI)) '(#\CR #\LF))
                         (TYCN char)
                         (SSET inbuf max char)
                         (INCR max))
                  (TYNEWLINE))))
     (SSET (INBUF) max #\CR)
     (INCR max)
     (SSET (INBUF) max #\LF)
     (INCR max)
     (INMAX max))))
```

Voici ci-dessous un exemple de fonction BOL qui permet de compter le nombre de lignes frappées par l'utilisateur au terminal. Ce nombre de lignes est constamment indiqué dans le signe d'invite.

```
? (DEFVAR #:NUMBERING:LINE 0)
= #:NUMBERING:LINE
?
? (DE #:NUMBERING:BOL ()
?     (WHEN (NULL (INCHAN))
?           (PROMPT (CATENATE (INCR #:NUMBERING:LINE) "? ")))
?     (SUPER-ITSOFT 'NUMBER 'BOL ()))
= #:NUMBERING:BOL
?
? (SETQ #:SYS-PACKAGE:ITSOFT (CONS 'NUMBERING #:SYS-PACKAGE:ITSOFT))
= (NUMBERING)
1?
2? 123
= 123
3? ()
= ()
4? (DE FOO (x)
5?    (+ x x))
= FOO
6? (FOO 12)
= 24
```

6.9.2 Le tampon de sortie

6.9.2.1 Introduction

Une chaîne de caractères, OUTBUF, représente le contenu du tampon de sortie courant. Un index dans la chaîne, OUTPOS, permet de repérer la position courante d'écriture, c'est-à-dire l'index dans le tampon où doit être placé le prochain caractère. Deux index, LMARGIN et RMARGIN, permettent de repérer les marges, gauche et droite du tampon de sortie.

Si pendant une écriture OUTPOS rejoint RMARGIN, ceci signifie que l'on essaye d'écrire un caractère au-delà de la marge droite. L'interruption programmable EOL est alors déclenchée. La fonction invoquée par cette interruption doit vider le tampon de sortie sur le canal courant (terminal ou fichier), y ajouter une marque de fin de ligne, remplir le tampon avec le caractère *espace*, et positionner l'index OUTPOS à la marge gauche (LMARGIN).

Si pendant une écriture OUTPOS rejoint la fin du tampon, ceci signifie que l'on essaye d'écrire une ligne plus longue que la longueur du tampon. Ceci est possible lorsque la marge droite (RMARGIN) est plus loin que la fin du tampon. En ce cas l'interruption programmable FLUSH est déclenchée. La fonction invoquée par cette interruption doit vider le tampon de sortie sur le canal courant, remplir le tampon de caractères *espace*, et positionner l'index OUTPOS à 0.

6.9.2.2 La manipulation du tampon de sortie

(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 paragrapheur d'expression Lisp 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 78). 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 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 0, 1 ou 2 arguments]**

OUTBUF retourne la chaîne de caractères représentant le tampon de sortie Cette chaîne peut être manipulée par les fonctions usuelles de manipulation de chaînes de caractères.

OUTBUF accepte deux arguments optionnels qui permettent de consulter et de

modifier directement les caractères du tampon. On a les équivalences :

```
(OUTBUF n)        ==    (SREF (OUTBUF) n)
(OUTBUF n cn)     ==    (SSET (OUTBUF) n cn)
```

6.9.2.3 Les interruptions programmables EOL et FLUSH

EOL [Interruption programmable]

Cette interruption programmable est déclenchée par les fonction PRIN, PRINT, PRINFLUSH, PRINCN ou PRINCH si la position d'écriture courante, OUTPOS, dépasse la marge droite du tampon, RMARGIN. Les fonctions PRINT et TERPRI invoquent de plus explicitement cette interruption programmable.

La fonction invoquée par cette interruption doit :
- vider le tampon de sortie, de l'index 0 à l'index OUTPOS sur le canal de sortie courant.
- insérer une fin de ligne sur le canal de sortie.
- remplir le tampon de sortie avec le caractère *espace*.
- positionner l'index OUTPOS à la valeur de la marge gauche, LMARGIN.

(EOL) [SUBR à 0 argument]

cette fonction fait le traitement standard de l'interruption programmable EOL. Nous en donnons ci-dessous une description partielle en Lisp. Nous n'y décrivons pas l'impression sur les fichiers qui ne peut être décrite en Lisp.

On remarquera que cette fonction utilise les fonctions du terminal virtuel TYSTRING et TYNEWLINE, pour réaliser les impressions au terminal. Ceci signifie qu'usuellement il suffira de définir un nouveau type de terminal virtuel pour rediriger les impressions vers de nouveaux flux (synthétiseurs de musique, fenêtres, etc.).

```
(DE EOL ()
    (IF (FIXP (OUTCHAN))
        (...impression sur fichier...) ; ne peut être décrit en Lisp
        (TYSTRING (OUTBUF) 0 (OUTPOS)) ; vide le tampon au terminal
        (TYNEWLINE)                    ; marque la fin de ligne
        (FILLSTRING (OUTBUF) 0 #\SP (RMARGIN)) ; nettoie le tampon
        (OUTPOS (LMARGIN))))           ; positionne à la marge gauche
```

FLUSH [Interruption Programmable]

Cette interruption programmable est déclenchée par les fonctions d'impression lorsque la position courante, OUTPOS, atteint la fin du tampon de sortie. Ceci est possible si la marge droite, RMARGIN, est plus loin que la longueur du tampon d'impression. L'interruption est aussi déclenchée explicitement par la fonction PRINFLUSH.

La fonction invoquée par cette interruption doit :
- vider le tampon de sortie, de l'index 0 à l'index OUTPOS sur la canal de sortie courant.
- remplir le tampon de sortie avec le caractère *espace*.
- positionner l'index OUTPOS au début du tampon de sortie.

(FLUSH) [SUBR à 0 argument]

cette fonction fait le traitement standard de l'interruption programmable FLUSH. Nous en donnons ci-dessous une description partielle en Lisp. Nous n'y décrivons pas l'impression sur les fichiers qui ne peut être décrite en Lisp.

```
(DE FLUSH ()
    (IF (FIXP (OUTCHAN))
        (...impression sur fichier...) ; ne peut être décrit en Lisp
        (TYSTRING (OUTBUF) 0 (OUTPOS)) ; vide le tampon au terminal
        (FILLSTRING (OUTBUF) 0 #\SP (RMARGIN)) ; nettoie le tampon
        (OUTPOS 0)))                    ; positionne au début du tampon
```

Les interruptions programmables EOL et, plus rarement FLUSH, peuvent être redéfinies pour étendre le système d'impression.

```
ex : la fonction suivante retourne la liste des lignes
     imprimées par l'évaluation de son corps sous la
     forme d'une liste de chaînes de caractères :

(DF STREAM-OUTPUT #:SYSTEM:F
    (LET ((#:SYS-PACKAGE:ITSOFT 'STREAM-OUTPUT)
          (#:SYSTEM:L ()))          ; la liste résultat
         (EPROGN #:SYSTEM:F)
         (NREVERSE #:SYSTEM:L)))

(DE #:STREAM-OUTPUT:EOL ()
    (NEWL #:SYSTEM:L (SUBSTRING (OUTBUF) 0 (OUTPOS)))
    (FILLSTRING (OUTBUF) 0 #\SP (RMARGIN))
    (OUTPOS (LMARGIN)))
```

Les fonctions suivantes implantent par exemple une limitiation de l'impression similaire à l'utilitaire MORE d'UNIX.

```
(SETQ #:SYS-PACKAGE:COLON 'MORE)

(DEFVAR :COUNT (TYYMAX))
(DEFVAR :STRING "--More--")

(DE  :BOL ()
     (SETQ :COUNT (TYYMAX))
     (SUPER-ITSOFT 'MORE 'BOL ()))

(DE  :EOL ()
     (WHEN (= :COUNT 0)
           (TYSTRING :STRING (SLENGTH :STRING))
           (SELECTQ (TYI)
              ((#^M #^J) (SETQ :COUNT 1))
              (#^D (SETQ :COUNT (DIV (TYYMAX) 2)))
              (#/  (SETQ :COUNT (TYYMAX)))
              (#/Q (SETQ :COUNT (TYYMAX))
               (FOR (I (SUB1 (SLENGTH :STRING)) -1 0)
                    (TYBACK (CHRNTH I :STRING)))
               (FILLSTRING (OUTBUF) 0 #\SP (OUTPOS))
               (OUTPOS (LMARGIN))
               (EXIT #:SYSTEM:TOPLEVEL-TAG))
              (T (SETQ :COUNT 1)))
          (FOR (I (SUB1 (SLENGTH :STRING)) -1 0)
```

```
                (TYBACK (CHRNTH I :STRING))))
        (DECR :COUNT)
        (SUPER-ITSOFT 'MORE 'EOL ()))
    (SETQ #:SYS-PACKAGE:ITSOFT (CONS 'MORE #:SYS-PACKAGE:ITSOFT))
```

6.10 Les Fonctions sur les Flux d'Entrée/Sortie

Le_Lisp gère des *flux d'entrée/sortie* composés de :
- un terminal d'entrée/sortie (3 flux distincts)
- un certain nombre (en général 12) de fichiers en entrée ou en sortie.

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. Par convention le numéro du flux d'entrée correspondant au terminal est (). Les deux flux de sortie correspondant au terminal (flux normal et flux du terminal virtuel) ont par convention les numéros () et T.

Une *spécification de fichier* <file> est en Lisp une chaîne de caractères (ou un argument qui peut se convertir en chaîne au moyen de la fonction STRING) qui utilise la même syntaxe que le système d'exploitation hôte.

```
ex : "LLIB.VIRTTY.LL"
     "/usr/local/lelisp/foo.ll"
     "lelisp$disk:[LELISP.LLIB]VDT.LL"
     ">udd>Vlsi>lelisp>startup.ll"
     "\lelisp\llib\pepe.ll"
```

Toutes les fonctions de cette section vont utiliser le système de gestion de fichiers du système hôte qui peut retourner un code erreur en cas d'anomalie. En ce cas, l'erreur Lisp ERRIOS se déclenche dont le libellé est :

```
** <fn> : erreur d'entrée sortie : <n>     ou bien
** <fn> : I/O error : <n>
```

dans lequel <fn> 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 détectées par Le_Lisp sont :

-1	la fonction n'est pas implantée
-2	plus de canaux disponibles
-3	numéro de canal incorrect
-4	canal non ouvert

Les codes d'erreur positifs sont dépendants du système d'exploitation utilisé. En général le numéro d'erreur est 1.

Dans certains cas il est possible de faire imprimer par le système lui-même le libellé en clair de l'erreur. Une variable permet de sélectionner cette possibilité.

#:SYSTEM:PRINT-MSGS [Variable]

si cette variable possède une valeur différente de 0, le système d'exploitation hôte va imprimer (s'il le peut) les libellés en clair des erreurs qu'il a rencontrées dans la gestion des entrées/sorties.

6.10.1 Catalogues et extensions par défaut

Pour écrire des fonctions indépendantes de l'installation du système de fichiers, les variables suivantes sont toujours initialisées :

#:SYSTEM:LLIB-DIRECTORY [Variable]

contient le nom du catalogue contenant la bibliothèque standard Le_Lisp.

#:SYSTEM:LLUB-DIRECTORY [Variable]

contient le nom du catalogue contenant la bibliothèque utilisateur Le_Lisp.

#:SYSTEM:LELISP-EXTENSION [Variable]

contient l'extension par défaut des fichiers source Le_Lisp.

6.10.2 La sélection des flux d'entrée/sortie

(OPENI <file>) [SUBR à 1 argument]

permet d'ouvrir le fichier de nom <file> *en lecture* et retourne le numéro de canal <chan> associé à ce fichier.

```
ex : (OPENI "foo.ll")  -> 11
```

(OPENIB <file>) [SUBR à 1 argument]

permet d'ouvrir le fichier de nom <file> *en lecture binaire* et retourne le numéro de canal <chan> associé à ce fichier. Ce fichier ne peut être lu qu'avec les fonctions READCN et READCH.

```
ex : (OPENIB "foo.dir")  -> 11
```

(OPENO <file>) [SUBR à 1 argument]

permet d'ouvrir le fichier de nom <file> *en écriture* et retourne le numéro de canal <chan> associé à ce fichier. Si ce fichier existait déjà, il est détruit.

```
ex : (OPENO "src.list")  -> 10
```

(OPENOB <file>) [SUBR à 1 argument]

permet d'ouvrir le fichier de nom <file> *en écriture binaire* et retourne le numéro de canal <chan> associé à ce fichier. Si ce fichier existait déjà, il est détruit. L'écriture sur ce fichier ne peut se faire qu'au moyen des fonctions PRINCN et PRINCH.

```
ex : (OPENOB "foo.o")  -> 10
```

(OPENA <file>) [SUBR à 1 argument]

permet d'ouvrir le fichier de nom <file> *en ajout* et retourne le numéro de canal <chan> associé à ce fichier. Si le fichier n'existait pas, il est créé et s'il existait il est positionné en fin de fichier de telle sorte que toutes les écritures auront lieu à la suite de ce qui existait déjà.

```
ex : (OPENA "subsys.log")  -> 9
```

(OPENAB <file>) [SUBR à 1 argument]

permet d'ouvrir le fichier de nom <file> *en ajout binaire* et retourne le numéro de canal <chan> associé à ce fichier. Si le fichier n'existait pas, il est créé et s'il existait il est positionné en fin de fichier de telle sorte que toutes les écritures auront lieu à la suite de ce qui existait déjà. L'écriture sur ce fichier ne peut se faire qu'au moyen des fonctions PRINCN et PRINCH.

```
ex : (OPENAB "subsys.lib")  -> 9
```

(INCHAN <chan>) [SUBR à 0 ou 1 argument]

associe au flux d'entrée courant le canal <chan>, c'est-à-dire que toutes les lectures suivantes auront lieu sur le canal <chan>. Si l'argument n'est pas fourni, INCHAN retourne le canal d'entrée courant. Il se produit une erreur (-4) lorsque le canal associé au flux d'entrée courant a été fermé (au moyen de la fonction EOF ou CLOSE), qu'une lecture a été demandée et qu'aucun autre canal n'a été sélectionné. Cette erreur provoque également la sélection du canal terminal () en entrée.

(OUTCHAN <chan>) [SUBR à 0 ou 1 argument]

associe au flux de sortie courant le canal <chan>, c'est-à-dire que toutes les écritures suivantes auront lieu sur le canal <chan>. Si l'argument n'est pas fourni, OUTCHAN retourne le canal de sortie courant. Il se produit une erreur (-4) lorsque le canal associé au flux de sortie courant a été fermé (au moyen de la fonction CLOSE), qu'une écriture a été demandée et qu'aucun autre canal n'a été sélectionné. Cette erreur provoque également la sélection du canal terminal () en sortie.

(CHANNEL <chan>) [SUBR à 0 ou 1 argument]

retourne un descriptif du canal de numéro <chan>. Cette fonction permet de contrôler l'état des canaux. Si <chan> est omis, la liste de tous les descriptifs des canaux est retournée. Un descriptif de canal est une liste de la forme :

(<état> <nom>)

<état> est le type codé de l'état du canal :

0	canal non ouvert
1	canal ouvert en lecture
2	canal ouvert en écriture
3	canal ouvert en lecture binaire
4	canal ouvert en écriture binaire

<nom> est le nom du fichier associé à ce canal (s'il existe).
Pour connaître le nombre de canaux disponibles, évaluez :

```
(LENGTH (CHANNEL))
```

(CLOSE <chan>) [SUBR à 0 ou 1 argument]

ferme le canal de numéro <chan>. CLOSE retourne toujours T. Si aucun argument n'est fourni, CLOSE ferme tous les canaux ouverts.

(INPUT <file>) [SUBR à 1 argument]

cette fonction permet d'ouvrir le fichier <file> en entrée et de sélectionner le canal ainsi défini. ATTENTION : cette fonction fait perdre le numéro du canal d'entrée précédemment ouvert et ne peut être utilisée qu'en boucle d'interaction sur le terminal. La fin du fichier provoquera néammoins l'erreur "échappement indéfini EOF" (voir la section suivante).

INPUT peut être défini en Lisp de la manière suivante :

```
(DE INPUT (f)
    (INCHAN (OPENI f)))
```

une manière peu orthodoxe de charger un fichier à partir du terminal, en imprimant les valeurs des évaluations est donc :

```
(INPUT <f>)
```

(OUTPUT <file>) [SUBR à 1 argument]

cette fonction permet d'ouvrir le fichier de nom <file> en sortie et de sélectionner le canal ainsi défini. ATTENTION : cette fonction fait perdre le numéro du canal de sortie précédemment ouvert et ne doit être utilisé qu'en boucle d'interaction sur le terminal pour stocker sur fichier les valeurs des évaluations. Pour imprimer de nouveau les valeurs sur le terminal, évaluez simplement :

```
(OUTPUT ())
```

OUTPUT peut être défini en Lisp de la manière suivante :

```
(DE OUTPUT (f)
    (OUTCHAN (OPENO f)))
```

ex : de manipulation de fichiers

; fonction COPYFILE qui copie entièrement le fichier
; d'entrée <filin> dans le fichier de sortie <filout>
; en y compactant les expressions Lisp.

```
(DE COPYFILE (filin filout)
    (WITH ((INCHAN (OPENI filin))
           (OUTCHAN (OPENO filout))
           (OBASE 10)
           (LMARGIN 0)
           (RMARGIN 70))
          (LET ((#:SYSTEM:PRINT-FOR-READ t)
                (#:SYSTEM:PRINT-PACKAGE t))
               (UNTILEXIT EOF (PRINT (READ))))
          (CLOSE (OUTCHAN))
          filout)))
```

```
? (COPYFILE 'foo.ll 'bar.ll)
= bar.ll
```

6.10.3 L'interruption programmable de fin de fichier

EOF [Interruption programmable]

Une fin de fichier provoque l'interruption programmable EOF qui reçoit en paramètre le numéro de canal ayant atteint une fin de fichier.

La fonction invoquée par cette interruption programmable doit en général fermer le canal au moyen de la fonction CLOSE et provoquer également un échappement de nom EOF.

(EOF <chan>) [SUBR à 1 argument]

cette fonction ferme le canal de numéro <chan> et effectue un échappement de nom EOF. Si cette fonction est appelée durant la lecture d'une S-expression (par exemple au milieu de la lecture d'une liste) elle provoque l'erreur de syntaxe *fin de fichier pendant une lecture.*

EOF peut être défini en Lisp de la manière suivante :

```
(DE EOF (n)
    (CLOSE n)
    (INCHAN ())
    (IF #:SYSTEM:IN-READ-FLAG
        (ERROR 'READ 'ERRSXT "EOF durant un READ")
        (EXIT EOF n)))
```

L'interruption programmable EOF peut être redéfinie par l'utilisateur.

Les fonctions suivantes permettent par exemple de charger en séquence des fichiers de nom <file>.1 ... <file>.n; Les S-expressions pouvant être à cheval sur deux fichiers.

```
(DEFVAR #:SYS-PACKAGE:COLON 'LOAD-IN-SEQUENCE)

(DEFVAR :NUMBER)   ; l'extension courante
(DEFVAR :FILE)     ; le nom du fichier

(DE LOAD-IN-SEQUENCE (FILE)
    (LET ((:NUMBER 1)
          (:FILE FILE)
          (#:SYS-PACKAGE:ITSOFT
             (CONS 'LOAD-IN-SEQUENCE #:SYS-PACKAGE:ITSOFT)))
         (LOADFILE (CATENATE FILE "." :NUMBER) T)))

(DE :EOF (N)
    (IF (NOT (PROBEFILE (CATENATE :FILE "." (1+ :NUMBER))))
        ; il n'y a plus de fichier :
        ; traitement standard de la fin de fichier
        (SUPER-ITSOFT 'LOAD-IN-SEQUENCE 'EOF (LIST N))
        ; passage au fichier suivant dans la séquence
        (CLOSE N)
        (INCHAN (OPENI (CATENATE :FILE "." (INCR :NUMBER))))))
```

6.10.4 Les fonctions sur les fichiers

(PROBEFILE <file>) [SUBR à 1 argument]

retourne T si le fichier <file> existe et () dans le cas contraire.

PROBEFILE peut être défini en Lisp de la manière suivante :

```
(DE PROBEFILE (f)
    (LET ((i (CATCHERROR () (OPENI f))))
         (WHEN (CONSP i)
               (CLOSE (CAR i))
               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.

6.10.5 La fonction LOAD et le mode AUTOLOAD

(LOADFILE <file> <i>) [SUBR à 2 arguments]

permet de charger (en évaluant toutes les expressions qui s'y trouvent) le fichier de nom <file>. L'indicateur <i> est la valeur de la variable #:SYSTEM:REDEF-FLAG (voir les fonctions de définition) durant ce chargement. LOADFILE retourne toujours <file> en valeur.

LOADFILE peut être défini en Lisp de la manière suivante :

```
(DE LOADFILE (file redef?)
    (IFN (PROBEFILE file)
         (ERROR 'LOADFILE "fichier inconnu" file)
         (LET ((#:SYSTEM:LOADED-FROM-FILE file)
               (#:SYSTEM:REDEF-FLAG redef?)
               (#:SYS-PACKAGE:COLON #:SYS-PACKAGE:COLON)
               (#:SYSTEM:IN-READ-FLAG ())
               (inchan (INCHAN)) )
              (INCHAN (OPENI file))
              (PROTECT (UNTILEXIT EOF (EVAL (READ)))
                       (LET ((in (INCHAN)))
                            (WHEN in (CLOSE in)))
                       (INCHAN inchan) ))
         file ))
```

(LOAD <file> <i>) [FSUBR à 1 ou 2 arguments]

cette fonction est la forme FSUBRée de la fonction précédente. De plus l'argument <i> est optionnel et vaut () par défaut.

LOAD peut être défini en Lisp de la manière suivante :

```
(DF LOAD (file . redef?)
    (LOADFILE file (CAR redef?)) )
```

(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 *autoload* (qui se chargent toutes seules dynamiquement à leur premier appel).
<file> est le nom d'un fichier qui doit être automatiquement chargé, au moyen de la fonction LOADFILE, si un des symboles <sym1> ... <symN> doit être évalué en tant que fonction. Le chargement du fichier suit la convention des appels par nécessité.

AUTOLOAD peut être défini en Lisp de la manière suivante :

```
(DF AUTOLOAD (f . l)
    ; (AUTOLOAD fichier at1 ... atN)
    (MAPC (LAMBDA (at)
              (EVAL (LIST 'DM at 'l
                          (LIST 'REMFN (KWOTE at))
                          (LIST 'LOAD f t)
                          'l)))
          l)))
```

Ainsi l'appel :

```
(AUTOLOAD PRETTY PPRINT)
```

donne la définition suivante à la fonction PPRINT :

```
(DM PPRINT l
    (REMFN 'PPRINT)
    (LOAD PRETTY T)
    l)
```

ce qui fait qu'à la première évaluation de (PPRINT ...) la MACRO est appelée. Cette MACRO se détruit elle-même (pour éviter de boucler au cas où la fonction ne serait pas définie dans le fichier) puis évalue (LOAD PRETTY T) qui charge en silence le fichier PRETTY.
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

6.10.6 Chemins d'accès aux fichiers

Afin de ne pas spécifier systématiquement le chemin d'accès à un fichier, il est possible de spécifier un ensemble ordonné de préfixes ou bibliothèques. Cet ensemble se trouve dans la variable suivante:

#:SYSTEM:PATH [Variable]

contient la liste des catalogues où seront fait par défaut la recherche des fichiers. Au lancement de Le_Lisp cette variable contient les catalogues systèmes :

```
(DEFVAR #:SYSTEM:PATH
        (LIST ""
              #:SYSTEM:LLIB-DIRECTORY
              #:SYSTEM:LLUB-DIRECTORY
              #:SYSTEM:LLMOD-DIRECTORY
              #:SYSTEM:LLOBJ-DIRECTORY
              #:SYSTEM:LLTEST-DIRECTORY
              #:SYSTEM:VIRTTY-DIRECTORY
              #:SYSTEM:VIRBITMAP-DIRECTORY ))
```

(SEARCH-IN-PATH <path> <file>) [SUBR à 2 arguments]

retourne le nom complet du fichier <file> dont le catalogue appartient à la liste <path>. Si ce fichier n'existe pas la fonction SEARCH-IN-PATH retourne ().

SEARCH-IN-PATH peut être défini en Lisp de la manière suivante :

```
(DE SEARCH-IN-PATH (path file)
    (WHEN path
          (LET ((real-file (CATENATE (IF (CONSP path)
                                         (CAR path)
                                         path)
                                     file)))
               (IF (PROBEFILE real-file)
                   real-file
                   (WHEN (CONSP path)
                         (SEARCH-IN-PATH (CDR path)
                                         file))))))
```

(PROBEPATHF <file>) [SUBR à 1 argument]

si le fichier <file> ayant pour suffixe #:SYSTEM:LELISP-EXTENSION existe dans le système de catalogue #:SYSTEM:PATH, PROBEPATHF retourne le nom complet du fichier, sinon retourne ().

PROBEPATHF peut être défini en Lisp de la manière suivante :

```
(DE PROBEPATHF (file)
    (SEARCH-IN-PATH
        #:SYSTEM:PATH
        (CATENATE file #:SYSTEM:LELISP-EXTENSION)))
```

6.10.7 Accès aux bibliothèques

(LIBLOADFILE <file> <i>) [SUBR à 2 arguments]

est identique à la fonction LOADFILE mais le fichier est cherché dans les différents catalogues donnés par la variable #:SYSTEM:PATH. LIBLOADFILE retourne le nom réel du fichier chargé ou provoque une erreur si le fichier n'existe pas.

LIBLOADFILE peut être défini en Lisp de la manière suivante :

```
(DE LIBLOADFILE (file redef?)
    (LET ((real-file (PROBEPATHF file)))
         (IFN real-file
              (ERROR 'LIBLOADFILE "fichier inconnu" file)
              (LOADFILE real-file redef?) )))
```

(LIBLOAD <file> <i>) [FSUBR à 1 ou 2 arguments]

cette fonction est la forme FSUBRée de la fonction précédente. De plus l'argument <i> est optionnel et vaut () par défaut.

LIBLOAD peut être défini en Lisp de la manière suivante :

```
(DF LIBLOAD (file . redef?)
    (LIBLOADFILE file (CAR redef?)) )
ex : (LIBLOAD HANOI)
```

(LIBAUTOLOAD <file> <sym1> ... <symN>) [FSUBR]

est identique à la fonction AUTOLOAD toutefois les fichiers seront rechargés au moyen de la fonction LIBLOAD à la place de LOAD.

6.11 Les Traits (Features)

Le_Lisp peut gérer le nom des bibliothèques chargées en mémoire.

Voici les noms des traits de la bibliothèque standard :

DEFSTRUCT	définition des structures
COMPLEX	arithmétique sur les nombres complexes
RATIO	arithmétique sur les nombres rationnels
LIBCIR	bibliothèque circulaire
PRETTY	paragrapheur de S-expressions
DEBUG	outils de mise au point
LOADER	chargeur mémoire
COMPILER	compilateur standard ou complice
COMPLICE	compilateur complice
VIRTTY	terminal virtuel
EDLIN	éditeur de ligne en entrée
PEPE	éditeur pleine page
TERMCAP	traducteur de TERMCAP -> VIRTTY
TERMINFO	traducteur de TERMINFO -> VIRTTY
WINDOW	fenêtrage virtuel

#:SYSTEM:FEATURES-LIST [Variable]

contient le nom des traits chargés en mémoire.

(ADD-FEATURE <symb>) **[SUBR à 1 argument]**

rajoute dans la liste des traits chargés le nom <symb>.

(REM-FEATURE <symb>) **[SUBR à 1 argument]**

enlève de la liste des traits chargés le nom <symb>.

(FEATUREP <symb>) **[SUBR à 1 argument]**

retourne <symb> si le trait <symb> est présent dans la liste des traits chargés.

CHAPITRE 7

Les Fonctions Système

Les fonctions de ce chapitre vont permettre de créer des sous-systèmes écrits en Le_Lisp. Grâce aux interruptions programmables, il va être possible de définir de nouvelles boucles d'interaction et de nouveaux traitements d'erreurs.

7.1 Les Interruptions Programmables

Le_Lisp traite ses interruptions internes en appelant une fonction spécialisée propre à chacune des interruptions. Les noms de ces fonctions ne sont pas fixes, mais sont recherchés à chaque interruption. Cette recherche utilise le chemin de recherche #:SYS-PACKAGE:ITSOFT.

#:SYS-PACKAGE:ITSOFT [Variable]

contient la liste des packages où doivent être recherchées les fonctions associées aux interruptions internes. Par défaut cette variable vaut (), les fonctions sont donc recherchées uniquement dans le package global, ||. Toutes les interruptions programmables internes possèdent une fonction prédéfinie dans ce package, qui est donc la fonction invoquée par défaut.

La valeur de cette variable peut être une liste de symboles ou bien un symbole. Dans le premier cas la fonction associée à l'interruption programmable de nom FOO, est la première fonction de nom FOO trouvée dans l'un des packages de la liste. Dans le second cas, c'est la première fonction de nom FOO trouvée dans le package ayant le nom du symbole, ou dans l'un de ses packages pères.
La recherche de la fonction s'effectue donc horizontalement dans la liste de packages pour le premier cas, et verticalement, par remontée hiérarchique dans les packages pour le second. Elle est modélisée par la fonction suivante :

```
(DE GET-IT-HANDLER (lpacks fn)
    (COND ((SYMBOLP lpacks)
               (GETFN lpacks fn))
          ((CONSP lpacks)
               (OR (GETFN1 (CAR lpacks) fn)
                   (GET-IT-HANDLER (CDR lpacks) fn)))))
```

(ITSOFT <symb> <larg>) [SUBR à 2 arguments]

permet d'appeler explicitement l'interruption programmable de nom <symb> avec <larg> comme liste d'arguments. La définition de la fonction ITSOFT se rapproche de :

```
(DE ITSOFT (nom larg)
    (APPLY (GET-IT-HANDLER #:SYS-PACKAGE:ITSOFT nom) larg))
```

mais crée un bloc d'activation spécial et remet à () l'indicateur de trace de l'évaluateur.

(SUPER-ITSOFT <package> <symb> <larg>) [SUBR à 3 arguments]

invoque explicitement l'interruption programmable de nom <symb> avec <larg> comme liste d'arguments. Cette fonction suppose qu'il existe une fonction #:<package>:<symb> qui serait invoquée par la fonction ITSOFT. La fonction invoquée par SUPER-ITSOFT est la fonction qui serait invoquée par la fonction ITSOFT si la fonction #:<package>:<symb> n'existait pas.
Cette fonction est particulièrement utile pour mettre des pré ou post processeurs aux interruptions programmables. Par exemple, pour compter dans une variable le nombres de lignes imprimées à l'écran :

```
(DEFVAR #:COUNT:NLINES 0)

(DE #:COUNT:EOL ()
    (INCR #:COUNT:NLINES)
    (SUPER-ITSOFT 'COUNT 'EOL ()))

(SETQ #:SYS-PACKAGE:ITSOFT (CONS 'COUNT #:SYS-PACKAGE:ITSOFT))
```

L'exemple suivant montre l'implémentation d'un utilitaire ressemblant au MORE du système UNIX.

```
(DEFVAR #:SYS-PACKAGE:COLON 'MINIMORE)

(DEFVAR :COUNT (TYYMAX))
(DEFVAR :STRING "--MORE--")

(DE MORE ()
    (SETQ #:SYS-PACKAGE:ITSOFT
          (CONS 'MINIMORE #:SYS-PACKAGE:ITSOFT)))

(DE MOREND ()
    (SETQ #:SYS-PACKAGE:ITSOFT
          (DELQ 'MINIMORE #:SYS-PACKAGE:ITSOFT)))

(DE :BOL ()
    (SETQ :COUNT (TYYMAX))
    (SUPER-ITSOFT '#.#:SYS-PACKAGE:COLON 'BOL ()))

(DE :EOL ()
    (WHEN (= :COUNT 0)
          (TYSTRING :STRING (SLENGTH :STRING))
          (SELECTQ (TYI)
             ((#^M #^J) (SETQ :COUNT 1))
             (#^D       (SETQ :COUNT (DIV (TYYMAX) 2)))
             (#\SP      (SETQ :COUNT (TYYMAX)))
             (#/Q       (SETQ :COUNT (TYYMAX))
                        (FOR (I (SUB1 (SLENGTH :STRING)) -1 0)
                             (TYBACK (CHRNTH I :STRING)))
                        (FILLSTRING (OUTBUF) 0 #\SP (OUTPOS))
                        (OUTPOS 0)
                        (EXIT #:SYSTEM:TOPLEVEL-TAG))
             (T         (SETQ :COUNT 1)))
          (FOR (I (SUB1 (SLENGTH :STRING)) -1 0)
               (TYBACK (CHRNTH I :STRING)))))
```

```
(DECR :COUNT)
(SUPER-ITSOFT '#.#:SYS-PACKAGE:COLON 'EOL ()))
```

Le_Lisp possède 11 interruptions programmables internes qui sont :

BOL	début de ligne
EOL	fin de ligne
FLUSH	fin de tampon
EOF	fin de fichier
TOPLEVEL	boucle d'interaction
GCALARM	fin de GC
SYSERROR	erreur interne
STEPEVAL	passage dans EVAL
USER-INTERRUPT	interruption utilisateur
MOUSE	interruption machine souris
CLOCK	interruption machine horloge

mais il est tout a fait possible de se définir ses propres interruptions programmables.

7.2 Les Interruptions Machine

Le_Lisp gère trois interruptions machine :
- l'interruption utilisateur
- l'interruption horloge
- l'interruption souris

La gestion des ces interruptions peut être activée (autorisée) ou inhibée (masquée). Durant l'exécution de la fonction associée à ces interruptions machine programmables, les interruptions sont masquées. Il appartient à l'utilisateur de les autoriser s'il le faut.

(WITHOUT-INTERRUPTS <e1> ... <eN>) [FSUBR]

en masquant les interruptions machine, les différentes expressions <e1> ... <en> sont évaluées à la manière d'un PROGN. WITHOUT-INTERRUPTS retourne la valeur de <eN>.

(WITH-INTERRUPTS <e1> ... <eN>) [FSUBR]

en autorisant les interruptions machine, les différentes expressions <e1> ... <en> sont évaluées à la manière d'un PROGN. WITH-INTERRUPTS retourne la valeur de <eN>.

7.2.1 L'interruption utilisateur

Il est possible à tout moment de provoquer une *interruption utilisateur* en envoyant un caractère spécial au niveau du clavier. Ce caractère d'interruption est dépendant des systèmes mais est toujours présent.

USER-INTERRUPT [Interruption programmable]

est le nom de l'interruption machine programmable déclenchée par l'utilisateur.

(USER-INTERRUPT) [SUBR à 0 argument]

est la fonction par défaut lancée par l'interruption programmable USER-INTERRUPT.

USER-INTERRUPT peut être défini en Lisp de la manière suivante :

```
(DE USER-INTERRUPT ()
    (WITH-INTERRUPTS
        (ITSOFT 'SYSERROR '(BREAK BREAK ()))))
```

7.2.2 L'horloge temps réel

(CLOCKALARM <n>) [SUBR à 1 argument]

<n> est un nombre flottant indiquant le nombre de secondes que doit attendre le système avant de provoquer une seule interruption machine programmable CLOCK. Si <n> vaut 0.0, l'horloge temps réel est inactivée et toute ancienne demande d'interruption est perdue. Il est conseillé de ne pas utiliser des valeurs trop petites (< 0.001 secondes) car le temps d'appel de l'interruption programmable CLOCK est assez lent (surtout s'il s'y produit des changements de tâches (cf la fonction SUSPEND)).

CLOCK [Interruption programmable]

est le nom de l'interruption machine programmable déclenchée par l'horloge temps réel.

(CLOCK) [SUBR à 0 argument]

est la fonction par défaut lancée par l'interruption programmable CLOCK.

CLOCK peut être défini en Lisp de la manière suivante :

```
(DE CLOCK ())
```

7.2.3 La souris

MOUSE [Interruption programmable]

est le nom de l'interruption machine programmable déclenchée en cas d'événement souris (cf le chapitre 19).

(MOUSE) [SUBR à 0 argument]

est la fonction par défaut lancée par l'interruption programmable MOUSE.

MOUSE peut être défini en Lisp de la manière suivante :

```
(DE MOUSE ())
```

7.3 Le Multi Tâches Intégré

Le_Lisp version 15.2 possède un système multi-tâches intégré. Il ne comprend que trois fonctions de base.

7.3.1 Les fonctions de base

(SCHEDULE <fnt> <e1> ... <eN>) [FSUBR]
(SUSPEND) [SUBR à 0 argument]
(RESUME <env>) [SUBR à 1 argument]

SCHEDULE permet de définir un *séquenceur*, SUSPEND d'arreter une tâche et RESUME de la relancer.

Dans SCHEDULE, <fnt> est une fonction à un argument. Cette fonction évalue, à la manière d'un PROGN, les expressions <e1> ... <eN>. A priori la valeur retournée est celle de <eN>. Mais si au cours de ces évaluations, un appel explicite ou implicite (par le biais d'une interruption programmable) à la fonction SUSPEND se produit, le calcul courant est stoppé, l'environnement dynamique d'exécution est sauvé et la fonction <fnt> associée à la dernière fonction SCHEDULE rencontrée dynamiquement est appelée avec l'environnement sauvé comme argument. Enfin, RESUME est une fonction qui prend en argument un environnement fabriqué par la fonction SUSPEND et permet de reprendre un calcul interrompu.

Attention : seul l'environnement dynamique compris entre l'appel de SUSPEND et SCHEDULE est sauvé.

7.3.2 Les séquenceurs de base

Il est possible de définir au moyen des trois fonctions précédentes les séquenceurs de base suivants. Le partage du temps est réalisé au moyen de l'horloge temps réel. Ces fonctions se trouvent dans le fichier SCHEDULE de la bibliothèque standard.

(PARALLEL <e1> ... <eN>) [FEXPR]

lance en parallèle l'évaluation des différentes expressions <e1> ... <eN>. PARALLEL retourne toujours ().

(PARALLELVALUES <e1> ... <eN>) [FEXPR]

lance en parallèle l'évaluation des différentes expressions <e1> ... <eN>. PARALLELVALUES retourne la liste de ces évaluations.

(TRYINPARALLEL <e1> ... <eN>) [FEXPR]

lance en parallèle l'évaluation des différentes expressions <e1> ... <eN>. Dès qu'une évaluation se termine TRYINPARALLEL retourne sa valeur et stoppe les évaluations en suspens.

Utilisation de l'horloge temps réel pour gérer le
partage du temps entre les différentes tâches

```
(DEFVAR #:SYSTEM:CLOCK-TICK 0.05)

(DE CLOCK ()
    ; c'est l'horloge qui provoque les suspensions des tâches.
    (WHEN (DEBUG) (PRINCN #/.))
    (SUSPEND))

(DMD CLOCKSTART ()
     ; fait partir l'horloge
     `(CLOCKALARM #:SYSTEM:CLOCK-TICK))

(DMD CLOCKSTOP ()
     ; arrete l'horloge
     `(CLOCKALARM 0.))

(DF PARALLEL :ll
    (WITHOUT-INTERRUPTS
        (LET ((:l (APPEND :ll ())))
            (WHILE :l
                (SCHEDULE (LAMBDA (:v)
                                  (NEWR :l (LIST 'RESUME (KWOTE :v))))
                     (LET ((:e (NEXTL :l)))
                          (CLOCKSTART)
                          (WITH-INTERRUPTS (EVAL :e))
                          (CLOCKSTOP)))))))

(DF PARALLELVALUES :ll
    (WITHOUT-INTERRUPTS
        (LET ((:ltask ())
              (:l :ll)
              (:r (MAKELIST (LENGTH :ll) ()))
              (:i -1))
            (WHILE (OR :l :ltask)
                (SCHEDULE (LAMBDA (:v) (NEWR :ltask :v))
                    (CLOCKSTART)
                    (IFN :l
                         (RESUME (NEXTL :ltask))
                         (LET ((:e (NEXTL :l)))
                              (INCR :i)
                              (RPLACA (NTHCDR :i :r)
                                      (WITH-INTERRUPTS (EVAL :e)))))
                    (CLOCKSTOP)))
            :r)))

(DF TRYINPARALLEL :ll
    (WITHOUT-INTERRUPTS
        (LET ((:ltask ())
              (:l :ll))
            (TAG :return-value
                 (WHILE (OR :l :ltask)
                        (SCHEDULE (LAMBDA (:v) (NEWR :ltask :v))
                            (CLOCKSTART)
                            (LET ((:e (NEXTL :l)))
                                 (IF :e
                                     (EXIT :return-value
                                           (PROG1
```

```
                                                    (WITH-INTERRUPTS
                                                       (EVAL :e))
                                                (CLOCKSTOP))))
                                        (RESUME (NEXTL :ltask))))))))))
```

Exemples d'utilisateurs des séquenceurs parallèles

```
ex : (DE FIB (n)
         (COND ((= n 1) 1)
               ((= n 2) 1)
               (t (APPLY '+ (PARALLELVALUES (FIB (1- n))
                                            (FIB (- n 2))))))))

     (DE NEG (n)
         (IF (= n 0)
             'negatif
             (NEG (1+ n))))

     (DE POS (n)
         (IF (= n 0)
             'positif
             (POS (1- n))))

     (DE SIGN (n)
         ; une manière (pas très rapide) de tester le signe
         ; d'un nombre :
         (IF (= n 0)
             'zero
             (TRYINPARALLEL (NEG n) (POS n))))
```

7.4 Les Erreurs Provoquées par le Système Le_Lisp

SYSERROR [Interruption programmable]

A l'apparition d'une erreur durant une évaluation, l'interruption programmable de nom SYSERROR est activée. Elle va invoquer une fonction (dont le calcul du nom est expliqué en 7.1) avec trois arguments :
- le nom de la fonction qui a provoqué l'erreur,
- le type de l'erreur
- l'argument défectueux.

En standard un message décrivant l'erreur est édité dans le flux de sortie courant.

Ce message contient toujours le nom de la fonction ayant provoqué l'erreur, un message explicatif (en français ou en anglais) et l'argument défectueux.

Le_Lisp permet également de provoquer explicitement une erreur, au moyen de la fonction ERROR et de tester si une évaluation produit une erreur au moyen des fonctions CATCHERROR et ERRSET.

7.4.1 Le traitement standard des erreurs

(SYSERROR <symb> <s1> <s2>) [SUBR à 3 arguments]

est la fonction par défaut invoquée par l'interruption programmable ITSOFT. <symb> 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. SYSERROR va imprimer sur le flux de sortie courant un message en clair au moyen de la fonction PRINTERROR puis va retourner au top-level de Lisp après avoir délié toutes les variables. En mode DEBUG, il va être également possible d'entrer dans une boucle d'inspection avant de retourner au top-level pour inspecter l'état de certaines variables locales ou de retester certaines évaluations (cf le chapitre 11).

(PRINTERROR <symb> <s1> <s2>) [SUBR à 3 arguments]

imprime le clair d'un message d'erreur sous la forme :

```
** <symb> : <s1> : <s2>
```

<symb> est le nom de la fonction ayant provoqué l'erreur; <s1> le type de l'erreur; <s2> en général l'argument défectueux. Les types d'erreur système sont peu nombreux, de l'ordre de la trentaine.

PRINTERROR peut être défini en Lisp de la manière suivante :

```
(defvar ERRMAC "erreur de la machine")
(defvar ERRUDV "variable indefinie")
(defvar ERRUDF "fonction indefinie")
(defvar ERRUDM "methode indefinie")
(defvar ERRUDT "echappement indefini")
(defvar ERRBDF "mauvaise definition")
(defvar ERRWNA "mauvais nombre d'arguments")
(defvar ERRBPA "mauvais parametre")
(defvar ERRILB "liaison illegale")
(defvar ERRBAL "mauvaise liste d'arguments")
(defvar ERRNAB "pas de portee lexicale")
(defvar ERRXIA "bloc lexical perime")
(defvar ERRSXT "erreur de syntaxe")
(defvar ERRIOS "erreur d'entree/sortie")
(defvar ERR0DV "division par 0")
(defvar ERRNNA "l'argument n'est pas un nombre")
(defvar ERRNIA "l'argument n'est pas un entier")
(defvar ERRNFA "l'argument n'est pas un flottant")
(defvar ERRNSA "l'argument n'est pas une chaine")
(defvar ERRNAA "l'argument n'est pas un atome")
(defvar ERRNLA "l'argument n'est pas une liste")
(defvar ERRNVA "l'argument n'est pas une variable")
(defvar ERRVEC "l'argument n'est pas un vecteur")
(defvar ERRSYM "l'argument n'est pas un symbole")
(defvar ERRNDA "l'argument n'est pas une adresse")
(defvar ERRSTC "l'argument n'est pas une structure")
(defvar ERROOB "argument hors limite")
(defvar ERRSTL "chaine trop longue")
(defvar ERRGEN "ne sait pas calculer")
(defvar ERRVIRTTY "terminal inconnu")
```

```
(DE PRINTERROR (#:system:f #:system:m #:system:b)
    (PRINT "** " #:system:f
           " : " (SELECTQ #:system:m
                          ; les erreurs standard
                          (ERRMAC ERRMAC)
                          (ERRUDV ERRUDV)
                          (ERRUDF ERRUDF)
                          (ERRUDM ERRUDM)
                          (ERRUDT ERRUDT)
                          (ERRBDF ERRBDF)
                          (ERRWNA ERRWNA)
                          (ERRBPA ERRBPA)
                          (ERRILB ERRILB)
                          (ERRBAL ERRBAL)
                          (ERRNAB ERRNAB)
                          (ERRXIA ERRXIA)
                          (ERRSXT ERRSXT)
                          (ERRIOS ERRIOS)
                          (ERR0DV ERR0DV)
                          (ERRNNA ERRNNA)
                          (ERRNIA ERRNIA)
                          (ERRNFA ERRNFA)
                          (ERRNSA ERRNSA)
                          (ERRNAA ERRNAA)
                          (ERRNLA ERRNLA)
                          (ERRNVA ERRNVA)
                          (ERRVEC ERRVEC)
                          (ERRSYM ERRSYM)
                          (ERRNDA ERRNDA)
                          (ERRSTC ERRSTC)
                          (ERROOB ERROOB)
                          (ERRSTL ERRSTL)
                          (ERRGEN ERRGEN)
                          (ERRVIRTTY ERRVIRTTY)
                          (T          #:system:m))
           " : " (COND
                   ((AND (EQ #:system:m 'ERRSXT)
                         (NUMBERP #:system:b)
                         (> #:system:b 0) (< #:system:b 13))
                    (SELECTQ #:system:b
                         (1   "liste trop courte")
                         (2   "chaine trop longue")
                         (3   "symbole trop long")
                         (4   "mauvais debut d'expression")
                         (5   "symbole special trop long")
                         (6   "mauvais package")
                         (7   "mauvaise construction pointee")
                         (9   "mauvaise liste argument")
                         (10  "mauvaise valeur de splice-macro")
                         (11  "EOF durant un READ")
                         (12  "mauvaise utilisation du backquote")
                         (t   #:system:b)))
```

```
((AND (MEMQ (SYSTEM) UNIX)
      (EQ #:system:m 'ERRMAC)
      (NUMBERP #:system:b))
 (SELECTQ #:system:b
      (4   "instruction illegale")
      (8   "exception flottante")
      (10  "erreur de bus")
      (11  "violation de segment")
      (t #:system:b)))
(T #:system:b))))
```

#:SYSTEM:ERROR-FLAG [Variable]

contient l'état de la variable indiquant s'il faut imprimer un message en cas d'erreur.

7.4.2 Appel explicite et test d'erreur

(ERROR <s1> <s2> <s3>) [SUBR à 3 arguments]

permet d'appeler explicitement une erreur. <s1> doit être le nom de la fonction qui provoque l'erreur, <s2> le type de l'erreur et <s3> le ou les arguments défectueux.

ERROR peut être défini en Lisp de la manière suivante :

```
(DMD ERROR l
     `(ITSOFT 'SYSERROR ,l))
```

(CATCHERROR <i> <s1> ... <sN>) [FSUBR]

<i> est un indicateur qui est évalué et qui sera contenu dans la variable système #:SYSTEM:ERROR-FLAG puis les différentes expressions <s1> ... <sN> sont évaluées en séquence. Si au cours de l'une de ces évaluations une erreur se produit, CATCHERROR retourne immédiatement () et en fonction de l'état de l'indicateur #:SYSTEM:ERROR-FLAG imprime ou non un message d'erreur au moyen de la fonction PRINTERROR. Si les évaluations ne provoquent aucune erreur, CATCHERROR retourne toujours une liste d'un élément qui est la valeur de la dernière expression évaluée c'est-à-dire celle de <sN> (ceci pour distinguer une valeur retournée égale à () d'une erreur dans l'évaluation).

(ERRSET <e> <i>) [MACRO]

est une forme historique de la fonction précédente et n'existe dans Le_Lisp que pour des raisons de compatibilité.

ERRSET peut être défini en Lisp de la manière suivante :

```
(DMD ERRSET (e i) `(CATCHERROR ,i ,e))
```

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

ERR évalue les différentes expressions <s1> ... <sN>; la valeur de <sN> est la valeur retournée du dernier CATCHERROR (ou du TOPLEVEL si aucun CATCHERROR 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.

7.4.3 Exemples de traitements spéciaux

Voici quelques exemples de traitements spéciaux en cas d'erreur.

Fonctionnement normal de l'erreur ERRUDV (variable indéfinie)

```
? (GENSYM)
= G101
? (EVAL (GENSYM))
** EVAL : variable indéfinie : G102
```

redéfinition de l'erreur ERRUDV : la variable indéfinie est mise à ()

```
?
? (DE #:PIERRE:SYSERROR (F M A)
?     (IF (EQ M 'ERRUDV)
?         (SET A ())
?         (SYSERROR F M A)))
= #:PIERRE:SYSERROR
?
? (SETQ #:SYS-PACKAGE:ITSOFT 'PIERRE)
= PIERRE
?
? (SETQ X (GENSYM))
= G103
? (EVAL X)
= ()
? X
= G103
? (EVAL 'G103)
= ()
```

redéfinition de la même erreur : les variables indéfinies sont transformées en constantes.

```
? (DE #:CHRISTIAN:SYSERROR (F M A)
?     (IF (EQ M 'ERRUDV)
?         (SET A A)
?         (SYSERROR F M A)))
= #:CHRISTIAN:SYSERROR
?
? (SETQ #:SYS-PACKAGE:ITSOFT 'CHRISTIAN)
= CHRISTIAN
?
? (SETQ X (GENSYM))
= G104
? (EVAL X)
= G104
? (EVAL 'G104)
= G104
```

retour au traitement normal

```
? (SETQ #:SYS-PACKAGE:ITSOFT ())
```

```
= ()
?
? (EVAL (GENSYM))
** EVAL : variable indéfinie : G105
```

redéfinition de l'erreur ERRUDF

```
? (DE DANGER (f l)
?     (f l))
= DANGER
? (DANGER 'CAR '(a b c))
** EVAL : fonction indéfinie : f
?
? (DE #:ODD:SYSERROR (#:system:f #:system:m #:system:a)
?     (IF (AND (EQ #:system:f 'EVAL)
?              (EQ #:system:m 'ERRUDF))
?         (EVAL #:system:a)
?         (SYSERROR #:system:f #:system:m #:system:a)))
= #:ODD:SYSERROR
? (DEFVAR #:sys-package:itsoft 'odd)
= ODD
? (DANGER 'CAR '(a b c))
? A     ; s'il n'existe pas de fonction de non F dans le système.
```

7.5 L'Accès à l'Interprète

Le_Lisp possède trois fonctions de contrôle interne de l'interprète. Ces fonctions sont utilisées pour s'assurer du bon fonctionnement de l'interprète et pour réaliser les outils de mise au point décrits dans les chapitres suivants, en particulier les traces, les exécutions incrémentales et les contrôles dynamiques.
L'évaluateur possède un indicateur interne, qui n'est pas une variable Lisp pour des raisons d'efficacité. Quand cet indicateur est positionné, tout appel interne à l'évaluateur, c'est-à-dire à la fonction EVAL, provoque une interruption programmable de nom STEPEVAL.

STEPEVAL [Interruption programmable]

cette interruption programmable va invoquer une fonction avec comme argument la valeur qui devait être évaluée.

(STEPEVAL <e> <env>) [SUBR à 2 arguments]

est la fonction par défaut appelée par l'interruption programmable STEPEVAL. L'argument <e> est la forme qui devait être évaluée et <env> l'environnement lexical d'évaluation. Par défaut, STEPEVAL imprime dans le flux de sortie courant :

```
-->   la forme qui devait être évaluée
<--   le résultat de l'évaluation de cette forme.
```

STEPEVAL peut être défini en Lisp de la manière suivante :

```
(DE STEPEVAL (e env)
    (PRINT "-->" e)
    (PRINT "<--" (TRACEVAL e env)))
```

(TRACEVAL <e> <env>) [SUBR à 1 ou 2 arguments]

évalue l'expression <e> en mode trace, c'est-à-dire avec l'indicateur interne de trace positionné. L'argument <env>, s'il est fourni, est l'environnement lexical d'évaluation (donné par la fonction STEPEVAL). TRACEVAL retourne la valeur de <e> évaluée en mode trace.

(CSTACK) [SUBR à 0 argument]

A chaque entrée dans une fonction, un échappement, un verrou ... l'interprète fabrique un bloc d'activation dans la pile d'exécution dynamique. CSTACK retourne les blocs d'activation 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 types des blocs actuellement fabriqués par le système.

```
1     LAMBDA
2     FLET
3     TAG
4     ITSOFT
5     LOCK
6     PROTECT
7     SYSPROT
8     SCHEDULE
9     TAGBODY
10    BLOCK
```

7.6 Les Fichiers Image-Mémoire

Il est possible de sauver à tout moment l'image de tout le système pour la restaurer par la suite. Ce trait est particulièrement utile pour sauver des sous-systèmes compilés et pouvoir les rappeler très rapidement.
ATTENTION : à la restauration d'une image mémoire, l'exécution reprend à l'endroit exact où elle avait été interrompue toutefois les canaux d'entrées/sorties ainsi que d'autres paramètres du système d'exploitation hôte (timers ...) ne sont pas restaurés.

#:SYSTEM:CORE-DIRECTORY [Variable]

contient le nom du catalogue contenant les images-mémoire standard du système.

#:SYSTEM:CORE-EXTENSION [Variable]

contient l'extension par défaut des fichiers image-mémoire.

(SAVE-CORE <file>) [SUBR à 1 argument]

sauve la mémoire actuelle dans le fichier de nom <file>. SAVE-CORE retourne toujours T en valeur.

(RESTORE-CORE <file>) [SUBR à 1 argument]

restaure une image-mémoire à partir du fichier <file>. RESTORE-CORE ne retourne pas de valeur propre car il continue l'évaluation à l'endroit même où elle avait été stoppée par le SAVE-CORE correspondant.

; Voici des exemples d'utilisation :

```
? (progn  (print 1)
?         (save-core 'qwer)
?         (print 2))
  1
  2
= 2
? (restore-core 'qwer)
  2
= 2
```

; voici une bonne manière de sauver une image-mémoire
; de façon à obtenir un sous-système :

```
? (progn
?    (save-core 'bar)
?    (herald)
?    (initty)
?    "vous êtes sous le système : bar")
  ******* Le_Lisp (by INRIA) version 15.2 (20/Mai/86) [system]
= vous êtes sous le système : bar
```

; et pour le recharger

```
? (restore-core 'bar)
  ******* Le_Lisp (by INRIA) version 15.2 (20/Mai/86) [system]
= vous êtes sous le système : bar
```

; encore un exemple montrant la suspension
; de l'interprétation :

```
? (de foo (n)
?     (if (> n 0)
?         (prog1 (* n (foo (1- n)))
?                (print n))
?         (save-core 'foo)
?         (print 'au-fond)
?         1))
= foo
? (foo 4)
 au-fond
 1
```

```
 2
 3
 4
= 24

? (restore-core 'foo)
 au-fond
 1
 2
 3
 4
= 24
```

; Un truc à éviter, pour ne pas faire boucler le système :

```
? (progn
?    (print "j'y vais")
?    (save-core 'qaz)
?    (print "je continue")
?    (restore-core 'qaz)
?    (print "j'ai termine"))
 j'y vais
 je continue
 je continue
 je continue
    .....
```

7.7 Les Fonctions d'Installation

Les fonctions de cette section permettent de charger et de compiler l'environnement standard ou l'environnement modulaire du système Le_Lisp. Elles sont utilisées uniquement lors de la construction des images mémoire, c'est-à-dire lors de l'installation du système sur un nouveau site.

Ces fonctions sont définies dans le fichier STARTUP de la bibliothèque standard, qui est chargé automatiquement lors du lancement du système si l'on ne demande pas la restauration d'une image mémoire.

(LOAD-STD ‹im› ‹min› ‹ed› ‹env› ‹ld› ‹cmp›) [SUBR de 1 à 6 arg.]

Charge tout ou partie de l'environnement standard interprété, et construit éventuellement une image mémoire. Cette fonction accepte de 1 à 6 arguments, qui sont des indicateurs. Les indicateurs non fournis sont positionnés à ().
Si l'indicateur ‹im› n'est pas (), il est passé à la fonction SAVE-STD après le chargement de l'environnement, pour créer une image mémoire de nom ‹im›.
L'indicateur ‹min› indique que l'on désire charger l'environnemnt minimum nécessaire au fonctionnement du système : toplevel, terminal et bitmap virtuels.
L'indicateur ‹ed› indique que l'on désire charger l'éditeur plein écran PEPE.
L'indicateur ‹env› indique que l'on désire charger l'environnement de développement Le_Lisp : structures, tris, tableaux, procédures externes, pistage, impression formattée, arithmétique générique et boucle d'inspection.
L'indicateur ‹ld› indique que l'on désire charger le chargeur mémoire, qui permettra de charger des modules Lisp compilés, et des programmes LAP.

L'indicateur <cmp> indique que l'on désire charger le compilateur standard Le_Lisp. Le chargement du compilateur provoque le chargement du chargeur mémoire si celui-ci n'est pas présent dans l'environnement Lisp.

```
ex : Chargement de tout l'environnement, sans création d'image mémoire
     (LOAD-STD () T T T T T)
```

(LLCP-STD <nom>) [SUBR à 1 argument]

compile l'environnement interprété qui a été chargé par la fonction LOAD-STD. Si l'argument <nom> est différent de (), il est passé à la fonction SAVE-STD, après la compilation, pour construire une image mémoire du système compilé.
Il faut absolument utiliser cette fonction pour compiler l'environnement Lisp car cette compilation pose de délicats problèmes d'initialisations que la fonction COMPILE-ALL-IN-CORE ne sait pas résoudre (compilation du chargeur mémoire notamment).

```
ex : Chargement de l'environnement, compilation,
     et sauvegarde dans COMPILE
        (LOAD-STD () T T T T T)
        (LLCP-STD 'COMPILE)
```

(LOAD-CPL <im> <min> <ed> <env> <ld> <cmp>) [SUBR de 1 à 6 arg.]

Cette fonction est identique à la fonction LOAD-STD mais elle charge les modules compilés au lieu de charger les modules interprétés. De plus l'indicateur <cmp> indique ici que l'on désire charger le compilateur modulaire Complice, et non le compilateur standard.
L'environnement chargé par cette fonction étant déja compilé, il ne faut pas utiliser la fonction LLCP-STD après un appel à la fonction LOAD-CPL.

```
ex : Chargement de l'environnement de développement compilé et
     sauvegarde dans MONLISP
     (LOAD-CPL "monlisp" T () T)
```

(SAVE-STD <nom> <msg> <features>) [SUBR à 2 ou 3 arguments]

<nom> est une chaîne de caractères ou un symbole Lisp. Cette fonction construit une image mémoire de nom <nom>, suffixé par le suffixe #:SYSTEM:CORE-EXTENSION. Cette image est créée dans le catalogue #:SYSTEM:CORE-DIRECTORY.
A la restauration de cette image mémoire le terminal virtuel puis le bitmap virtuels sont initialisés (par les fonctions INITTY et INIBITMAP), la bannière est imprimée (par la fonction HERALD) et un fichier de démarrage dont le nom dépend des systèmes est éventuellement chargé. Le système imprime ensuite le message de bienvenue et la liste des traits chargés dans le système, décrite par l'argument <feature>, qui est une liste d'indicateurs (<ed> <env> <ld> <cmp>). La boucle d'interaction principale (fonction TOPLEVEL) est finalement lancée.
Cette fonction est utilisée par les fonction LOAD-STD, LOAD-CPL et LLCP-STD lorsque leur paramètre <im> n'est pas ().
Différentes erreurs peuvent se produire à la restauration de l'image mémoire créée par la fonction SAVE-STD. Ces erreurs viennent souvent de l'initialisation du terminal et du bitmap virtuels. Voir la documentation des fonction INITTY et INIBITMAP.

```
ex : créer l'image mémoire TOUT
   ? (SAVE-STD "foo" "J'ai tout chargé:" '(t t t t t t))
   Attendez, je sauve :
   J'ai tout chargé: editeur, environnement, chargeur et compilateur.
   = J'ai tout chargé: editeur, environnement, chargeur et compilateur.
```

7.8 L'Appel et la Sortie de l'Interprète

(HERALD) [SUBR à 0 argument]

imprime sur le terminal d'entrée la bannière du système. Si vous trouvez une erreur dans le système, avant de la signaler, appelez cette fonction pour déterminer le type exact du système que vous utilisez.

(SYSTEM) [SUBR à 0 argument]

retourne le nom du système Le_Lisp. Ce nom est dépendant du système que vous utilisez.

(VERSION) [SUBR à 0 argument]

retourne le numéro de version du système Le_Lisp sous la forme d'un nombre (aujourd'hui la 15.2).

```
ex : (VERSION)              ->   15.2
     (FLOATP (VERSION))     ->   15.2
     (> (VERSION) 15)       ->   15.2
```

(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 utilisé pour sortir définitivement de l'interprète Le_Lisp et pour revenir au moniteur standard. END est la seule fonction prédéfinie qui ne retourne pas de valeur...

```
ex : ? (END)
     Que Le_Lisp soit avec vous.

     %        retour au système hôte
```

7.9 Le TOP-LEVEL

La boucle principale (ou TOP-LEVEL) de l'interprète consiste à évaluer indéfiniment la forme :

```
(WHILE T (ITSOFT 'TOPLEVEL ()))
```

C'est donc l'interruption programmable TOPLEVEL qui détermine le mode de fonctionnement de l'interprète. Cette interruption est bien évidemment redéfinissable par l'utilisateur qui désire se construire son propre système.

TOPLEVEL [Interruption programmable]

est le nom de l'interruption programmable lancée par le top-level Le_Lisp.

(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

#:TOPLEVEL:STATUS [Variable]

cette variable contient l'indicateur 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.

#:TOPLEVEL:READ [Variable]

contient la dernière forme lue par le toplevel.

#:TOPLEVEL:EVAL [Variable]

contient la dernière valeur calculée par le toplevel.

#:SYSTEM:TOPLEVEL-TAG [Echappement]

est le nom de l'échappement qui permet de retourner directement au toplevel.

A l'initialisation du système la fonction TOPLEVEL est équivalente à :

```
(DEFVAR #:TOPLEVEL:STATUS ())
(DEFVAR #:TOPLEVEL:READ   ())
(DEFVAR #:TOPLEVEL:CREAD  ())
(DEFVAR #:TOPLEVEL:EVAL   ())

(DE TOPLEVEL ()
    (TAG #:SYSTEM:TOPLEVEL-TAG
         (SETQ #:TOPLEVEL:READ  #:TOPLEVEL:CREAD
               #:TOPLEVEL:CREAD (READ)
               #:TOPLEVEL:EVAL  (EVAL #:TOPLEVEL:CREAD))
         (WHEN #:TOPLEVEL:STATUS
               (PRINT "= " #:TOPLEVEL:EVAL)))
```

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))
```

7.10 Le Garbage-Collector

Les zones mémoire qui contiennent les objets Lisp sont allouées dynamiquement. Quand l'une de ces zones 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.

Si la zone contenant les listes est pleine le libellé est :

```
***** Erreur fatale : zone des listes pleine.      ou bien
***** Fatal error : no room for lists.
```

si la zone contenant les vecteurs est pleine :

```
***** Erreur fatale : zone des vecteurs pleine.      ou bien
***** Fatal error : no room for vectors.
```

si la zone contenant les chaînes de caractères est pleine :

```
***** Erreur fatale : zone des chaînes pleine.      ou bien
***** Fatal error : no room for strings.
```

si la zone contenant les symboles est pleine :

```
***** Erreur fatale : zone des symboles pleine.      ou bien
***** Fatal error : no room for symbols.
```

si la zone contenant les nombres flottants est pleine :

```
***** Erreur fatale : zone des flottants pleine.      ou bien
***** Fatal error : no room for floats.
```

si la zone contenant les nombres entiers est pleine :

```
***** Erreur fatale : zone des entiers pleine.      ou bien
***** Fatal error : no room for fixs.
```

Le_Lisp gère également deux autres espaces qui ne contiennent pas d'objets Lisp :

- l'*espace code* qui contient le code engendré par le compilateur. Cet espace est géré par les différents chargeurs. S'il n'y a plus de place, une erreur fatale se produit dont le libellé est :

```
***** Erreur fatale : zone du code pleine.      ou bien
***** Fatal error : no room for code.
```

- l'*espace tas* (HEAP) qui contient les valeurs des chaînes de caractères, les valeurs des vecteurs de S-expressions ... Cet espace est géré dynamiquement et est compacté en cas de manque de place. Si cet essai s'avère infructueux, une erreur fatale se produit dont le libellé est :

```
***** Erreur fatale : zone du tas pleine.      ou bien
***** Fatal error : no room for heap.
```

Enfin Le_Lisp utilise une pile unique d'exécution dynamique pour le contrôle et les données. En cas de débordement de pile une erreur fatale se produit dont le libellé est :

```
***** Erreur fatale : pile pleine.      ou bien
***** Fatal error : stack overflow.
```

(GC ‹i›) [SUBR à 0 ou 1 argument]

permet d'appeler le *garbage-collector* explicitement. Si l'argument est absent, GC retourne T. Si l'argument est présent, GC retourne une liste identique à celle retournée par la fonction suivante GCINFO. De plus une compaction de l'espace tas est également réalisée.

(GCINFO) [SUBR à 0 argument]

retourne une liste contenant les informations concernant la dernière récupération de mémoire. Cette liste a la forme :

```
(GC ‹n1› ‹n2› ‹n3› ‹n4› ‹n5› ‹n6› ‹n7› ‹n8›
    CONS   ‹ncons›
    SYMBOL ‹nsymb›
    STRING ‹nstrg›
    VECTOR ‹nvect›
    FLOAT  ‹nfloat›
    FIX    ‹nfix›
    HEAP   ‹nheap›
    CODE   ‹ncode›)
```

dans laquelle :
‹n1› est le nombre total de GC dus à un manque de cellules de liste effectués depuis le début de la session,
‹n2› est le nombre total de GC dus à un manque de symboles effectués depuis le début de la session,
‹n3› est le nombre total de GC dus à un manque de chaînes de caractères effectués depuis le début de la session,
‹n4› est le nombre total de GC dus à un manque de vecteurs effectués depuis le début de la session,
‹n5› est le nombre total de GC dus à un manque de nombres flottants effectués depuis le début de la session,
‹n6› est le nombre total de GC dus à un manque de nombres entiers effectués depuis le début de la session,
‹n7› est le nombre total de GC dus à un manque de place dans le tas.
‹n8› est le nombre total de GC dus à un appel explicite de la fonction GC.

Les autres valeurs indiquent la taille restante dans chacune des zones. Ces valeurs sont exprimées en *nombre d'objets* si celui-ci est suffisamment petit ou bien en *k objets*. Dans ce cas le nombre est dans une liste.
‹ncons› est le nombre de cellules de liste restant dans la zone des listes.
‹nsymb› est le nombre de symboles restant dans la zone symbole.
‹nstrg› est le nombre de chaînes restant dans la zone chaîne.
‹nvect› est le nombre de vecteurs restant dans la zone vecteur.
‹nfloat› est le nombre de nombres flottants restant dans la zone des nombres flottants.
‹nfix› est le nombre de nombres fixes restant dans la zone des nombres fixes (si celle-ci existe).
‹nheap› est la taille restante de la zone tas.
‹ncode› est la taille restante de la zone code.

GCALARM [Interruption programmable]

est une interruption programmable lancée automatiquement par le système après chaque récupération de la mémoire. Cette interruption permet de contrôler dynamiquement le fonctionnement du récupérateur ou bien d'éviter les erreurs fatales vues précédemment en provoquant une erreur normale pendant qu'il est encore temps.

(GCALARM) [SUBR à 0 argument]

en standard cette fonction ne fait rien.
La redéfinition qui suit de GCALARM va permettre, en cas de diminution trop importante du nombre des cellules de liste, de provoquer une erreur douce :

```
(DE GCALARM ()
    (LET ((nbcons (CADR (MEMQ 'CONS (GCINFO)))))
         (WHEN (AND (FIXP nbcons) (< nbcons 500))
               (PRINT "J'arrete les frais.")
               (ERROR 'GCALARM "nb de cellules restantes" nbcons)))
```

(FREECONS <cons>) [SUBR à 1 argument]

permet de rendre à la liste libre des CONS, le CONS <cons>. Cette fonction autorise, dans des cas très clairement étudiés, l'amélioration des performances du gestionnaire de mémoire en explicitant incrémentalement les objets qui ne seront plus utilisés. FREECONS n'est pas interruptible et retourne toujours ().
Attention : cette fonction peut définitivement perturber le gestionnaire de mémoire s'il existait des pointeurs sur le CONS libéré. FREECONS ne peut pas être décrit en Lisp d'une manière simple.

```
(DE NUMBER-of-CONS ()
    ; retourne le nombre de CONS de la liste libre des CONS
    (LET ((x (CONS () ())) (n 0))
         (FREECONS x)
         (WHILE x (INCR n) (NEXTL x))
         n))
```

(FREETREE <tree>) [SUBR à 1 argument]

permet de rendre à la liste libre des CONS, l'arbre <tree> L'arbre <tree> ne peut pas être une liste circulaire ou partagée sous peine de détruire tout le système. Comme la fonction précédente, FREETREE autorise, dans des cas très clairement étudiés, l'amélioration des performances du gestionnaire de mémoire en explicitant incrémentalement les objets qui ne seront plus utilisés. FREETREE n'est pas interruptible et retourne toujours ().
Attention : cette fonction peut définitivement perturber le gestionnaire de mémoire s'il existait des pointeurs sur l'arbre libéré.

FREETREE peut être défini en Lisp de la manière suivante :

```
(DE FREETREE (s)
    (WHEN (CONSP s)
          (FREECONS s)
          (FREETREE (CAR s))
          (FREETREE (CDR s))))
```

7.11 Autres Accès au Système

Le_Lisp permet d'accéder à d'autres caractéristiques du système. Vu les différences considérables entre les systèmes, il est sage, avant d'utiliser les fonctions qui vont suivre, de tester d'abord leurs effets sur des cas simples.

(RUNTIME) [SUBR à 0 argument]

retourne le temps CPU utilisé depuis l'appel du système Le_Lisp. Ce temps est donné en secondes entières ou flottantes, en fonction du système Le_Lisp utilisé.

(TIME <e>) [SUBR à 1 argument]

retourne le temps utilisé par l'unité centrale pour évaluer l'expression <e>. Comme pour la fonction précédente, le temps est donné en secondes entières ou flottantes en fonction du système Le_Lisp utilisé.

TIME peut être défini en Lisp de la manière suivante :

```
(DE TIME (e)
    (LET ((runt (RUNTIME)))
         (EVAL e)
         (- (RUNTIME) runt)))
```

```
ex : (TIME '(FIB 20))   ->   4.25
```

(DATE) [SUBR à 0 argument]

retourne, sous la forme d'un vecteur typé (de type DATE), la date et l'heure locale.

Voici le contenu de ce vecteur :
- 0 : l'année (à partir de la naissance de JC)
- 1 : le numéro du mois (Janvier=1, Décembre=12)
- 2 : le numéro du jour dans le mois (1-31)
- 3 : l'heure (0-23)
- 4 : la minute (0-59)
- 5 : la seconde (0-59)
- 6 : le nb de millisecondes (0-999)
- 7 : le numéro du jour dans la semaine (Lundi=1 ... Dimanche=7)

Si un champ n'est pas rempli, sa valeur reste à ().

Le fichier DATE de la bibliothèque standard contient la fonction qui imprime le vecteur typé de type DATE sous forme lisible.

```
? (date)
= #:date:#[1986 5 26 14 42 44 () 1]
? ^Ldate
= /udd/lelispv15.2/llib/date.ll
? (date)
= "lun 26 mai  86 14:42:51 "
```

(SLEEP <n>) [SUBR à 1 argument]

demande au système hôte de mettre Le_Lisp en état inactif durant <n> secondes. <n> est donné en secondes entières ou flottantes en fonction du système utilisé.

(COMLINE <strg>) [SUBR à 1 argument]

envoie au système d'exploitation hôte la ligne de commande <strg>. Cette fonction n'est pas disponible sur tous les systèmes Le_Lisp.
Il existe un macro-caractère qui, au niveau principal, permet de simplifier cette écriture, le macro-caractère point d'exclamation : !

```
ex : ? (COMLINE "pwd")
     /udd/lelisp/chailloux/ll/work
     -> T
     ? (COMLINE 'nil)
     sh: nil: not found
     ? !ls -ls

     (COMLINE "dir")
```

(GETENV <strg>) [SUBR à 1 argument]

envoie au système hôte le nom <strg> qui est supposé être le nom d'une variable système. GETENV retourne une chaîne de caractères qui est la valeur associée à ce symbole ou bien la chaîne vide "" si cette valeur n'existe pas. Cette fonction n'est pas disponible sur tous les systèmes.

```
ex : (getenv "TERM")   ->  hp
     (getenv "TTY")    ->  VT100
```

CHAPITRE 8

Le Paragrapheur de S-Expressions

Les fonctions prédéfinies PRINT et PRIN sont d'ordinaire utilisées pour éditer les S-expressions Le_Lisp. Les seules mesures prises pour améliorer la lisibilité sont :
- l'insertion d'un espace entre chaque atome ;
- l'interdiction d'éditer un atome (symbole ou nombre) à cheval sur deux lignes.

Ces mesures sont nettement insuffisantes pour éditer vos programmes. Les fonctions du paragrapheur (ou Pretty-Print en anglais) vont les éditer d'une manière beaucoup plus lisible en faisant ressortir, au moyen de renfoncements à gauche et de sauts de lignes ad hoc, la structure de contrôle de vos fonctions.

Pour améliorer encore la lisibilité, les appels de la fonction QUOTE sont représentés au moyen du macro-caractère ', et les macro-générations des LET sous leur forme d'entrée.

Pour démontrer l'utilité du paragrapheur, voici une fonction telle qu'elle est imprimée par les fonctions normales :

```
? (GETDEF '#:PRETTY:COND)
= (DE #:PRETTY:COND () (WITH ((LMARGIN (+ (LMARGIN) 3))) (WHILE
(CONSP L) (TERPRI) (IF (#:PRETTY:INLINEP (CAR L)) (NEXTL L) (
PRINCN 40) (LET ((L (NEXTL L)) (F T)) (#:PRETTY:P (NEXTL L)) (
WHEN L (#:PRETTY:PROGN))) (PRINCN 41)))))
```

Et voici cette même fonction traitée par le paragrapheur :

```
? (PRETTY #:PRETTY:COND)
(DE #:PRETTY:COND ()
    (WITH ((LMARGIN (+ (LMARGIN) 3)))
          (WHILE (CONSP L)
                 (TERPRI)
                 (IF (#:PRETTY:INLINEP (CAR L))
                     (NEXTL L)
                     (PRINCN 40)
                     (LET ((L (NEXTL L)) (F T))
                          (#:PRETTY:P (NEXTL L))
                          (WHEN L (#:PRETTY:PROGN)))
                     (PRINCN 41)))))
```

8.1 Les Fonctions du Paragrapheur

Les fonctions qui vont être décrites se trouvent dans un fichier de la bibliothèque standard de nom PRETTY et sont AUTOLOAD, c'est-à-dire qu'elles sont chargées automatiquement au premier appel de l'une d'elles.

PRETTY [Feature]

ce trait indique que le paragrapheur est présent en mémoire.

(PPRINT <s>) [EXPR à 1 argument]

paragraphe l'expression <s> dans le flux de sortie courant et saute une ligne. PPRINT retourne <s> en valeur.

(PPRIN <s>) [EXPR à 1 argument]

paragraphe l'expression <s> dans le flux de sortie courant et reste sur la même ligne. PPRIN retourne <s> en valeur.

(PRETTY <sym1> ... <symN>) [FEXPR à N arguments]

<sym1> ... <symN> sont des symboles qui possèdent une définition de fonction. PRETTY va éditer ces fonctions dans le flux de sortie courant et retourner () en valeur. Cette fonction a également connaissance des fonctions redéfinies ou tracées, il utilisera donc les véritables définitions de ces fonctions. Il existe un macro-caractère qui facilite cette écriture, le macro-caractère ^P.

(PRETTYF <file> <sym1> ... <symN>) [FEXPR à N arguments]

est identique à la fonction précédente mais édite les fonctions <sym1> ... <symN> dans un fichier de sortie de nom <file>. PRETTYF retourne <file> en valeur.

; pour créer un fichier de nom foo.ll contenant le texte
; des définitions des fonctions foo et bar, évaluez :

```
(PRETTYF "foo.ll" foo bar)
```

(PRETTYEND) [EXPR à 0 argument]

permet de récupérer la place occupée par les fonctions du paragrapheur après usage (de l'ordre de 700 cellules de liste). Les fonctions du paragrapheur redeviennent AUTOLOAD et le trait PRETTY disparait.

8.2 Le contrôle des Fonctions du Paragrapheur

Le paragrapheur utilise les fonctions LMARGIN et RMARGIN pour calculer la taille de la ligne utilisable. Deux variables permettent de régler l'impression des constantes dans les programmes :

#:PRETTY:QUOTELEVEL [Variable]

contient la valeur à donner à la fonction PRINTLEVEL en cas d'impression de constante dans les programmes. Par défaut cette valeur vaut 0.

#:PRETTY:QUOTELENGTH [Variable]

contient la valeur à donner à la fonction PRINTLENGTH en cas d'impression de constante dans les programmes. Par défaut cette valeur vaut 0.

8.3 Les Formats Standard du Paragrapheur

Par défaut le paragrapheur édite sur une même ligne les S-expressions. Si celle-ci déborde, il recommence son édition sur une nouvelle ligne (après avoir réalisé un renfoncement à gauche) en utilisant un des 7 formats standard suivants :

Format 1 : de type PROGN

```
(PROGN
   <e1>
   ....
   <eN>)
```

Format 2 : de type IF

```
(IF <e1>
    <e2>
    ....
    <eN>)
```

Format 3 : de type DE

```
(DE <e1> <e2>
    <e3>
    ....
    <eN>)
```

Format 4 : de type COND

```
(COND
      (<e11>
         <e12> ... <e1x>)
      ....
      (<eN1>
         <eN2> ... <eNz>))
```

Format 5 : de type SELECTQ

```
(SELECTQ <e1>
     (<e21>
         <e22> ... <e2x>)
      ....
      (<eN1>
         <eN2> ... <eNz>))
```

Format 6 : de type SETQ

```
(SETQ <e1> <e2>
         ....
      <eN-1> <eN>)
```

Format 7 : de type TAGBODY

```
(TAGBODY
        <e1>
  <label>
        ...
        <eN>)
```

Ces différents formats sont rangés dans le P-type des symboles. L'accès à ce P-type est réalisé au moyen de la fonction prédéfinie PTYPE. Il est donc possible de donner des types de formats par défaut à n'importe quelle fonction.

8.4 L'Extension du Paragrapheur

Il est également possible de définir des formats d'édition spéciaux pour les types étendus Le_Lisp. Ces formats étendus sont des fonctions de nom PRETTY dans le package du type étendu.

```
? (DEFSTRUCT foo a b c)
= foo
? (SETQ v (#:FOO:MAKE))
= #:foo:[() () ()]
? (#:FOO:B v '(1 2 3 4 5))
= (1 2 3 4 5)
? (#:FOO:C v 100)
= 100
? v
= #:foo:[() (1 2 3 4 5) 100]
? (DE #:FOO:PRETTY (x)
?     (PRINCH "<")
?     (PPRIN (#:FOO:B v))
?     (PRINCH "/"
?     (PPRIN (#:FOO:C v))
?     (PRINCH "/"
?     (PPRIN (#:FOO:A v))
?     (PRINCH ">"))
= #:FOO:PRETTY
```

```
? (PPRINT (LIST v v v))
(‹(1 2 3 4 5)/100/()›
 ‹(1 2 3 4 5)/100/()›
 ‹(1 2 3 4 5)/100/()›)
= (#:foo:[() (1 2 3 4 5) 100] #:foo:[() (1 2 3 4 5) 100]
#:foo:[() (1 2 3 4 5) 100])
```

CHAPITRE 9

Les Impressions Spécialisées

Francis Dupont

Ce chapitre décrit la fonction d'édition FORMAT, compatible avec Common Lisp, ainsi qu'un ensemble de fonctions permettant de traiter les listes circulaires ou partagées.

9.1 Les Formats d'Edition

FORMAT [Feature]

ce trait indique si la fonction FORMAT est présente en mémoire.

9.1.1 La fonction d'édition FORMAT

(FORMAT <dest> <cntrl> <e1> ... <eN>) [SUBR à 2 ou N arguments]

édite la chaîne de caractères <cntrl> dans le canal de sortie <dest>. Cette chaîne peut contenir des *directives d'édition* formées du caractère "~" suivi d'un caractère spécifiant un format d'édition. Durant l'édition de la chaîne <cntrl> ces directives provoqueront l'édition des arguments <e1>...<eN>, qui sont des objets Lisp quelconques. La première directive de la chaîne <cntrl> décrit le format d'édition de l'argument <e1>, la seconde celui de l'argument <e2>, etc.
L'argument <dest> permet de spécifier le canal de sortie utilisé pour l'édition. Il peut prendre différentes valeurs :

- un numéro de canal d'entrée/sortie ouvert par l'une des fonctions OPENO ou OPENA,
- le symbole T, l'édition est alors réalisée sur le canal terminal (),
- le symbole (), l'édition est alors réalisée dans une chaîne de caractères qui est rendue en résultat par la fonction FORMAT. Dans ce dernier cas aucune impression n'est réalisée.

(PRINF <cntrl> <e1> ... <eN>) [SUBR à 1 ou N arguments]

est identique à la fonction précédente mais les sorties ont lieu dans le canal de sortie courant.

La directive la plus couramment utilisée est la directive ~A qui permet d'éditer un argument de la même manière que la fonction PRIN.

```
ex : ; édition au format ~A
     ? (DE AIME (x y)
     ?     (PRINF T "~A aime ~A." x y)
     ?     (TERPRI))
```

```
= AIME
? (AIME 'romeo 'juliette)
romeo aime juliette.
= t
```

Certains formats d'édition acceptent des modificateurs et des paramètres numériques. Un modificateur est l'un des deux caractères ":" ou "@", apparaissant juste avant le caractère spécifiant le format d'édition. Il permet de modifier l'effet obtenu lors de l'édition de l'argument correspondant à la directive dont il fait partie. L'effet de chaque modificateur dépend bien sûr du format d'édition auquel il est appliqué.

```
ex : ; directive ~C (caractère)
     ? (FORMAT () "le caractere ~C" #/a)
     = le caractere a
     ; directive ~C, modifiée ":" (nom de caractère)
     ? (FORMAT () "le caractere de nom ~:C" #^A)
     = le caractere de nom control-A
     ; directive ~C, modifiée "@" (caractère lisible par une #-macro)
     ? (FORMAT () "(setq char ~@C)" #^A)
     = (setq char #^A)
```

Les paramètres numériques apparaissent entre le caractère "~" et les modificateurs éventuels. Ils permettent de préciser certaines caractéristiques du format auquel ils s'appliquent : nombre de colonnes utilisées, caractère de remplissage, positions des taquets de tabulation, base d'écriture, etc.
Lorsque plusieurs paramètres numériques sont requis, il doivent être séparés par des virgules. Certains paramètres sont optionnels et prennent des valeurs par défaut s'ils ne sont pas fournis.

Chaque paramètre numérique peut être décrit de plusieurs manières :
- un nombre entier, lu dans la base de lecture courante,
- un caractère quoté, le paramètre est alors le code interne de ce caractère,
- le caractère v ou V, le paramètre est alors l'argument <eI> correspondant à la directive en cours de traitement. L'argument édité sera alors l'argument suivant, <eI+1>.
- le caractère #, le paramètre est alors le nombre d'arguments <eI>, non encore traités par la fonction FORMAT.

```
ex: ; la directive ~D (nombre en base 10) accepte deux paramètres
    ; optionnels : nombre de colonnes utilisées et caractère de remplissage
    ;
    ; Directive ~D sans paramètre
    ? (FORMAT () "le nombre ~D" 314)
    = le nombre 314
    ; Directive ~D, nombre de colonnes utilisées 12
    ? (FORMAT () "le nombre ~12D" 314)
    = le nombre          314
    ; Directive ~D, nombre de colonnes 12, remplissage par le caractère 0
    ? (FORMAT () "le nombre ~12,'0D" 314)
    = le nombre 000000000314
    ; Directive ~D, nombre de colonnes donné dans la liste d'arguments
    ? (FORMAT () "le nombre ~vD" 10 314)
    = le nombre        314
    ; Directive ~D, nombre de colonnes et caractère de remplissage
```

```
; donnés dans la liste d'arguments
? (FORMAT () "le nombre ~v,vD" 10 #/. 314)
= le nombre .......314
```

9.1.2 Les différents formats d'édition

~A : ASCII [Format d'édition]

Syntaxes :
~A
~<mincol>,<padchar>A
~<mincol>,<colinc>,<minpad>,<padchar>A
Modificateurs : "@".

Edite un argument de la même manière que la fontion PRIN.

Avec 1 ou 2 paramètres, <mincol> est la largeur minimale, le résultat est complété à gauche (à droite si @ figure) par le caractère <padchar> ou #\SP par défaut. Avec 3 ou 4 paramètres (valeurs par défaut 0, 1, 0 et #\SP), on complète le résultat par <minpad> caractères <padchar>, puis on ajuste à une largeur au moins égale à <mincol> par des groupes de <colinc> caractères.

```
ex : (FORMAT () "~A" 'abcd)                    ->   "abcd"
     (FORMAT () "~6A" 'abcd)                   ->   "  abcd"
     (FORMAT () "~6@A" 'abcd)                  ->   "abcd  "
     (FORMAT () "~6,'*A" 'abcd)                ->   "**abcd"
     (FORMAT () "~1,,2A" 'abcd)                ->   "  abcd"
     (FORMAT () "~8,,2A" 'abcd)                ->   "    abcd"
     (FORMAT () "~8,10,2,v@A" #/u 'abcd)       ->   "abcduuuuuuuuuu"
```

~S : S-EXPRESSION [Format d'édition]

Cette directive est équivalente à la directive précédente, mais positionne l'indicateur #:SYSTEM:PRINT-FOR-READ. Le résulat de l'impression peut donc être relu par le lecteur Lisp. Cette directive accepte les même paramètres et les mêmes modificateurs que la directive ~A.

```
ex : (FORMAT () "~S ~A" "abcd" "efgh")         ->   ""abcd" efgh"
```

~R : BASE (RADIX) [Format d'édition]

Syntaxes :
~<base>R
~<base>,<mincol>,<padchar>R
Modificateurs : "@".

Si l'argument correspondant est un nombre entier ou rationnel il est édité en utilisant la base de sortie <base> (voir la fonction OBASE). Si l'argument n'est pas un nombre le format ~A est employé. Les paramètres <mincol>, <padchar> et le modificateur "@", ont le même rôle que pour la directive ~A.

```
ex : (FORMAT () "~2R" 15)                      ->   "1111"
     (FORMAT () "~3,5,'*@R" 13)                ->   "111**"
```

~D : DECIMAL [Format d'édition]

La directive ~D est équivalente à la directive ~10R qui spécifie une édition en base 10. ~D accepte les mêmes paramètres et modificateurs que la directive ~R.

```
ex : (FORMAT () "~D" 5)                  ->   "5"
     (FORMAT () "~3,'0D" 5)              ->   "005"
     (FORMAT () "~3,'*@D" 5)             ->   "5**"
     (FORMAT () "~7D" 'abcd)             ->   "   abcd"
```

~B : BINAIRE [Format d'édition]

La directive ~B est équivalente à la directive ~2R qui spécifie une édition en base 2. ~B accepte les mêmes paramètres et modificateurs que la directive ~R.

```
ex : (FORMAT () "~5,vB" #/0 15)          ->   "01111"
```

~O : OCTAL [Format d'édition]

La directive ~O est équivalente à la directive ~8R qui spécifie une édition en base 8. ~O accepte les mêmes paramètres et modificateurs que la directive ~R.

```
ex : (FORMAT () "~#,'0O" 63 7 8)         ->   "077"
```

~X : HEXADECIMAL [Format d'édition]

La directive ~X est équivalente à la directive ~16R qui spécifie une édition en base 16. ~X accepte les mêmes paramètres et modificateurs que la directive ~R.

```
ex : (FORMAT () "~5@X" 17)               ->   "11   "
```

~P : PLURIEL ANGLAIS [Format d'édition]

Syntaxe :

~P

Modificateurs : ":" et "@", combinables.

Imprime le caractère "s" si l'argument correspondant est différent du nombre entier 1, rien sinon.
Le modificateur "@" permet d'imprimer les caractères "ies" si l'argument est différent de 1, le caractère "y" sinon. Ceci est particulièrement utile pour les pluriels anglo-saxons.
Si le modificateur ":" est présent l'argument testé est l'argument précédemment traité par la fonction format. Ceci permet d'imprimer un argument et la marque du pluriel lui correspondant sans dupliquer l'argument dans la liste de paramètres.

```
ex : (FORMAT () "~P" 1)                        ->   ""
     (FORMAT () "~P" 'abcd)                    ->   "s"
     (FORMAT () "~D tr~:@P/~D win~:P" 7 1)     ->   "7 tries/1 win"
     (FORMAT () "~D tr~:@P/~D win~:P" 1 0)     ->   "1 try/0 wins"
     (FORMAT () "~D tr~:@P/~D win~:P" 1 3)     ->   "1 try/3 wins"
```

~C : CARACTERE [Format d'édition]

Syntaxe :
~C
Modificateurs : ":" ou "@", exclusifs.

Imprime le caractère dont le code interne est donné en argument.
Le modificateur "@" permet d'imprimer le caractère de façon à pouvoir le relire avec l'une des #-macros #/, #^ ou #\.
Le modificateur ":" permet d'imprimer le nom du caractère.

```
ex : (FORMAT () "~C" #/+)              -> "+"
     (FORMAT () "~@C" #/+)             -> "#/+"
     (FORMAT () "~@C" #\SP)            -> "#\SP"
     (FORMAT () "~@C" #^A)             -> "#^A"
     (FORMAT () "~:C" #/+)             -> "+"
     (FORMAT () "~:C" #^A)             -> "control-A"
     (FORMAT () "~:C" #^I)             -> "tab"
```

~E : NOMBRE FLOTTANT [Format d'édition]

Syntaxes :
~E
~[<wide>],[<def>],[<exp>],[<k-factor>],
[<overflowchar>],[<padchar>],[<exponentchar>]E
Modificateurs : "@".

Format flottant exponentiel : imprime l'argument qui doit être un nombre. <wide> spécifie la largeur du résultat, le remplissage éventuel étant à gauche, <def> le nombre de chiffres significatifs, <exp> le nombre de chiffres de l'exposant, <exponantchar> (par défaut e) est le caractère de début d'exposant (l'exposant est imprimé avec un signe), <k-factor> le nombre de chiffres avant le point décimal (par défaut 1). Si le résultat déborde des <wide> caractères, alors <wide> <overflowchar> remplacent le nombre. Le caractère @ précise s'il faut imprimer un + si l'argument n'est pas négatif. Si l'argument n'est pas un flottant, ~<wide>D est utilisé. Si <wide>, <def> et <exp> sont omis, on prend le format libre (comme PRIN).

```
ex : (FORMAT () "~9,2E" 3.14159)              ->   "  3.14e+0"
     (FORMAT () "~13,6,2,VE" -7 3.14159)      ->   ".00000003e+08"
     (FORMAT () "~13,6,2,VE" -6 3.14159)      ->   "0.0000003e+07"
     (FORMAT () "~13,6,2,VE" -5 3.14159)      ->   " 0.000003e+06"
     (FORMAT () "~13,6,2,VE" -4 3.14159)      ->   " 0.000032e+05"
     (FORMAT () "~13,6,2,VE" -3 3.14159)      ->   " 0.000314e+04"
     (FORMAT () "~13,6,2,VE" -2 3.14159)      ->   " 0.003142e+03"
     (FORMAT () "~13,6,2,VE" -1 3.14159)      ->   " 0.031416e+02"
     (FORMAT () "~13,6,2,VE" 0 3.14159)       ->   " 0.314159e+01"
     (FORMAT () "~9,2,1,,'*E" 3.14159)        ->   "  3.14e+0"
     (FORMAT () "~9,2,1,,'*E" -3.14159)       ->   " -3.14e+0"
     (FORMAT () "~9,2,1,,'*E" 1100.)          ->   "  1.10e+3"
     (FORMAT () "~9,2,1,,'*E" 1.1e+13)        ->   "*********"
     (FORMAT () "~9,3,2,-2,'%@E" 3.14159)     ->   "+.003e+03"
     (FORMAT () "~9,3,2,-2,'%@E" -3.14159)    ->   "-.003e+03"
     (FORMAT () "~10,3,2,2,'?,,'$E" 3.14159)       ->   " 31.42$-01"
     (FORMAT () "~10,3,2,2,'?,,'$E" -3.14159)      ->   "-31.42$-01"
     (FORMAT () "~10,3,2,2,'?,,'$E" 1100.)         ->   " 11.00$+02"
```

```
(FORMAT () "~10,3,2,2,'?,,'$E" 1.1e+13)      ->   " 11.00$+12"
```

~F : NOMBRE FLOTTANT [Format d'édition]

Syntaxes :

~F

~[<wide>],[<def>],[<k-factor>],[<overflowchar>],[<padchar>]F

Modificateurs : "@".

Format flottant fixe : imprime l'argument qui doit être un nombre. <wide> spécifie la largeur du résultat, le remplissage éventuel par <padchar> se faisant à gauche (par défaut <padchar> est le caractère #\SP). <def> donne le nombre de chiffres après le point décimal. <k-factor> est un facteur d'échelle, c'est 10 puissance <k-factor> fois l'argument qui est édité (par défaut <k-factor> = 0). Si le résultat déborde des <wide> caractères, alors <wide> <overflowchar> remplacent le nombre. Le caractère @ précise s'il faut imprimer un + si l'argument n'est pas négatif.

```
ex : (FORMAT () "~6F" 3.14159)          ->   "3.1416"
     (FORMAT () "~6F" -3.14159)         ->   "-3.142"
     (FORMAT () "~6F" 100)              ->   "100.00"
     (FORMAT () "~6F" 1234.0)           ->   "1324.0"
     (FORMAT () "~6F" 0.006)            ->   "0.0060"
     (FORMAT () "~,2F" 3.14159)         ->   "3.14"
     (FORMAT () "~6,2F" 3.14159)        ->   "  3.14"
     (FORMAT () "~6,2@F" 3.14159)       ->   " +3.14"
     (FORMAT () "~6,2,1,'*F" 3.14159)   ->   " 31.42"
     (FORMAT () "~6,2,1,'*F" 100)       ->   "******"
     (FORMAT () "~6,2,1,'*F" 0.006)     ->   "  0.06"
```

~G : NOMBRE FLOTTANT [Format d'édition]

Syntaxes :

~G

~[<wide>],[<def>],[<exp>],[<k-factor>],
[<overflowchar>],[<padchar>],[<exponentchar>]G

Modificateurs : "@".

Format flottant général : imprime l'argument qui doit être un nombre.

Si l'impression déborde en format fixe F, alors on emploie le format E avec tous les arguments. Sinon on édite le nombre avec <exp> + 2 (ou 4 si <exp> n'est pas précisé) blancs à droite en format F sur la largeur restante avec <def> chiffres significatifs.

~~ : TILDE [Format d'édition]

Syntaxes :

~~

~<nb>~

Aucun modificateur.

Imprime 1 ou <nb> caractères ~.

```
ex :
     (FORMAT () "~~")                   ->   "~"
     (FORMAT () "~v~" 3)                ->   "~~~"
```

~% : FIN DE LIGNE [Format d'édition]

Syntaxes :
~%
~<nb>%
Aucun modificateur.

Imprime 1 ou <nb> caractères #\LF. Cette directive n'existe que par compatibilité avec Common Lisp. Il est préférable d'appeler la fonction TERPRI.

```
ex : (PNAME (FORMAT () "~% a"))          ->   (10 32 97)
     (PNAME (FORMAT () "~2% a"))         ->   (10 10 32 97)
```

~#\LF : FIN DE LIGNE [Format d'édition]

Syntaxe :
~<caractère #\LF>
Modificateurs : ":" ou "@", exclusifs.

Cette directive est constituée du caractère ~ suivi immédiatement d'une marque de fin de ligne.

Sans modificateur, saute le caractère fin-de-ligne ainsi que tous les caractères d'espace (type CSEP) suivants. Avec @, imprime fin-de-ligne, mais saute les caractères d'espace suivants. Avec :, n'imprime pas fin-de-ligne et ne saute pas d'autre caractère. Cette directive n'existe que par compatibilité avec Common Lisp.

```
ex : (FORMAT () (STRING '(#/~ #\LF #\SP #\TAB #/b)))   ->   "b"
     (PNAME (FORMAT () (STRING '(#/~ #/: #\LF #\SP #\TAB #/b))))
                                        ->   (32 9 98)
     (PNAME (FORMAT () (STRING '(#/~ #/@ #\LF #\SP #\TAB #/b))))
```

~T : TABULATION [Format d'édition]

Syntaxes :
~T
~<colnum>,<colinc>T
Modificateurs : "@".

Imprime des espaces jusqu'à une tabulation. La valeur par défaut est de 1 pour les paramètres.
Sans @, la tabulation est absolue : on imprime des espaces jusqu'à la colonne <colnum> (les colonnes commencent à 0, on laisse le curseur dans la colonne suivante). Si on est dans <colnum> ou si on l'a déjà dépassée, on va dans la colonne <colnum> + k * <colinc> avec k minimal.
Avec @, la tabulation est relative : on imprime <colnum> espaces, puis on va dans la colonne k * <colinc> avec k minimal.

```
ex : (FORMAT () "~T")                    ->   " "
     (FORMAT () "~8T")                   ->   "       "
     (FORMAT () "ab~8,2T")               ->   "ab      "
     (FORMAT () "0123456789~8T")         ->   "0123456789"
     (FORMAT () "0123456789~8,5T")       ->   "0123456789    "
     (FORMAT () "ab~2,5@T")              ->   "ab    "
```

~* : IGNORE [Format d'édition]

Syntaxes :
~*
~<nb>*
Modificateurs : ":" ou "@", exclusifs

Ignore l'argument correspondant. La valeur par défaut de <nb> est 1, sauf s'il y a un @, la valeur par défaut étant alors 0. Sans modificateur, on saute les <nb> arguments suivants. Avec :, on remonte de <nb> arguments. Avec @, on revient à la liste d'arguments initiale à partir <nb>-ième élément en comptant à partir de 0. Dans une indirection ou une itération, on travaille sur les sous-listes.

```
ex : (FORMAT () "~2*A" 1 2 3 4)          ->   "3"
     (FORMAT () "~D~ ~:* ~D" 1 2)        ->   "1  1"
     (FORMAT () "~D~2* ~:2*~D" 12 3 4) ->   "12 3"
     (FORMAT () "~D ~D ~@*~D" 1 2 3)     ->   "1 2 1"
```

~? : INDIRECT [Format d'édition]

Syntaxe :
~?
Modificateurs : "@"

Indirect : on prend comme format l'argument suivant, qui doit être une chaîne. S'il n'y a pas d'@, on prend comme argument, l'argument suivant la chaîne, qui doit être une liste. Sinon on utilise les arguments suivants.

```
ex : (FORMAT () "~? ~D" "<~A ~D>" '("foo" 5) 7)   ->   "<foo 5> 7"
     (FORMAT () "~@? ~D" "<~A ~D>" "foo" 5 7)     ->   "<foo 5> 7"
```

~[;] : CONDITIONNELLE [Format d'édition]

Syntaxes :
~[<nb>][<strg0>~;<strg1> ... ~;<strgN>~]
~[<nb>][<strg0>~;<strg1> ... ~:;<defaut>~]
~:[<false>~;<true>~]
~@[<true>~]

Conditionnelle : le format <strgI> est utilisé.

On choisit le format <strgI> avec I égal à <nb> ou au premier argument s'il n'y a pas de <nb> (le compte commence à 0). S'il n'y a pas assez de <strg>, alors <defaut> est choisie. S'il n'y a pas de <defaut>, rien n'est produit. Le modificateur : sert pour un argument booléen. Si le modificateur @ est présent, alors : ou bien l'argument est (), et rien n'est produit, et l'argument suivant servira pour le format suivant; ou bien l'argument n'est pas (), il est alors imprimé selon le format <true>. Un ~^ dans une conditionnelle renvoie au ~? ou ~{ englobant s'il existe.

```
ex : (FORMAT () "~[Siamese~;Manx~;Persian~] Cat" 0)     ->   "Siamese Cat"
     (FORMAT () "~[Siamese~;Manx~;Persian~] Cat" 2)     ->   "Persian Cat"
     (FORMAT () "~[Siamese~;Manx~;Persian~] Cat" 8)     ->   " Cat"
     (FORMAT () "~[Siamese~;Manx~:;Persian~] Cat" 8)    ->   "Persian Cat"
     (FORMAT () "~:[true~;false~]" t)                   ->   "false"
     (FORMAT () "~:[true~;false~]" ())                  ->   "true"
     (FORMAT () "~@[ print level = ~D~]~@[ print length = ~D~]" () 5)
```

```
                                        ->   " print length = 5"
```

~{} : ITERATION [Format d'édition]

Syntaxes :
~[<nb>]{<strg>~[:]}
Modificateurs : ":" et "@", combinables. z Itération : on applique <strg> sur les arguments tant qu'il y en a.

Si <nb> figure, on boucle <nb> fois au plus sur <strg>. S'il y a un ~:}, on boucle au moins une fois, sauf si <nb> est 0. Si <strg> est vide, on prend l'argument suivant, qui doit alors être une chaîne. Sans modificateur, on exécute l'itération avec l'argument suivant, qui doit alors être une liste. Avec @, on prend les arguments suivants. Si : est présent, chaque itération se fait sur une sous-liste différente à chaque fois. Remarque : les ~{ et ~[ne peuvent être imbriqués que s'il n'y a pas 2 formats du même type.

```
ex : (FORMAT () "The winners are:~{ ~S~}." (fred harry))
                                        ->   "The winners are: fred harry."
     (FORMAT () "Pairs:~{ <~S,~S>~}." (a 1 b 2)) -> "Pairs: <a,1> <b,2>."
     (FORMAT () "Pairs:~:{ <~S,~S>~}." ((a 1))    -> "Pairs: <a,1>."
     (FORMAT () "Pairs:~@{ <~S,~S>~}." a 1 b 2)   -> "Pairs: <a,1> <b,2>."
     (FORMAT () "Pairs:~:@{ <~S,~S>~}." (a 1) (b 2) (c 3) (d 4)
                                        ->   "Pairs: <a,1> <b,2> <c,3> <d,4>."
     (FORMAT () "~2~{~D ~}" (1 2 3 4)) ->    "1 2 "
```

~^ : ECHAPPEMENT [Format d'édition]

Syntaxes : ~[<n1>],[<n2>],[<n3>][:]^

Echappement : on sort de l'indirection ou de l'itération ou s'il n'y en a pas du format courant.

S'il n'y a pas de paramètre, on prend le nombre d'arguments restants comme unique paramètre. S'il n'y a qu'un paramètre, la sortie se fait si il est nul. Si il y a deux paramètres, quand ils sont égaux, s'il y en a 3, quand <n1> <= <n2> ou <n2> <= <n3>. Dans une indirection avec un :, l'échappement se fait au niveau de la sous-liste, sauf si le modificateur : figure.

```
ex : (FORMAT () "Done.~^ ~D warning~:P.~^ ~D error~:P.")     ->   "Done."
     (FORMAT () "Done.~^ ~D warning~:P.~^ ~D error~:P." 3)
                                        ->   "Done. 3 warnings."
     (FORMAT () "Done.~^ ~D warning~:P.~^ ~D error~:P." 1 5)
                                        ->   "Done. 1 warning. 5 errors."
```

9.2 Le Traitement des Objets Circulaires ou Partagés

Les fonctions de limitation d'impression du chapitre 6, pour autant qu'elles stoppent l'impression et permettent d'en avoir une vue abrégée, ne permettent pas, en cas de structures circulaires ou partagées de connaître les cellules de liste, ou les vecteurs effectivement partagés.

Des fonctions spécialisées dans l'impression et la lecture des objets circulaires (listes, vecteurs, symboles circulaires sur les packages) sont fournies dans la bibliothèque standard.

On les charge par la commande :

```
? (LIBLOAD LIBCIR)
```

LIBCIR [Feature]

ce trait indique que la librairie de traitement des objets circulaires est présente en mémoire.

#<n>= [#-Macro]

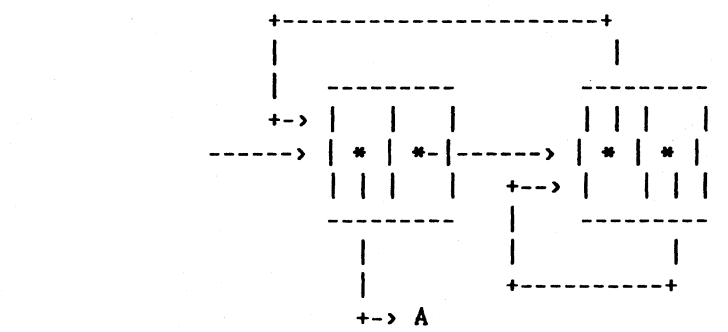
numérote un objet partagé ou circulaire qui pourra être référencé par #<n>#.

#<n># [#-Macro]

référence le <n>ième objet partagé ou circulaire (numéroté par #<n>=).

```
#1=(A . #2=(#1# . #2#))
```

a la structure suivante :



(CIRPRIN <e>) [SUBR à 1 argument]
(CIRPRINFLUSH <e>) [SUBR à 1 argument]
(CIRPRINTs <e>) [SUBR à 1 argument]

imprime leurs arguments en employant les #-macros #<N>= et #<N># pour les éventuelles constructions circulaires ou partagées si elles ne peuvent pas être imprimées normalement.

```
ex : (PROGN (SETQ 1 '(1)) (RPLACA 1 1) 1)     -> (((((((&)))))))
     (CIRPRINT 1)                             ->   #1=(#1#)
     (SETQ 1 (CIRLIST 1))                     -> (1 1 1 1 1 1 1 1 ...
     (CIRPRINT 1)                             ->   #1=(1 . #1#)
     (CIRPRINT '(a a))                        ->   (a a)
     (PROGN (SETQ v #[0]) (VSET v 0 v) v)     -> #[#[#[#[#[#[#[&]]]]]]]
```

```
(CIRPRINT v)                                    ->   #1=#[#1# ]
(PROGN (SETQ x '(p q) y (LIST x 'foo x))
       (RPLACD (LAST y) (CDR y)) y)
                                        ->   ((p q) foo (p q) foo (p q) ...
(CIRPRINT y)
                          ->   (#2=(p q) . #1=(foo #2# . #1#))
```

(CIREQUAL <e1> <e2>) [SUBR à 2 arguments]
(CIRNEQUAL <e1> <e2>) [SUBR à 2 arguments]

comparent leurs deux arguments même s'ils comprennent des constructions circulaires ou partagées. L'égalité doit être comprise comme l'isomorphisme des structures.

```
ex : (SETQ x '(a) y (LIST x x) z (LIST '(a) '(a)))    ->  ((a) (a))
     (CIRPRINT y)                                     ->  (#1=(a) . #1#)
     (EQUAL y z)                                      ->  t
     (CIREQUAL x y)                                   ->  ()
```

(CIRCOPY <e>) [SUBR à 1 argument]

copie son argument en reproduisant les structures circulaires ou partagées.

```
ex : (PROGN (SETQ x (CIRCOPY y)) (CIRPRINT x))    ->   (#1=(a) . #1#)
     (CIREQUAL x y)                               ->   t
```

#:LIBCIR:PACKAGE-PARANO [Variable]

s'il est égal à T, indique que les packages des symboles peuvent contenir des structures bouclées (l'usage de tels symboles est formellement interdit). Les fonctions sur les objets circulaires ou partagés sont fortement ralenties, mais on peut enfin débusquer des erreurs redoutables!

```
ex : (PROGN (SETQ x 'foo) (PACKAGECELL x x) x)    ->   ...
     (CIRPRINT x)                                 ->   #1=#:#1#:foo
```

CHAPITRE 10

L'Arithmétique Rationnelle et Complexe

Jean Vuillemin

Pour charger les rationnels Q, et (ou) les complexes C, faire respectivement :

```
(LIBLOAD RATIO)
(LIBLOAD COMPLEX)  ; ne suppose pas nécessairement le chargement de Q.
```

Ces bibliothèques étendent l'arithmétique générique Le_Lisp aux rationnels en précision arbitraire et aux nombres complexes.

RATIO [Feature]
COMPLEX [Feature]

ces traits indiquent si la bibliothèque arithmétique rationnelle ou complexe est présente en mémoire.

10.1 Les Rationnels Q

10.1.1 Lecture et écriture de Q

Les rationnels s'écrivent sous la forme Z/N ou Z, le numérateur, est un entier relatif, et N, le dénominateur, un entier naturel positif ou nul.

```
ex : 12345678910    -> 12345678910
     00000123       -> 123
     -1234567       -> -1234567
     2/3            -> 2/3
     4/6            -> 2/3
     5/1            -> 5
     -10/2          -> -5
     1/0            -> 1/0
     -2/0           -> -1/0
```

Le système écrit les rationnels en forme réduite : les zéros non significatifs sont éliminés, numérateur et dénominateur sont divisés par leur pgcd. Les éléments de Q dont le dénominateur vaut 1 sont réduits en éléments de Z, les entiers relatifs.

L'évaluation d'un nombre de Q est ce nombre lui-même, nul besoin de les *quoter* donc.

Signalons la présence dans Q de l'infini 1/0, moins l'infini -1/0, ainsi que de l'indéfini 0/0. Toute opération arithmétique avec 0/0 comme opérande retourne 0/0. On peut considérer 0/0 comme une valeur d'erreur. L'indéfini 0/0 représente de fait l'intervalle [-1/0,1/0] comprennant tous les rationnels, y compris plus et moins l'infini.

La présence de ces éléments fait que Q n'est pas un corps.

10.1.1.1 Ecriture décimale approchée de Q

En plus de l'impression standard Le_Lisp, qui écrit un rationnel sous forme exacte réduite, on peut obtenir une écriture décimale approchée, dont la précision est réglable.

La fonction de réglage de l'affichage est :

(PRECISION <n>) [SUBR à 0 ou 1 argument]

Sans argument, retourne la précision d'affichage courante. Avec un argument, elle règle la précision d'affichage.
Si l'argument est T (ou une constante non-numérique), l'affichage est exact.
Si l'argument est un flottant, sa partie entière indique le nombre de chiffres à écrire après la virgule dans la base courante (OBASE). Quand le dénominateur est petit, l'écriture décimale (qui est toujours périodique) est exacte, la partie périodique étant signalée entre { et }. Sinon, le nombre de chiffres demandé après la virgule est écrit, suivi de ... .
Si l'argument est un entier de type FIXP positif, il indique le nombre de termes affichés dans l'écriture en fraction continue normale.

Les rationnels écrits sous un autre mode que (précision T) ne sont pas relisibles par le lecteur.

```
(PRECISION 10.)      ->    10.
22/7                 ->    3.{142857}          ; écriture exacte
333/106              ->    3.1415094339...     ; 10 chiffres décimaux
103993/33102         ->    3.1415926530...
(PRECISION 10)       ->    10
22/7                 ->    /3 7/               ; 22/7 = (+ 3 (/ 7))
333/106              ->    /3 7 15/
103993/33102         ->    /3 7 15 1 292/
```

10.1.2 Tests de type

On peut tester l'appartenance à Z = {...,-1,0,1,2,...} ou Q grâce aux prédicats :

(RATIONALP <q>) [SUBR à 1 argument]

Retourne <q> si <q> est un élément de Q, () sinon.

```
ex : (RATIONALP 234567)      ->  234567
     (RATIONALP -4/6)        ->  -2/3
     (RATIONALP 2.34567)     ->  ()   ; un float n'est pas un rationnel.
```

(INTEGERP <z>) SUBR à 1 argument

Retourne <z> si <z> est un élément de Z, () sinon.

```
ex : (INTEGERP -4/2)         ->  -2
     (INTEGERP 2)            ->  2
     (INTEGERP 4/3)          ->  ()
     (INTEGERP 2.)           ->  ()   ; un flottant n'est pas un Z.
```

Les deux fonctions INTEGERP et RATIONNALP sont génériques. Il suffit, pour les étendre au type <gog>, de définir <gog>:INTEGERP et <gog>:RATIONALP.

10.1.3 Arithmétique générique rationnelle

Toutes les fonctions arithmétiques génériques de Le_Lisp (voir le chapitre 4) fonctionnent sur Q, en essayant, quand c'est possible, de faire des calculs exacts. Leur syntaxe est la même que sur les petits entiers Le_Lisp.

```
(+ 3 (/ (+ 7 (/ 15))))          ->      333/106
(QUOTIENT 333/106 1)             ->      3          ; modulo dans #:EX:MOD
(MODULO 333/106 1)               ->      15/106     ; c'est lui!
(* 22/7 7/11 -1/3)               ->      -2/3
(< 1/3 2 7/2)                    ->      1/3
(* 1/0 0)                        ->      0/0        ; q*0=0 est faux sur Q.
(- 1/0)                          ->      -1/0
(+ 1/0 -1/0)                     ->      0/0
```

La division entière (QUOTIENT n d), pour n et d dans Q, renvoie un quotient q dans Z, et un reste r (par la variable #:EX:MOD) dans Q tels que :

```
n = d*q + r, avec 0 <= r < abs(d).
```

10.1.4 Fonctions propres à Z

(GCD <z0> ... <zn>) [SUBR à N arguments]

Calcule le plus grand commun diviseur de <z0> ... <zn>, entiers relatifs.

(PGCD <z0> <z1>) [SUBR à 2 arguments]

La même avec 2 arguments, et sans vérification de type.

(EVEN? <z>) [SUBR à 1 argument]

Renvoie <z> si l'entier relatif <z> est pair, () sinon.

(FACT <n>) [SUBR à 1 argument]

Calcule la factorielle 1*2* ... *<n> = <n>!, pour <n> entier naturel.

(FIB <n>) [SUBR à 1 argument]

Calcule le <n>-ième nombre de Fibonnacci. F0=0, F1=1, Fi+2 = Fi + Fi+1.

```
ex : (GCD 864164 11578 -168 252) ->      14
     (FACT 30)      ->      265252859812191058636308480000000
     (FIB 200)      ->      280571172992510140037611932413038677189525
```

10.1.5 Fonctions propres à Q

Les fonctions NUMERATOR et DENOMINATOR accèdent respectivement au numérateur et dénominateur des rationnels, sans réduction de diviseurs communs. Les calculs sur Q sont faits sans réduction des diviseurs communs. Cette réduction se fait explicitement lors d'un PRIN, ou d'un test INTEGERP.

(NUMERATOR <f>) [SUBR à 1 argument]
(DENOMINATOR <f>) [SUBR à 1 argument]

```
ex : (NUMERATOR 6/10)           ->    6      ; pas de réduction
     (DENOMINATOR 6/10)         ->    10
     (SETQ a (PRIN 6/10))       ->    3/5    ; réduction par le print
     (NUMERATOR a)              ->    3      ; a est réduit.
     (NUMERATOR 12)             ->    12
     (DENOMINATOR (fib 100)))   ->    1
```

La fonction puissance (**) tente de faire une arithmétique exacte plus souvent que la fonction POWER.

(<n> <m>) [SUBR à 2 arguments]**

Calcule le produit n*n* ..*n (m fois), pour m entier naturel.

```
ex : (** 2 128)     ->  340282366920938463463374607431768211456
     (** 10 10)     ->  10000000000
     (** 2/3 10)    ->  1024/59049
```

10.1.6 Deux exemples

La fonction (ZETA n e) calcule la somme des 1/i**e pour i de 1 à n.

```
(DE ZETA (n e)
    (LET ((z 0))
         (FOR (i 1 1 n) (SETQ z (+ z (/ (** i e)))))
         z))
```

Pour e=1, on obtient les nombres harmoniques, série divergente.
Pour e=2, la série converge lentement vers le carré de pi divisé par 6.
Pour e=3, la série converge vers zeta(3) dont Apéry a démontré la transcendance.

Calculons enfin (Qe n) par la somme des 1/i!, pour i de 1 à n. Cette expression converge rapidement vers le nombre e.

```
(DE Qe (n)
    (LET ((e 0))
         (FOR (i 0 1 n) (SETQ e (+ e (/ (fact i)))))
         e))
```

```
ex : (ZETA 10 1)          ->    7381/2520
     (ZETA 10 2)          ->    1968329/1270080
     (ZETA 10 3)          ->    19164113947/16003008000
     (Qe 10)              ->    9864101/3628800
     (PRECISION 20.)      ->    20.
     (ZETA 10 1)          ->    2.92896825396825396825...
     (ZETA 10 2)          ->    1.54976773116654069035...
     (ZETA 10 3)          ->    1.19753198567419325166...
```

```
(Qe 10)              ->     2.71828180114638447971...
```

10.2 Les Complexes C

Les noms des fonctions choisies dans cette extension aux complexes de l'arithmétique Le_Lisp sont ceux de Common Lisp.

Après chargement de C, les symboles i et pi valent respectivement [i] et 3.141593... . Il n'est pas conseillé de changer leur valeur.

10.2.1 Lecture et écriture de C

[[Macro caractère]

Les complexes s'écrivent entre [et], sous la forme [xi+y] où x est la partie imaginaire et y la partie réelle. On autorise les simplifications d'écriture : [xi] pour y=0, [xi-y] pour y<0, [i+y] pour x=1 et [-i+y] pour x=-1. Ainsi, l'imaginaire pur s'écrit [i], et son carré est -1.

#C [#-Macro]

Par compatibilité avec Common Lisp, les complexes peuvent également être représentés sous la forme #C(y x), où x est la partie imaginaire et y la partie réelle. Pour obtenir qu'ils s'écrivent sous la forme Common Lisp, faire (PRECISION 'cl).

```
ex : (PRECISION 'cl)   ->    cl
     [i]               ->    #C(0 1)
     [2i+1]            ->    #C(1 2)
     [-i+1]            ->    #C(1 -1)
```

10.2.2 Tests de type

(COMPLEXP <c>) [SUBR à 1 argument]

Retourne <c> si <c> est un complexe, () sinon.

(REALP <r>) [SUBR à 1 argument]

Retourne <r> si <r> est un nombre réel non complexe, () sinon. C'est une fonction générique, étendue au type <gog> en définissant <gog>:realp.

```
ex : (REALP 2)            -> 2
     (REALP 2.3)          -> 2.3
     (REALP [2/3])        -> 2/3
     (REALP [i])          -> ()
     (COMPLEXP [-1234])   -> ()
     (COMPLEXP [-i])      -> [-i]
```

10.2.3 Arithmétique générique complexe

Les fonctions +,-,* et / sont étendues aux complexes.

```
ex : (+ [i] 3)              ->  [i+3]
     (* [i] [i])            ->  -1
     (* [i] (+ [i] [i]))    ->  -2
     (* (* 1/3 [i]) [i])    ->  -1/3
     (+ [i] (/ [i]))        ->  0
     (/ [2i+3])             ->  [-2/13i+3/13]
     (<?> [i] 1)            ->  [i-1]
     (<?> [-2i+3] [i+1])    ->  [-i+1]
     (FLOAT [2/3i+5])       ->  [.6666667i+5.]
```

La division entière QUOTIENT n'a pas de sens sur C, et provoque une erreur. Il en va de même pour TRUNCATE et FLOOR.

La fonction FLOAT convertit en flottant parties réelles et imaginaires. Quant à la comparaison (<?> [ai+b] [ci+d]) entre deux complexes, elle retourne [(<?> a c)i+(<?> b d)], qui n'a de véritable sens mathématique que si le résultat est 0, c'est-à-dire [ai+b]=[ci+d].

Le domaine de définition de SQRT et LOG est étendu à R par les formules :
sqrt(-r)=rsqrt(r) log(-r)=pi*i+log(r).

Les fonctions ASIN et ACOS sont définies sur C par :
acos(z)=pi/2-asinz et asin(z)=-i*log(iz+sqrt(1-z*z)).

```
ex : (SQRT -2)      ->   [1.414214i]
     (LOG -2)       ->   [3.141593i+.6931472]
     (LOG -1)       ->   [3.141593i]
     (LOG 0)        ->   -1/0
     (SQRT -1)      ->   [i]
     (ASIN 2)       ->   [-1.316958i+1.570796]
     (ACOS 2)       ->   [1.316958i]
```

10.2.4 Fonctions propres à C

(MAKECOMPLEX <c> <i>) [SUBR à 2 arguments]

Retourne [<i>i+<c>]. Les arguments <c> et <i> doivent être réels.

(REALPART <c>) [SUBR à 1 argument]

Si <c> = [xi+y], retourne y.

(IMAGPART <c>) [SUBR à 1 argument]

Si <c> = [xi+y], retourne x.

(CONJUGATE <c>) [SUBR à 1 argument]

Si <c> = [xi+y], retourne [-xi+y].

```
(CONJUGATE [2/5i+3/5])  -> [-2/5i+3/5]
(REALPART [2i-3])       -> -3
(IMAGPART [2i-3])       -> 2
```

10.2.5 Coordonnées polaires

On note [rc:tc] la représentation polaire de c = [yi+x]. Le choix modulo pi de l'angle tc=artan(y/x) est déterminé par la position de [yi+x] dans les quatre quadrants du plan complexe :

```
y>0 x>0     0<tc<pi/2     y>0 x=0 tc=pi/2      y>0 x<0 pi/2<tc<pi
y<0 x>0 -pi/2<tc<0        y<0 x=0 tc=-pi/2     y<0 x<0  -pi<tc<-pi/2
```

Le module rc est donné par (ABS c), et l'angle tc par :

(PHASE <c>) [SUBR à 1 argument]

Si <c> = [xi+y] = [rc:tc], retourne tc.

(SIGNUM <c>) [SUBR à 1 argument]

Si <c> = [xi+y] = [rc:tc], retourne [1:tc] = c / (abs c).

```
ex : (PHASE [i+1])        -> .7853982
     (PHASE [-i+1])       -> -.7853982
     (PHASE [i-1])        -> 2.356194
     (PHASE [-i-1])       -> -2.356194
     (ABS [i+1])          -> 1.414214
     (SIGNUM (/ pi 2))    -> 1
```

Le logarithme et l'exponentielle complexe sont définis à partir de la représentation polaire par:
log[r:t] = [ti+log(r)] et exp[r:t] = [exp(r)sin(t)i+exp(r)cos(t)] .

(CIS <r>) [SUBR à 1 argument]

Calcule [sin(r)i+cos(r)].

La fonction SQRT est définie par:

```
sqrt(z) = exp(log(z)/2)
```

Les fonctions trigonométriques sont définies par:

```
sin(z)  = (exp[ci]-exp[-ci])/2i
cos(z)  = (exp[ci]+exp[-ci])/2
atan(z) = -i*log([zi+1]/sqrt(1+z*z))
tan(z)  = sin(z)/cos(z)
```

Le résultat des calculs qui suivent dépend de la précision des flottants utilisés.

```
ex : (LOG (EXP [3i+2]))    -> [3.i+2.]
     (EXP (LOG [3i+2]))    -> [3.i+2.]
     (EXP (* i pi))        -> [0.i-1.]
```

```
(SETQ a (SQRT [i]))   -> [.7071068i+.7071068]
(* a a)              -> [i]
(ASIN (SIN i))       -> [i]
(COS (ACOS [i]))     -> [i]
(ATAN [i])           -> 1/0
(ATAN (TAN [i]))     -> [i]
```

10.2.6 Fonctions hyperboliques

(SINH <z>) [SUBR à 1 argument]

Calcule (exp(z)-exp(-z))/2.

(COSH <z>) [SUBR à 1 argument]

Calcule (exp(z)+exp(-z))/2.

(TANH <z>) [SUBR à 1 argument]

Calcule sinh(z)/cosh(z).

(ASINH <z>) [SUBR à 1 argument]

Calcule log(z+sqrt(1+z*z)).

(ACOSH <z>) [SUBR à 1 argument]

Calcule log(z+(z+1)sqrt((z-1)/z+1)).

(ATANH <z>) [SUBR à 1 argument]

Calcule log((1+z)sqrt(1-1/x*x)).

10.3 Une Mini-Extension Complexe de l'Arithmétique Générique.

Cet exemple de construction des complexes sert de documentation aux extensions de l'arithmétique générique. C'est une simplification du fichier complex.ll. Il est incomplet, ne vérifie pas les cas d'erreurs, et n'est pas conforme à la documentation de C qui précède. Nous le présentons dans un souci pédagogique.

On se place dans le package C :

```
(setq #:sys-package:colon 'C)
```

10.3.1 Représentation

Un complexe C est représenté par sa partie réelle (:r c) et sa partie imaginaire (:i c).

```
(defstruct C r i)
```

Teste le type C :

```
(de complexp (c) (eq 'C (type-of c)))
```

Construit a+ib = [a:b] :

```
(de makecomplex (a b)
    (if (= 0 b)
        a
```

```
        (let ((c (:make)))
             (:i c b)
             (:r c a)
             c)))
```

L'imaginaire pur i*i=-1 :

```
(defvar i (makecomplex 0 1))
```

10.3.2 Lecture et écriture

Par convention, a+ib s'écrit [bi+a] :

```
(de :prin (c)
    (prin "[" (:i c) "i+" (:r c) "]"))
```

Pour pouvoir relire les complexes :

```
(dmc  |[| ()
      (let ((j) (r))
           (setq j          ; Lire la partie imaginaire.
                (with ((typecn #/i 'csep)) (read)))
           (readcn)         ; Lecture du i.
           (readcn)         ; Lecture du +.
           (setq r (read))  ; Lire la partie réelle
           (readcn)         ; Lecture du ].
           (makecomplex r j)))
```

10.3.3 Arithmétique complexe

La comparaison n'est bien définie qu'en cas d'égalité :

```
(de :<?> (a b)
    (if (and (complexp b) (= (:i a) (:i b)) (= (:r a) (:r b)))
        0
        (error '<?>
               "Comparaison complexe non définie"
               (list a b))))
```

Addition binaire :

```
(de :+ (c r)
    (if (complexp r)
        (makecomplex (+ (:r c) (:r r)) (+ (:i c) (:i r)))
        (makecomplex (+ r (:r c)) (:i c))))
```

Extension de + aux fix, et aux float :

```
(de #:fix:+ (r c) (+ c r))    ; r+c = c+r
(de #:float:+ (r c) (+ c r))
```

Soustraction binaire :

```
(de :- (c r) (:+ c (0- r)))   ;  c-r = c+(-r)
```

Extension de - aux fix, et aux float :

```
(de #:fix:- (r c) (+ r (0- c)))
(de #:float:- (r c) (+ r (0- c)))
```

Négation unaire :

```
(de :0- (c) (makecomplex (- (:r c)) (- (:i c))))
```

Multiplication binaire :

```
(de :* (c r)
    (if (complexp r)
        (makecomplex (- (* (:r c) (:r r)) (* (:i c) (:i r)))
                     (+ (* (:r c) (:i r)) (* (:i c) (:r r))))
        (makecomplex (* r (:r c)) (* r (:i c)))))
```

Extension de * aux fix, et aux float :

```
(de #:fix:* (r c) (* c r))    ; r*c = c*r
(de #:float:* (r c) (* c r))
```

Division binaire :

```
(de :/ (a b) (:* a (1/ b)))
```

Extension de / aux fix, et aux float :

```
(de #:fix:/ (r c) (+ r (1/ c)))
(de #:float:/ (r c) (+ r (1/ c)))
```

Inverse unaire 1/: 1/a+ib = a/m-ib/m avec m=a*a+b*b :

```
(de :1/ (c)
    (:* (makecomplex (:r c) (- (:i c))) (/ (:module2 c))))
(de #:fix:1/ (r) (/ (float r)))
```

Calcule le module de c au carré :

```
(de :module2 (c)
    (+ (* (:i c) (:i c)) (* (:r c) (:r c))))
```

Calcule le module de c :

```
(de :abs (c) (if (zerop (:r c)) (abs (:i c)) (sqrt (:module2 c))))
```

10.3.4 exp, log et sqrt

exp[ai+b] = exp(b)*[sina*i+cosa] :

```
(de :exp (c)
    (:*  (makecomplex (cos (:i c)) (sin (:i c))) (exp (:r c))))
```

Le nombre pi :

```
(defvar pi (* 4 (atan 1)))
```

log(r*e**i*theta) = log(r)+i*theta

```
; La définition qui suit est mathématiquement fausse pour simplifier!
(de :log (c)
    (makecomplex (log (:abs c))
                 (if (= 0 (:r c))
                     (/ pi 2)
                     (atan (/ (:i c) (:r c))))))
```

Pour étendre log aux fix négatifs :

```
(de #:fix:log (r)
    (selectq (<?> r 0)
      (1 (log (float r)))
      (0 (- (/ 0)))
      (-1 (makecomplex (log (- r)) pi))))
```

Extension aux float :

```
(de #:float:log (r) (:log r))
```

(sqrt c) = c**1/2 = e**(log(c)/2)

```
(de :sqrt (c) (:exp (/ (:log c) 2)))
```

Pour étendre sqrt aux fix et float négatifs :

```
(de #:fix:sqrt (r)
    (selectq (<?> r 0)
      (1 (sqrt (float r)))
      (0 0)
      (-1 (makecomplex 0 (sqrt (- r))))))
(de #:float:sqrt (r) (#:fix:sqrt r))
```

CHAPITRE 11

Les Outils de Mise au Point

Matthieu Devin

Ce chapitre décrit plusieurs utilitaires, écrits en Lisp, qui permettent de mettre au point plus aisément les programmes Lisp. Ces utilitaires sont chargés automatiquement à leur premier appel; ils comprennent des outils de pistage (trace et pas à pas) et une boucle d'inspection qui permet d'examiner l'environnement en cas d'erreur. Tous ces outils utilisent le multifenêtrage virtuel si celui-ci est présent.

DEBUG [Feature]

Ce trait indique si les outils de mise au point sont présents en mémoire.

(DEBUGEND) [EXPR à 0 argument]

récupère la place occupée par les fonctions de pistage, de mise au point et de pas à pas. Le trait DEBUG n'est plus présent. Toutes ces fonctions seront de nouveau chargées à leur premier appel.

11.1 Le Pistage

Rappelons qu'il est possible de pister à l'exécution d'un programme tous les appels internes de la fonction interprète EVAL au moyen de la fonction TRACEVAL (voir cette fonction). Ce trait est utile pour comprendre le fonctionnement *interne* de l'évaluateur. Il est très peu utilisé pour la mise au point de programmes importants car il fournit beaucoup trop d'informations.

Il est également possible de pister les appels d'une fonction de l'utilisateur. Le pistage consiste à imprimer certaines informations choisies par l'utilisateur à l'entrée et à la sortie de la fonction. L'impression des informations peut être conditionnée par la valeur d'une expression Lisp évaluée à l'entrée et à la sortie de la fonction.

Il est enfin possible de créer conditionellement une boucle d'inspection à l'entrée de la fonction.

11.1.1 Les fonctions de pistage

(TRACE <trace1> ... <traceN>) [FEXPR]

TRACE permet de pister les fonctions décrites par les arguments <trace1>, <traceN>. Ces arguments ne sont pas évalués. Ce sont soit des noms de fonctions, soit des listes dont le CAR est un nom de fonction. Le premier type d'argument active le pistage standard de la fonction fournie. Le second type permet de préciser chaque paramètre du pistage pour la fonction concernée : formes évaluées en entrée et en sortie, trace et point d'arrêt conditionnels. Ces paramètres du pistage sont décrits dans la section 11.1.2. TRACE retourne la liste de toutes les fonctions traitées.

(UNTRACE <sym1> ... <symN>) [FEXPR]

permet d'enlever le pistage pour les différentes fonctions de nom <sym1>, <symN>. Ces noms ne sont pas évalués. Si aucun argument n'est donné (c'est-à-dire si on invoque (UNTRACE)), cette fonction permet d'enlever l'ensemble des pistages actifs. UNTRACE retourne la liste des fonctions traitées.

11.1.2 Les paramètres du pistage

Le pistage standard d'une fonction consiste à imprimer
- à l'entrée : le nom de la fonction, puis le nom de ses paramètres et leurs valeurs.
- à la sortie : le nom de la fonction suivi de la valeur qu'elle retourne

Lorsqu'un argument de la fonction TRACE est une liste celle-ci doit avoir la structure suivante :

```
(<fn> <par1> ... <parN>)
```

Où <fn> est le nom de la fonction à pister et <par1>, <parN> sont les paramètres.

Les différents paramètres reconnus par la fonction TRACE sont :

- (WHEN <e>) : conditionnement de la trace.
- (BREAK <e>) : conditionnement de la création d'un point d'arrêt.
- (STEP <e>) : conditionnement du passage en mode pas à pas.
- (ENTRY <e1> ... <en>) : choix des formes Lisp évaluées à l'entrée de la fonction.
- (EXIT <e1> ... <en>) : choix des formes Lisp évaluées à la sortie de la fonction.

Le paramètre WHEN est une forme Lisp qui est évaluée à l'entrée et à la sortie de la fonction. Si l'évaluation de cette forme rend () les paramètres BREAK, STEP et ENTRY (à l'entrée) et EXIT (à la sortie) ne sont pas évalués.

Le paramètre BREAK est une forme Lisp qui est évaluée à l'entrée de la fonction. Si l'évaluation ne rend pas () une *boucle d'inspection* est créée à l'entrée de la fonction (voir la fonction BREAK, section 11.2). Notez que le paramètre BREAK est évalué après le paramètre WHEN. Le point d'arrêt est créé uniquement lorsque le paramètre WHEN et le paramètre BREAK sont vrais.

Le paramètre STEP est une forme Lisp qui est évaluée après le paramètre BREAK mais avant le corps de la fonction tracée. Si sa valeur est différente de (), le corps de la fonction est évalué en mode pas à pas (voir la fonction STEP, section 11.3). Sinon le corps de la fonction est évalué normalement. Le paramètre STEP n'est évalué que lorsque le paramètre WHEN est vrai.

Toutes les opérations qui ont lieu *à l'entrée de la fonction* sont réalisées après la liaison des variables. Toutes les opérations qui ont lieu *à la sortie de la fonction* sont réalisées

après l'évaluation de la dernière forme de la fonction, et dans l'environnement local de la fonction. De plus, à la sortie de la fonction, la variable globale #:TRACE:VALUE contient la valeur que va retourner la fonction. On peut éventuellement utiliser cette variable pour imprimer le résultat de la fonction.

Si certains paramètres ne sont pas fournis ils prennent des valeurs par défaut qui correspondent à ceux du pistage standard. Ils sont définis comme suit pour une fonction FOO à deux arguments X et Y :

```
(trace (foo (when t)        ; la trace est toujours active
            (break ())      ; on ne crée pas de point d'arrêt
            (step ())       ; pas de mode pas à pas
            (entry (print "foo --> x=" x ", y=" y))
            (exit  (print "foo <-- " #:trace:value))))
```

11.1.3 Les variables globales du pisteur

#:TRACE:ARG1 [Variable]
#:TRACE:ARG2 [Variable]
#:TRACE:ARG3 [Variable]

Les valeurs de ces variables ne sont significatives que durant l'évaluation des paramètres WHEN, BREAK, ENTRY et EXIT d'une fonction compilée (prédéfinie ou définie par l'utilisateur) pistée.

Leur valeur dépend du type de la fonction pistée de la manière suivante :
- SUBR0 : pas de valeur significative.
- SUBR1 : #:TRACE:ARG1 contient la valeur du premier argument de la fonction
- SUBR2 : #:TRACE:ARG1 et #:TRACE:ARG2 contiennent les valeurs du premier et du second argument de la fonction.
- SUBR3 : #:TRACE:ARG1, #:TRACE:ARG2 et #:TRACE:ARG3 contiennent les valeurs du premier, du second et du troisième argument de la fonction.
- SUBRN : #:TRACE:ARG1 contient la liste des valeurs de tous les arguments de la fonction.
- FSUBR, MSUBR et DMSUBR : #:TRACE:ARG1 contient la liste des arguments non évalués de la fonction.

L'utilisation de ces variables dans les paramètres du pistage permet un pistage très fin des fonctions compilées.
En voici quelques exemples d'utilisation :

```
; Point d'arrêt sur CAR lorsque la valeur de l'argument est un nombre
(TRACE (CAR (WHEN (NUMBERP #:TRACE:ARG1)) (BREAK T)))

; Pistage de PUTPROP lorsque la valeur du troisième argument est ()
(TRACE (PUTPROP (WHEN (NULL #:TRACE:ARG3))))

; Pistage des appels à LIST avec 7 arguments
(TRACE (LIST (WHEN (= (LENGTH #:TRACE:ARG1) 7))))

; Pistage de tous les SETQ sur la variable FOO
(TRACE (SETQ (WHEN (EQ (CAR #:TRACE:ARG1) 'FOO))))
```

On peut aussi obtenir des effets très pervers de la fonction TRACE. L'appel

suivant renverse l'ordre des arguments de tous les appels à la fonction CONS.

```
(TRACE (CONS
         (ENTRY (PSETQ #:TRACE:ARG1 #:TRACE:ARG2
                       #:TRACE:ARG2 #:TRACE:ARG1))))
```

#:TRACE:VALUE [Variable]

Cette variable contient la dernière valeur retournée par une fonction pistée. Cette valeur n'est significative qu'à l'intérieur des formes correspondant au paramètre EXIT du pistage.

#:TRACE:NOT-IN-TRACE-FLAG [Variable]

Cette variable vaut () pendant l'évaluation des paramètres du pistage, T dans les autres cas. Elle est utilisée par le pisteur comme condition supplémentaire au pistage des fonctions. Les fonctions ne sont effectivement pistées que lorsque cette variable vaut (). Ceci permet d'utiliser des fonctions pistées pendant l'évaluation des paramètres du pistage sans risque de boucle infinie. On peut par exemple pister la fonction PRINT, bien qu'elle soit utilisée pour imprimer les messages du pistage.

Voici un exemple de pistage de la fonction PRINT. Attention, on piste également les appels à PRINT du TOPLEVEL Lisp
(PRINT (EVAL (READ))) :

```
? (trace print)
= print ---> ((print))
(print)
print <--- (print)
? (print 1 2 3)
print ---> (1 2 3)
123
print <--- 3
= print ---> (3)
3
print <--- 3
? (untrace)
= (print)
```

#:TRACE:TRACE [Variable]

Cette variable contient la liste de toutes les fonctions pistées à un instant donné.

11.1.4 Exemple d'utilisation du pisteur

Nous reproduisons ci-dessous une session Lisp illustrant l'utilisation du pisteur.

```
? (debug t) ; passage en mode mise au point
= t
?
? (de rv (s res)
?     (if (null s)
?         res
?         (rv (cdr s) (cons (car s) res))))
```

```
= rv
?
? ; essai normal sans pistage
? (rv '(1 2 3) ())
= (3 2 1)
?
? ; pistage standard
? (trace rv)
= (rv)
? (rv '(1 2 3) ())
rv ---> s=(1 2 3) res=()
rv ---> s=(2 3) res=(1)
rv ---> s=(3) res=(2 1)
rv ---> s=() res=(3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
= (3 2 1)
?
? ; Réglage des informations imprimées en entrée et en sortie
? (trace (rv (entry (print "rv: res=" res))
?            (exit (print "rv retourne:" #:trace:value))))
= ((rv (entry (print rv: res= res)) (exit (print rv retourne:
#:trace:value))))
? (rv '(1 2 3) ())
rv: res=()
rv: res=(1)
rv: res=(2 1)
rv: res=(3 2 1)
rv retourne:(3 2 1)
rv retourne:(3 2 1)
rv retourne:(3 2 1)
rv retourne:(3 2 1)
= (3 2 1)
?
? ; Pistage conditionnel
? (trace (rv (when (< (length s) 2))))
= ((rv (when (< (length s) 2))))
? (rv '(1 2 3) ())
rv ---> s=(3) res=(2 1)
rv ---> s=() res=(3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
= (3 2 1)
?
? ; Invocation d'une boucle d'inspection sur condition
? (trace (rv (break (null s))))
= ((rv (break (null s))))
? (rv '(1 2 3) ())
rv ---> s=(1 2 3) res=()
rv ---> s=(2 3) res=(1)
rv ---> s=(3) res=(2 1)
```

```
rv ---> s=() res=(3 2 1)
** rv : break : tracebreak
(de rv (s res)
    (if (null s) res (rv (cdr s) (cons (car s) res))))
>? v
 s=()
 res=(3 2 1)
>? r
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
= (3 2 1)
?
? ; Utilisation du mode pas à pas
? (trace (rv (step (= 1 (length s)))))
= ((rv (step (= 1 (length s)))))
? (rv '(1 2 3) ())
rv ---> s=(1 2 3) res=()
rv ---> s=(2 3) res=(1)
rv ---> s=(3) res=(2 1)
 1 -> (if (null s) res (rv (cdr s) (cons & res))) step>? return
  2 -> (null s) step>? return
   3 -> s step>? return
   3 <- (3)
  2 <- ()
  2 -> (rv (cdr s) (cons (car s) res)) step>? return
   3 -> (cdr s) step>? return
    4 -> s step>? return
    4 <- (3)
   3 <- ()
   3 -> (cons (car s) res) step>? <
   3 <- (3 2 1)
rv ---> s=() res=(3 2 1)
    4 -> (if (null s) res (rv (cdr s) (cons & res))) step>? <
    4 <- (3 2 1)
rv <--- (3 2 1)
  2 <- (3 2 1)
 1 <- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
= (3 2 1)
?
? ; Suppression du pistage
? (untrace rv)
= (rv)
```

11.2 Le Mode Arrêt et Mise au Point

Quand une erreur se déclenche un message est imprimé puis l'environnement est restauré ce qui réinitialise tous les objets dynamiques. Or il est parfois souhaitable de rester dans l'environnement de l'erreur pour examiner les variables ou la pile *au moment de l'erreur.* Ceci est réalisé en passant en mode mise au point.

(DEBUG <s>) [FEXPR]

Si l'argument <s>, qui n'est pas évalué est égal à T, passage en mode mise au point globalement. Si l'argument vaut (), le mode mise au point est désactivé globalement. Enfin si l'argument <s> est une expression quelconque, le mode mise au point ne sera actif que durant l'évaluation de cette expression. DEBUG retourne en valeur la valeur de l'évaluation de <s> (ce qui permet d'encapsuler temporairement toute expression Lisp avec la fonction DEBUG pour la mettre au point).

L'appel sans argument de cette fonction retourne la valeur de l'indicateur global de mise au point, T ou ().

(BREAK) [EXPR à 0 argument]

Hors du mode mise au point cette fonction se limite à provoquer un retour immédiat au TOPLEVEL Lisp en invoquant la fonction ERR (voir cette fonction). En mode mise au point elle crée une *boucle d'inspection* dans l'environnement dynamique de son appel. La boucle d'inspection est caractérisée par le signe d'invite ">? ". Sous la boucle d'inspection on peut retourner au TOPLEVEL Lisp en évaluant (EXIT BREAK) ou en abrégé T. Cette fonction est systématiquement appelée par la fonction de traitement d'erreurs standard SYSERROR. En mode mise au point, toute erreur de l'utilisateur provoque donc la création d'une boucle d'inspection.

Si une nouvelle erreur se produit dans la boucle d'inspection, BREAK n'est appelée récursivement que si la forme (DEBUG T) a été évaluée à l'interieur de la première boucle d'inspection. En ce cas, le signe d'invite indique explicitement le nombre d'appels à BREAK imbriqués.

11.2.1 La boucle d'inspection

Une boucle d'inspection est une boucle READ-EVAL-PRINT qui a lieu dans un environnement local à une fonction. Elle est créée en mode mise au point par tout appel à la fonction BREAK, c'est-à-dire lors d'une erreur, lors d'un point d'arrêt (voir fonction TRACE), ou bien lors d'un appel explicite de l'utilisateur à cette fonction. Une boucle d'inspection est caractérisée par le signe d'invite ">? ".

On peut terminer une boucle d'inspection en retournant au TOPLEVEL Lisp ou bien en continuant le calcul interrompu (par un point d'arrêt par exemple).

A l'entrée d'une boucle d'inspection la fonction qui a provoqué cette boucle est affichée à l'écran, et, si possible, la partie de la fonction en cours d'évaluation est mise en valeur (avec la fonction TYATTRIB).

Dans une boucle d'inspection il est possible d'évaluer n'importe quelle expression Lisp (comme au TOPLEVEL du système). L'évaluation des expressions a cependant lieu dans l'environnement local de la fonction qui a appelé la boucle d'inspection : il est donc possible d'examiner les valeurs des paramètres de cette fonction. De plus un

certain nombre de *commandes* permettent de dépiler ou de rempiler les environnements des différents appels en attente dans la pile. On peut donc ainsi examiner les valeurs des paramètres et des variables locales de toutes les fonctions en attente.

Les commandes sont des caractères qui doivent être tapés en début de ligne. Si vous devez évaluer des formes Lisp commençant par un caractère de commande vous devez précéder ces formes du caractère *espace*.

Les commandes de la boucle d'inspection sont les suivantes :

- v : impression des variables locales à la fonction courante.
- e : impression en clair de l'erreur qui a provoqué la boucle d'inspection.
- . : impression de la fonction courante
- + : dépilage d'un appel de fonction
- - : ré-empilage d'un appel de fonction
- h (histoire) : impression des premiers appels de fonctions à partir de la fonction courante. Le nombre de niveaux imprimés est donné par la valeur de la variable globale #:SYSTEM:STACK-DEPTH.
- H : impression de tous les appels de fonction en suspens.
- q : sortie de la boucle d'inspection.
- t : retour immédiat au toplevel Lisp.
- r : reprise du programme qui a créé la boucle d'inspection. Cette commande permet de relancer l'évaluation après un point d'arrêt et de corriger les erreurs *variable indéfinie (ERRUDV)* et *fonction indéfinie (ERRUDF)*.
- z : continuation en mode pas à pas d'une fonction interrompue par un point d'arrêt.
- ? : liste les commandes de la boucle d'inspection.

Les commandes qui permettent de terminer la boucle d'inspection réalisent d'abord un réempilage de tous les blocs éventuellement dépilés.

```
? ; Passage en mode mise au point globalement
? (debug t)
= t
? ; Définition d'une fonction incorrecte
? (de foo (x)
?     (if (= x 0)
?         (lisp 1)
?         (cons (1+ x) x (foo (1- x)))))
= foo
? ; Essai de cette fonction
? (foo 3)
** cons : mauvais nombre d'arguments : 2
(ds cons subr2)
(de foo (x)
    (if (= x 0) (lisp 1) (cons (1+ x) x (foo (1- x)))))
>? ; La fonction est visualisée.
>? ; La partie fautive (cons...) est mise en valeur.
>? ; Il fallait deux arguments à la fonction CONS.
>? ; retour au toplevel Lisp
>? t
? ; Correction de la fonction
? (de foo (x)
?     (if (= x 0)
```

```
?         (lisp 1)
?         (mcons (1+ x) x (foo (1- x)))))
** de : fonction redefinie : foo
= foo
? (foo 3)
** eval : fonction indefinie : lisp
(de foo (x)
    (if (= x 0) (lisp 1) (mcons (1+ x) x (foo (1- x)))))
>? ; La partie ayant provoqué l'erreur (lisp 1) est mise en valeur.
>? ; examen des variables locales
>? v
 x=0
>? ; Evaluation de formes Lisp
>? x
= 0
>? (mcons (1+ x) x '(1))
= (1 0 1)
>? ; historique des appels de fonctions en suspens
>? h
   [stack   4]   (foo ...)
   [stack   3]   (foo ...)
   [stack   2]   (foo ...)
   [stack   1]   (foo ...)
>? ; liste des commandes disponibles
>? ?
; v:   show variables
; h:   print top of stack
; H:   print complete stack
; e:   show error message
; .:   show current stack frame
; +:   down stack
; -:   up stack
; t:   back to toplevel
; q:   exit inspection loop
; r:   resume
; z:   step traced functions
; ?:   list commands
>? ; Dépilage d'un appel de fonction
>? +
(de foo (x)
    (if (= x 0) (lisp 1) (mcons (1+ x) x (foo (1- x)))))
>? ; x vaut 1 dans cet environnement
>? v
 x=1
>? ; rempilage de l'appel
>? -
(de foo (x)
    (if (= x 0) (lisp 1) (mcons (1+ x) x (foo (1- x)))))
>? ; continuation du calcul en corrigeant l'erreur "fonction indéfinie"
>? r
Function >? list
= (4 3 3 2 2 1 1)
? ; Il faut corriger la fonction FOO
```

```
? (de foo (x)
?     (if (= x 0)
?         (list 1)
?         (mcons (1+ x) x (foo (1- x)))))
** de : fonction redefinie : foo
= foo
? (foo 3)
= (4 3 3 2 2 1 1)
? ; Point d'arrêt sur condition
? (trace (foo (break (= x 0))))
= ((foo (break (= x 0))))
? (foo 5)
foo ---> x=5
foo ---> x=4
foo ---> x=3
foo ---> x=2
foo ---> x=1
foo ---> x=0
** foo : break : tracebreak
(de foo (x)
    (if (= x 0) (list 1) (mcons (1+ x) x (foo (1- x)))))
>? v
 x=0
>? ; Modifions le paramètre de FOO!
>? (setq x 4)
= 4
>? ; reprise du calcul
>? r
foo ---> x=3
foo ---> x=2
foo ---> x=1
foo ---> x=0
** foo : break : tracebreak
(de foo (x)
    (if (= x 0) (list 1) (mcons (1+ x) x (foo (1- x)))))
>? ; De nouveau le point d'arrêt
>? ; on termine le calcul
>? r
foo <--- (1)
foo <--- (2 1 1)
foo <--- (3 2 2 1 1)
foo <--- (4 3 3 2 2 1 1)
foo <--- (5 4 4 3 3 2 2 1 1)
foo <--- (2 1 5 4 4 3 3 2 2 1 1)
foo <--- (3 2 2 1 5 4 4 3 3 2 2 1 1)
foo <--- (4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
foo <--- (5 4 4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
foo <--- (6 5 5 4 4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
= (6 5 5 4 4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
```

11.2.2 Les fonctions du mode mise au point

Les fonctions ci-dessous peuvent être utiles à l'utilisateur qui désire étendre le mode mise au point.

(PRINTSTACK <n> <s>) [EXPR à 0, 1 ou 2 arguments]

visualise les <n> derniers niveaux de la pile d'exécution dynamique Lisp ou de toute cette pile si l'argument <n> n'est pas fourni. Si l'argument <s> est fourni ce doit être une liste rendue par la fonction CSTACK (voir cette fonction), c'est-à-dire une liste qui représente un état de la pile Lisp, et c'est le contenu de cette liste qui est imprimé au lieu du contenu de la pile.

(DEBUG-COMMAND <cn>) [EXPR à 1 argument]

<cn> est un code interne d'un caractère. Cette fonction exécute la commande de la boucle d'inspection associée à ce caractère.

#:SYSTEM:STACK-DEPTH [Variable]

La valeur de cette variable est un nombre qui indique le nombre de niveaux de pile que doit imprimer la commande **h** de la boucle d'inspection. La valeur initiale de cette variable est 5. Cette variable est ignorée par la commande S qui imprime le contenu de toute la pile.

#:SYSTEM:DEBUG-LINE [Variable]

Cette variable permet de limiter le nombre de lignes des divers messages informatifs imprimés par la boucle d'inspection. Sa valeur indique le nombre maximum de lignes à imprimer. La valeur initiale de cette variable est 5. Cette variable est particulièrement utile pour limiter la visualisation des fonctions ayant provoqué des erreurs.

11.3 L'Exécution Pas à Pas

L'exécution en mode pas à pas permet de s'arrêter à chaque appel interne de l'évaluateur. Durant ces arrêts l'utilisateur peut consulter la valeur des objets dynamiques et suivre très finement le déroulement de ses programmes.

Pour réaliser cette exécution incrémentale le système, à chaque appel interne de l'évaluateur, invoque l'interruption programmable STEPEVAL avec comme argument la forme à évaluer.

Voici les différentes actions à chaque arrêt :

return continuer d'évaluer l'expression courante en mode pas à pas.

< évaluer l'expression courante sans arrêt et reprendre le mode pas à pas au retour de cette évaluation.

q arrêter le mode pas à pas et retourner la valeur de l'expression de départ.

= créer une boucle d'inspection. Toutes les commandes de la boucle d'inspection sont alors disponibles. On revient au pas à pas par la commande *r* de la boucle d'inspection.

h imprimer un historique des évaluations en suspens.

? imprimer un resumé de ces commandes

(STEP <s>) [FSUBR]

Evalue l'expression <s> en mode pas à pas. Retourne la valeur de l'évaluation de <s>.

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

arrête momentanément le mode pas à pas, évalue les différentes expressions <s1> ... <sN> et retourne la valeur de <sN>.

```
? ; Passage en mode mise au point globalement
? (debug t)
= t
? (de f (n)
?     (if (<= n 1)
?         1
?         (* (f (1- n)) n)))
= f
?
? (step (f 5))
 1 -> 5 step> 1 <- 5
 1 -> (if (<= n 1) 1 (* (f &) n)) step>? return
  2 -> (<= n 1) step>? return
   3 -> n step>? return
   3 <- 5
   3 -> 1 step>? return
   3 <- 1
  2 <- ()
  2 -> (* (f (1- n)) n) step>? return
   3 -> (f (1- n)) step>? return
    4 -> (1- n) step>? return
     5 -> n step>? return
     5 <- 5
    4 <- 4
    4 -> (if (<= n 1) 1 (* (f &) n)) step>? return
     5 -> (<= n 1) step>? return
      6 -> n step>? return
      6 <- 4
      6 -> 1 step>? return
      6 <- 1
     5 <- ()
     5 -> (* (f (1- n)) n) step>? ?
;Les commandes de pas a pas sont :
;   CR  pour passer a l'expression suivante
;   .   pour voir l'expression courante
;   =   appel un fois le toplevel
;   <   pour evaluer sans pas a pas et y revenir
;   q   retour au toplevel
;   h   pour avoir l'historique du pas a pas
```

```
;   ?   pour avoir ce texte ....
     5 -> (* (f (1- n)) n) step>? h
1 (if (<= n 1) 1 (* (f (1- n)) n))
 2 (* (f (1- n)) n)
  3 (f (1- n))
   4 (if (<= n 1) 1 (* (f (1- n)) n))
    5 (* (f (1- n)) n)
     5 -> (* (f (1- n)) n) step>? =
** step : break : (* (f (1- n)) n)
(de f (n)
    (if (<= n 1) 1 (* (f (1- n)) n)))
>? v
 n=4
>? h
   [stack   2]   (f ...)
   [stack   1]   (f ...)
>? r
     5 -> (* (f (1- n)) n) step>? return
      6 -> (f (1- n)) step>? <
      6 <- 6
      6 -> n step>? return
      6 <- 4
     5 <- 24
    4 <- 24
   3 <- 24
   3 -> n step>? return
   3 <- 5
  2 <- 120
 1 <- 120
= 120
```

CHAPITRE 12

Le Chargeur/Assembleur LLM3

Le coeur du système Le_Lisp est écrit dans le langage de la machine virtuelle LLM3 [Chailloux 85b]. Un sous-ensemble de ce langage est également disponible en Lisp. Il est bien évidemment utilisé par les différents compilateurs mais peut également être utilisé pour décrire de nouvelles fonctions standard ou pour écrire des compilateurs de langages autre que Lisp. Ce chapitre décrit les fonctions qui permettent d'accéder aux ressources internes de la machine et la syntaxe Lisp utilisée par le chargeur/assembleur LLM3. Ce chargeur est souvent nommé LAP, pour *Lisp Assembly Program.*

12.1 L'Accès à la Mémoire et à l'Unité Centrale

Certaines fonctions permettent d'accéder directement à la mémoire et à l'unité centrale. Le chargeur/assembleur LLM3 utilise une zone mémoire spéciale, la *zone code*, pour y charger les instructions. Les fonctions qui vont suivre sont évidemment dépendantes de la machine et servent à construire le chargeur/assembleur LLM3.

Ces fonctions utilisent des adresses mémoire quelconques. Les nombres entiers Le_Lisp, codés sur 16 bits, ne permettent pas toujours de représenter une adresse parfois codée 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)
- un CONS 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'écrire    (#$1F . #$3C00)
```

Les fonctions qui vont suivre testent la validité des arguments de type adresse mémoire. Elles peuvent déclencher l'erreur ERRNDA dont le libellé par défaut est :

```
** <fn> : l'argument n'est pas une adresse : <s>      ou bien
** <fn> : bad adress : <s>
```

dans lequel <fn> est le nom de la fonction appelée et <s> l'argument qui n'est pas de type adresse mémoire.

(#:SYSTEM:CCODE <adr>) [SUBR à 0 ou 1 argument]

permet de lire (ou d'écrire si l'argument <adr> est fourni), l'adresse mémoire courante de chargement dans la zone code.

(#:SYSTEM:ECODE <adr>) [SUBR à 0 ou 1 argument]

permet de lire (ou d'écrire si l'argument <adr> est fourni), l'adresse mémoire de fin de la zone code.

(ADDADR <adr1> <adr2>) [SUBR à 2 arguments]

retourne la somme des 2 adresses mémoire <adr1> et <adr2>. ADDADR alloue un nouveau CONS si le résultat de cette fonction doit être représenté sous la forme d'un CONS.

```
ex : (ADDADR '(3 . 657) '(4 . 4567))    -> (7 . 5224))
```

(SUBADR <adr1> <adr2>) [SUBR à 2 arguments]

retourne la différence des 2 adresses mémoire <adr1> et <adr2>. SUBADR alloue un nouveau CONS si le résultat de cette fonction doit être représenté sous la forme d'un CONS.

```
ex : (SUBADR '(56 . 7899) '(45 . 3333))    -> (11 . 4566)
```

(INCRADR <adr> <n>) [SUBR à 2 arguments]

ajoute <n> à l'adresse mémoire <adr>. Si cette adresse mémoire est représentée sous la forme d'un CONS, INCRADR modifie physiquement ce CONS.

```
ex : (SETQ adr '(3 . #$FFFE))       -> (3 . -2)
     (INCRADR adr 1)                -> (3 . -1)
     (INCRADR adr 1)                -> (4 . 0)
     (INCRADR adr 1)                -> (4 . 1)
     (INCRADR adr 100)              -> (4 . 101)
     adr                            -> (4 . 101)
```

(GTADR <adr1> <adr2>) [SUBR à 2 arguments]

retourne T si l'adresse mémoire <adr1> est plus grande que l'adresse mémoire <adr2>.

```
ex : (SETQ adr '(4 . 101))          -> (4 . 101)
     (GTADR adr '(4 . -10))         -> ()
     (GTADR adr '(4 . 100))         -> T
     (GTADR adr '(4 . 101))         -> ()
```

(LOC <s>) [SUBR à 1 argument]

permet de connaître l'adresse mémoire absolue de l'objet Lisp <s>. L'adresse mémoire 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 créés dynamiquement. LOC ne peut pas être décrit en Lisp.

```
ex : (LOC '(A B C))       -> (15 . 288)
     (LOC '(A B C))       -> (13 . 1200)
```

(VAG <adr>) [SUBR à 1 argument]

est la fonction inverse de LOC. VAG retourne l'objet Lisp dont l'adresse mémoire absolue est fournie en argument.
Donc (VAG (LOC <s>)) = <s>.
VAG ne peut pas être décrit en Lisp.

Attention : VAG ne teste pas si <adr> est l'adresse mémoire d'un véritable objet Lisp, un appel erroné de cette fonction peut provoquer une erreur du système hôte. En particulier il est possible de faire référence à des objets disparus :

```
ex : (LET ((x (LOC (CONS 'a 'b))))
          (GC)
          (VAG x))        ; la valeur de x n'existe plus!
                          ; VAG retourne un pointeur sur la
                          ; liste libre des CONS!
```

(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 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. MEMORY retourne en valeur un nombre représentant la valeur du mot après modification éventuelle si le 2ème argument <n> est fourni. Cette fonction a en général des limitations en fonction des unités centrales utilisées : mots de 8 ou 16 bits, adresses sur des frontières de mots ...

(CALL <adr> <a1> <a2> <a3>) [SUBR à 4 arguments]

le premier argument <adr> doit être l'adresse mémoire d'un sous-programme en mémoire. CALL va lancer ce sous-programme après avoir chargé les trois accumulateurs LLM3 A1, A2, et A3 avec les valeurs respectives <a1>, <a2> et <a3>. Ce format d'appel est celui des SUBR0, SUBR1, SUBR2, SUBR3, FSUBR, MSUBR ou DMSUBR (voir la section sur les appels de fonctions). Cette fonction doit être utilisée avec précaution car elle ne teste pas la validité du code qui est lancé.
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, 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 mémoire <adr>. Le nombre d'arguments empilés sera disponible dans l'accumulateur A4. Ce format est celui des NSUBR (voir la section sur les appels de fonctions).
Attention : le code invoqué au moyen de CALLN doit impérativement nettoyer la pile de ses arguments (dont le nombre est dans A4 à l'entrée de la fonction par exemple au moyen de l'instruction LLM3, ADJSTK), se terminer par l'instruction LLM3, RETURN, et charger le registre A1 avec une valeur Lisp qui est retournée par la fonction CALLN.

12.2 Le Chargeur Mémoire LLM3

LOADER [Feature]

ce trait indique si le chargeur/assembleur LLM3 est présent en mémoire.

(LOADER <l> <i>) [SUBR à 2 arguments]

permet de charger en mémoire la liste d'instructions LLM3 <l>. Le format des instructions est décrit dans la section suivante. Si l'indicateur <i> est vrai, le chargeur va éditer dans le flux de sortie courant la liste d'assemblage de ces instructions dans le langage machine de la machine hôte. LOADER retourne () en valeur.

#:LD:SPECIAL-CASE-LOADER [Variable]

La pseudo-instruction FENTRY change la valeur fonctionnelle des fonctions durant le chargement. Dans certains cas d'amorçage difficiles (comme par exemple pour charger le chargeur lui-même), il faut résoudre les valeurs fonctionnelles à la fin du chargement (c'est-à-dire à l'apparition de la pseudo-instruction END). L'indicateur #:LD:SPECIAL-CASE-LOADER vaut T dans ce cas spécial mais vaut () par défaut.

#:LD:SHARED-STRINGS [Variable]

Indique s'il faut partager les constantes de chaînes de caractères pendant le chargement de la liste d'instructions.

12.3 Le Format des Instructions LLM3

Une liste d'instructions LLM3 contient des atomes représentant des étiquettes et des listes représentant des instructions. Une instruction est de nouveau une liste de la forme :

```
(<codop> <op1> ... <opN>)
```

Dans laquelle <codop> est le mnémonique de l'instruction et <op1> ... <opN> les différents opérandes de cette instruction. Seul le champ <codop> est obligatoire.

12.4 Les Modules et Les Etiquettes

Un module est un ensemble de listes d'instructions débutant par la pseudo-instruction TITLE et se terminant par la pseudo-instruction END.

Il existe 3 types d'étiquettes :
- les étiquettes locales à une liste d'instructions
- les étiquettes locales à un module
- les étiquettes globales

Les *étiquettes locales à une liste d'instructions* sont représentées par des nombres entiers ou par des symboles déclarés au moyen de la pseudo-instruction LOCAL. Une

étiquette symbolique locale à une liste d'instructions doit être déclarée avant sa première utilisation. Ces étiquettes doivent être résolues à la fin de la liste d'instructions ou à l'apparition de la pseudo-instruction ENDL. Toute référence à ce type d'étiquette engendre un déplacement par rapport au compteur ordinal.

Les *étiquettes locales à un module* sont toujours symboliques. Elles sont définies au moyen de la pseudo-instruction ENTRY. Elles peuvent ne pas être définies entre deux appels du chargeur mais doivent l'être à l'apparition de la pseudo-instruction END. Toute référence à ce type d'étiquette engendre un déplacement par rapport au compteur ordinal.

Les *étiquettes globales* sont toujours symboliques. Elles sont définies au moyen de la pseudo-instruction FENTRY. Toute référence à ce type d'étiquette engendre un accès à la valeur fonctionnelle du symbole. Toutes les fonctions standard sont de ce type.

12.5 Les Opérandes des Instructions LLM3

LLM3 dispose de 4 types d'opérandes :

- les registres rapides
- les valeurs immédiates d'objets Lisp
- les champs des objets Le_Lisp
- les étiquettes

A1 **[opérande LLM3]**
A2 **[opérande LLM3]**
A3 **[opérande LLM3]**
A4 **[opérande LLM3]**

ces 4 opérandes servent à représenter les registres (accumulateurs) de la machine LLM3.

NIL **[opérande LLM3]**

cet opérande sert à représenter le symbole spécial marqueur de fin de liste ou bien la valeur fausse, c'est-à-dire le symbole vide ||.

(QUOTE <exp>) **[opérande LLM3]**

cet opérande sert à représenter une constante Le_Lisp de n'importe quel type.

(@ <lab>) **[opérande LLM3]**

cet opérande sert à représenter l'adresse d'une étiquette LLM3.

(CAR <accu>) **[opérande LLM3]**
(CDR <accu>) **[opérande LLM3]**

ces opérandes servent à représenter l'accès aux constituants d'un élément de liste.

(CVAL <accu>) **[opérande LLM3]**
(PLIST <accu>) **[opérande LLM3]**
(FVAL <accu>) **[opérande LLM3]**
(PKGC <accu>) **[opérande LLM3]**
(OVAL <accu>) **[opérande LLM3]**
(ALINK <accu>) **[opérande LLM3]**
(PNAME <accu>) **[opérande LLM3]**

ces opérandes servent à représenter l'accès aux différents constituants (propriétés naturelles) d'un symbole.

(CVALQ <symb>) **[opérande LLM3]**

cet opérande permet d'accéder directement à la valeur (CVAL) du symbole de nom <symb>.

(TYP <accu>) **[opérande LLM3]**
(VAL <accu>) **[opérande LLM3]**

ces opérandes servent à représenter l'accès aux constituants d'un objet de type chaîne de caractères ou vecteur de S-expressions.

(& <n>) **[opérande LLM3]**

cet opérande sert à représenter le <n>ième mot empilé. Le sommet de pile correspond à l'opérande (& 0), le sous-sommet (& 1) ...

(EVAL <e>) **[opérande LLM3]**

cet opérande permet d'appeler dynamiquement l'évaluateur Le_Lisp qui doit être employé dans l'instruction. La valeur retournée par EVAL doit être un opérande LLM3 légal.

Les étiquettes représentées par <lab> sont toujours de type locales à une liste d'instructions ou bien locales à un module.

12.6 Les Pseudos-Instructions

(TITLE <symb>) **[pseudo-instruction LLM3]**

déclare le nom d'un module dont c'est la première instruction.

(LOCAL <symb>) **[pseudo-instruction LLM3]**

permet de définir une étiquette symbolique locale à une liste d'instructions. Par défaut les étiquettes numériques sont toujours locales et les étiquettes symboliques toujours globales.

(FENTRY <symb> <ftype>) **[pseudo-instruction LLM3]**

<symb> est le nom du symbole Lisp auquel va être attaché une fonction globale (de type SUBR) décrite en LLM3. <ftype> est un des types fonctionnels de Le_Lisp qui peut être : SUBR0, SUBR1, SUBR2, SUBR3, NSUBR, FSUBR, MSUBR ou DMSUBR.

(ENTRY ‹symb› ‹ftype›) [pseudo-instruction LLM3]

‹symb› est le nom du symbole Lisp auquel va être attaché une fonction locale (de type SUBR) décrite en LLM3. ‹ftype› est un des types fonctionnels de Le_Lisp qui peut être : SUBR0, SUBR1, SUBR2, SUBR3, NSUBR, FSUBR, MSUBR ou DMSUBR. Après le chargement cette fonction n'est plus disponible.

(ENDL) [pseudo-instruction LLM3]

termine le chargement d'une fonction locale. Toutes les étiquettes locales doivent être résolues et sont oubliées.

(END) [pseudo-instruction LLM3]

termine le chargement d'un module.

(EVAL ‹e›) [pseudo-instruction LLM3]

évalue, dans le contexte du chargeur l'expression ‹e›. A priori ne charge rien en mémoire (sauf demande explicite).

12.7 Les Instructions de base

12.7.1 Les transferts de pointeurs

En plus de l'instruction MOV qui permet de transférer n'importe quel objet Lisp, il existe d'autres instructions spécialisées dans le transfert de pointeurs ou d'octets contenus dans le tas (HEAP) ou dans la pile.

(MOV ‹op1› ‹op2›) [instruction LLM3]

transfère le pointeur contenu dans l'opérande source ‹op1› dans l'opérande destination ‹op2›. L'opérande ‹op2› ne peut pas être une valeur immédiate.

12.7.2 Les comparaisons de pointeurs

(CABEQ ‹op1› ‹op2› ‹lab›) [instruction LLM3]

réalise une comparaison de pointeurs. Si l'opérande ‹op1› est égal à l'opérande ‹op2›, branchement à l'étiquette locale ‹lab›.

(CABNE ‹op1› ‹op2› ‹lab›) [instruction LLM3]

réalise une comparaison de pointeurs. Si l'opérande ‹op1› n'est pas égal à l'opérande ‹op2›, branchement à l'étiquette locale ‹lab›.

12.7.3 Le contrôle

(BRA ‹lab›) [instruction LLM3]

branchement à l'étiquette locale ‹lab› qui doit être à l'intérieur du module LLM3. Le branchement est réalisé par déplacement par rapport au compteur ordinal.

(JMP <symb>) [instruction LLM3]

branchement à l'adresse de la valeur fonctionnelle associée au symbole <symb>. Le branchement réalisé est indirect à travers la FVAL du symbole <symb>. Les branchements vers le code des fonctions standard utilisent JMP et non BRA.

(BRI <op>) [instruction LLM3]

réalise un branchement indirect à l'adresse contenue dans l'opérande <op>.

(BRX <llab> <op>) [instruction LLM3]

réalise un aiguillage à travers la liste d'étiquettes <llab>. <op> est l'index à utiliser compté à partir de 0. La liste d'étiquettes est une liste d'opérandes de type référence à une étiquette, c'est-à-dire (@ <lab>).

(SOBGEZ <op> <lab>) [instruction LLM3]

décrémente l'opérande <op>. Si cet opérande est plus grand ou égal à 0, branchement à l'étiquette <lab>.

(NOP) [instruction LLM3]

occupe de la place, prend du temps mais ne fait pas d'action visible...

12.8 La Pile

LLM3 dispose d'une pile unique pour le contrôle et les données. Elle est utilisée comme opérande implicite dans des instructions spécialisées. Ces instructions qui utilisent le registre pointeur de pile <SP> ne nécessitent pas de sens de construction privilégié de la pile ni de test de débordement implicite de celle-ci. La pile occupe en général 4k objets.

12.8.1 Gestion du pointeur de pile

deux instructions permettent de manipuler explicitement le pointeur de pile <SP>.

(STACK <op>) [instruction LLM3]

transfère le contenu actuel du pointeur de pile dans l'opérande <op>.

(SSTACK <op>) [instruction LLM3]

transfère l'opérande <op> dans le pointeur de pile.

12.8.2 Pile de contrôle

trois instructions permettent de manipuler des adresses de retour stockées dans la pile.

(CALL <lab>) [instruction LLM3]

rajoute en sommet de pile la valeur courante du compteur ordinal et fait un branchement à l'étiquette locale <lab> qui doit se trouver dans le module LLM3 courant. Le branchement réalisé est direct er relatif.

(CALLI <op>) [instruction LLM3]

rajoute en sommet de pile la valeur courante du compteur ordinal et fait un branchement à l'adresse contenue dans l'opérande <op>.

(JCALL <symb>) [instruction LLM3]

est identique à l'instruction précédente mais le branchement peut avoir lieu dans un autre module LLM3. Le branchement réalisé est indirect au travers de la FVAL du symbol <symb>. Tous les appels aux fonctions standard doivent utiliser JCALL et non CALL.

(RETURN) [instruction LLM3]

dépile une adresse qui devient le nouveau compteur ordinal.

12.8.3 Pile de données

(PUSH <op>) [instruction LLM3]

empile l'opérande <op>. Le pointeur de pile est mis à jour.

(POP <op>) [instruction LLM3]

dépile le sommet de pile dans l'opérande <op>. Le pointeur de pile est mis à jour.

(ADJSTK <op>) [instruction LLM3]

ajuste le pointeur de pile de telle sorte que les <op> derniers objets empilés soient enlevés de la pile. Si <op> est négatif, création de <n> emplacements en sommet de pile.

Pour les 2 instructions suivantes, l'indice du sommet de pile vaut 0, celui du sous-sommet 1, du sous-sous sommet 2 ....

(MOVXSP <op1> <op2>) [instruction LLM3]

transfère l'opérande <op1> dans le <op2>ième emplacement de la pile.

(XSPMOV <op1> <op2>) [instruction LLM3]

transfère l'objet contenu dans le <op1>ième emplacement de la pile dans l'opérande <op2>.

12.9 Les Cellules de Liste (CONS)

12.9.1 Test du type cellule de liste

(BTCONS <op> <lab>) [instruction LLM3]

si l'opérande <op> est de type cellule de liste, branchement à l'adresse <lab>.

(BFCONS <op> <lab>) [instruction LLM3]

si l'opérande <op> n'est pas de type cellule de liste, branchement à l'adresse <lab>.

12.9.2 Accès aux champs des cellules de liste

au moyen d'opérandes de la machine LLM3. Ces opérandes ne sont valides que pour des pointeurs sur des cellules de liste. Le code du système Lisp écrit en LLM3 s'assure toujours que <accu> contient bien un pointeur de type cellule de liste avant d'utiliser ces opérandes.

```
(CAR <accu>)      [opérande LLM3]
(CDR <accu>)      [opérande LLM3]
```

12.9.3 Création d'une cellule de liste

Il n'existe pas d'instruction spéciales; il faut appeler explicitement les fonctions Le_Lisp CONS, NCONS ou XCONS ...

12.10 NIL

NIL est un opérande contenant la valeur de l'indicateur fin de liste ou bien de la valeur fausse.

(BTNIL <op> <lab>) [instruction LLM3]

si l'opérande <op> est égal à NIL branchement à l'adresse <lab>.

(BFNIL <op> <lab>) [instruction LLM3]

si l'opérande <op> n'est pas égal à NIL branchement à l'adresse <lab>.

12.11 Les Symboles

12.11.1 Test de type symbole

(BTSYMB <op> <lab>) [instruction LLM3]

si l'opérande <op> est un symbole, branchement à l'étiquette <lab>.

(BFSYMB <op> <lab>) [instruction LLM3]

si l'opérande <op> n'est pas un symbole, branchement à l'étiquette <lab>.

12.11.2 Accès aux différents champs d'un symbole

Au moyen d'opérandes de la machine LLM3.

```
(CVAL <accu>)      [opérande LLM3]
(CVALQ <symb>)     [opérande LLM3]
(PLIST <accu>)     [opérande LLM3]
(FVAL <accu>)      [opérande LLM3]
(OVAL <accu>)      [opérande LLM3]
(ALINK <accu>)     [opérande LLM3]
(PKGC <accu>)      [opérande LLM3]
(PNAME <accu>)     [opérande LLM3]
```

12.11.3 Les variables

Une variable est un symbole dont on peut changer la valeur (cf la fonction Le_Lisp VARIABLEP).

(BTVAR <op> <lab>) [instruction LLM3]

si l'opérande <op> est une variable, branchement à l'étiquette <lab>.

(BFVAR <op> <lab>) [instruction LLM3]

si l'opérande <op> n'est pas une variable branchement à l'étiquette <lab>.

12.12 Les Nombres

LLM3 utilise deux types de nombres :
- les nombres entiers (sur 16 bits)
- les nombres flottants (sur 31/32/48 ou 64 bits, dépendant de l'implantation).

12.12.1 Les nombres entiers sur 16 bits

12.12.1.1 Les tests de type

(BTFIX <op> <lab>) [instruction LLM3]

si l'opérande <op> est un nombre entier branchement à l'étiquette <lab>.

(BFFIX <op> <lab>) [instruction LLM3]

si l'opérande <op> n'est pas un nombre entier branchement à l'étiquette <lab>.

12.12.1.2 Les instructions de calcul

(INCR <op>) [instruction LLM3]

réalise le calcul entier : <op> + 1 -> <op>

(DECR <op>) [instruction LLM3]

réalise le calcul entier : <op> - 1 -> <op>

(PLUS <op1> <op2>) [instruction LLM3]

réalise le calcul entier : <op2> + <op1> -> <op2>

(DIFF <op1> <op2>) [instruction LLM3]

réalise le calcul entier : <op2> - <op1> -> <op2>

(NEGATE <op>) [instruction LLM3]

réalise le calcul entier : 0 - <op> -> <op>

(TIMES <op1> <op2>) [instruction LLM3]

réalise le calcul entier : <op2> * <op1> -> <op2>

(QUO <op1> <op2>) [instruction LLM3]

réalise le calcul entier : <op2> div <op1> -> <op2>

(REM <op1> <op2>) [instruction LLM3]

réalise le calcul entier : <op2> rem <op1> -> <op2>

12.12.1.3 Les comparaisons numériques entières

(CNBEQ <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique entière. Si l'opérande <op1> est égal à l'opérande <op2>, branchement à l'étiquette <lab>.

(CNBNE <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique entière. Si l'opérande <op1> n'est pas égal à l'opérande <op2>, branchement à l'étiquette <lab>.

(CNBLE <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique entière. Si l'opérande <op1> est plus petit ou égal à l'opérande <op2>, branchement à l'étiquette <lab>.

(CNBLT <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique entière. Si l'opérande <op1> est plus petit que l'opérande <op2>, branchement à l'étiquette <lab>.

(CNBGE <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique entière. Si l'opérande <op1> est plus grand ou égal à l'opérande <op2>, branchement à l'étiquette <lab>.

(CNBGT <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique entière. Si l'opérande <op1> est plus grand que l'opérande <op2>, branchement à l'étiquette <lab>.

12.12.1.4 Les instructions logiques

Les opérandes de ces intructions doivent toujours être des valeurs entières sur 16 bits. Le résultat de ces opérations est toujours sur 16 bits.

(LAND <op1> <op2>) [instruction LLM3]

réalise le calcul : <op2> and <op1> -> <op2>

(LOR <op1> <op2>) [instruction LLM3]

réalise le calcul : <op2> or <op1> -> <op2>

(LXOR <op1> <op2>) [instruction LLM3]

réalise le calcul : <op2> xor <op1> -> <op2>

(LSHIFT <op1> <op2>) [instruction LLM3]

décale l'opérande <op2> de <op1> positions et range le résultat dans l'opérande <op2>. Si l'opérande <op1> est positif, le décalage s'effectue à gauche (multiplication), s'il est négatif, il s'effectue à droite (division).

12.12.2 Les nombres flottants

12.12.2.1 Les tests de type nombre flottant

(BTFLOAT <op> <lab>) [instruction LLM3]

si l'opérande <op> est un nombre flottant branchement à l'étiquette <lab>.

(BFFLOAT <op> <lab>) [instruction LLM3]

si l'opérande <op> n'est pas un nombre flottant branchement à l'étiquette <lab>.

12.12.2.2 Les instructions de base

(FPLUS <op1> <op2>) [instruction LLM3]

réalise le calcul en flottant : <op2> + <op1> -> <op2>

(FDIFF <op1> <op2>) [instruction LLM3]

réalise le calcul en flottant : <op2> - <op1> -> <op2>

(FTIMES <op1> <op2>) [instruction LLM3]

réalise le calcul en flottant : <op2> * <op1> -> <op2>

(FQUO <op1> <op2>) [instruction LLM3]

réalise le calcul en flottant : <op2> / <op1> -> <op2>

12.12.2.3 Les comparaisons

(CFBEQ <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique flottante. Si l'opérande <op1> est égal à l'opérande <op2>, branchement à l'étiquette <lab>.

(CFBNE <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique flottante. Si l'opérande <op1> est différent de l'opérande <op2>, branchement à l'étiquette <lab>.

(CFBLT <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique flottante. Si l'opérande <op1> est plus petit que l'opérande <op2>, branchement à l'étiquette <lab>.

(CFBLE <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique flottante. Si l'opérande <op1> est plus petit ou égal à l'opérande <op2>, branchement à l'étiquette <lab>.

(CFBGT <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique flottante. Si l'opérande <op1> est plus grand que l'opérande <op2>, branchement à l'étiquette <lab>.

(CFBGE <op1> <op2> <lab>) [instruction LLM3]

réalise une comparaison numérique flottante. Si l'opérande <op1> est plus grand ou égal à l'opérande <op2>, branchement à l'étiquette <lab>.

12.13 Les Vecteurs de Pointeurs Lisp

`(VAL <accu>)` `[opérande LLM3]`

donne accès au pointeur sur le tas d'un vecteur. ATTENTION : cette valeur ne peut pas résider dans un registre sous peine de perturber le GC.

`(TYP <accu>)` `[opérande LLM3]`

donne accès au type du vecteur.

12.13.1 Test de type vecteur de pointeur

(BTVECT <op> <lab>) [instruction LLM3]

si <op> est un vecteur de pointeurs, branchement à l'étiquette <lab>.

(BFVECT <op> <lab>) [instruction LLM3]

si <op> n'est pas un vecteur de pointeurs, branchement à l'étiquette <lab>.

12.13.2 Création

il n'existe pas d'instruction spéciale; il faut appeler explicitement la fonction Le_Lisp MAKEVECTOR.

12.13.3 Accès aux éléments d'un vecteur de pointeurs

(HPXMOV <vect> <n> <op>) [instruction LLM3]

transfère le <n>ième élément du vecteur Lisp <vect> dans l'opérande <op>.

(HPMOVX <op> <vect> <n>) [instruction LLM3]

transfère l'opérande <op> dans le <n>ième élément du vecteur Lisp <vect>.

12.14 Les Chaînes de Caractères

`(VAL <accu>)` `[opérande LLM3]`

donne accès au pointeur sur le tas d'une chaîne. ATTENTION : cette valeur ne peut pas résider dans un registre sous peine de perturber le GC.

`(TYP <accu>)` `[opérande LLM3]`

donne accès au type de la chaîne.

12.14.1 Test de type chaîne de caractères

(BTSTRG <op> <lab>) [instruction LLM3]

si <op> est une chaîne de caractères, branchement à l'étiquette <lab>.

(BFSTRG <op> <lab>) [instruction LLM3]

si <op> n'est pas une chaîne de caractères, branchement à l'étiquette <lab>.

12.14.2 Création

il n'existe pas d'instruction spéciale; il faut appeler explicitement la fonction Le_Lisp MAKESTRING.

12.14.3 Accès aux caractères

(HBXMOV <strg> <index> <op>) [instruction LLM3]

transfère le <index>ième caractère de la chaîne Lisp <strg> dans l'opérande <op>.

(HBMOVX <op> <strg> <index>) [instruction LLM3]

transfère l'opérande <op> dans le <index>ième caractère de la chaîne Lisp <strg>.

12.15 Zone du tas (HEAP)

(HGSIZE <op1> <op2>) [instruction LLM3]

retourne la taille de l'objet <op1> (chaîne de caractères ou vecteurs de pointeurs) dans l'opérande <op2>.

12.16 Les Extensions du Chargeur/Assembleur

Il est possible d'étendre la syntaxe du chargeur/assembleur LLM3 en définissant des fonctions dans des packages spéciaux.

LD-CODOP [Constante]

nom du package dans lequel doivent se trouver les fonctions associées aux codes opérations étendus.

LD-DIR [Constante]

nom du package dans lequel doivent se trouver les fonctions associées aux opérandes directs étendus.

LD-IND [Constante]

nom du package dans lequel doivent se trouver les fonctions associées aux opérandes indirects étendus.

12.17 Les Fonctions

12.17.1 Les types des fonctions

Les fonctions écrites en LLM3 sont de 4 types :
- les SUBR qui évaluent leurs arguments
- les FSUBR qui ne les évaluent pas
- les MSUBR qui ne les évaluent pas
- les DMSUBR qui ne les évaluent pas

12.17.2 Règle d'appel des fonctions

pour les SUBR à 0,1,2 ou 3 arguments (SUBR0, SUBR1, SUBR2, SUBR3), les arguments sont passés respectivement dans les accumulateurs : A1, A2, A3. Pour les SUBR à plus de 3 arguments ou à nombre variable d'arguments (NSUBR), les arguments sont tous empilés et le nombre d'arguments total est fourni dans A4. Pour les FSUBR, la liste des arguments non évalués (i.e. le CDR de l'appel) se trouve dans A1. Pour les MSUBR, la liste des arguments non évalués (i.e. la forme d'appel elle-même) se trouve dans A1. Pour les DMSUBR, la liste des arguments non évalués (i.e. le CDR de l'appel) se trouve dans A1.

Toutes les fonctions retournent une valeur dans A1.

12.18 Exemples

Cette section montre de petits exemples de fonctions écrites en LLM3. Il existe un fichier de test dans le catalogue test de nom TESTLAP qui contient l'ensemble des tests du chargeur. Il est également possible de visualiser le code LLM3 engendré par les différents compilateurs (voir le chapitre suivant).

```
? ; Ces exemples ont été produits sur machine VAX
? ;
? ; traduction en LAP de la fonction :
? ;
? ; (de dlq (a l)
? ;     (cond ((not (consp l)) ())
? ;           ((eq a (car l)) (dlq a (cdr l)))
? ;           (t (cons (car l) (dlq a (cdr l))))))
?
? (defvar llap1 '(
?        (fentry dlq subr2)
?        (btcons a2 1001)
?        (mov nil a1)
?        (return)
?     1001
?        (cabne a1 (car a2) 1003)
?        (mov (cdr a2) a2)
?        (bra dlq)
?     1003
?        (push (car a2))
```

```
?       (mov (cdr a2) a2)
?       (call dlq)
?       (mov a1 a2)
?       (pop a1)
?       (jmp cons))
= llap1
?
? (loader llap1 t)
    (fentry dlq subr2)          00043E29    01 01 01
    (btcons a2 1001)            00043E2C    D1 5A 53 15 00
    (mov nil a1)                00043E31    D0 58 52
    (return)                    00043E34    05
1001                            00043E35
    (cabne a1 (car a2) 1003)    00043E35    D1 52 63 12 00
    (mov (cdr a2) a2)           00043E3A    D0 A3 04 53
    (bra dlq)                   00043E3E    11 EC
1003                            00043E40
    (push (car a2))             00043E40    DD 63
    (mov (cdr a2) a2)           00043E42    D0 A3 04 53
    (call dlq)                  00043E46    10 E4
    (mov a1 a2)                 00043E48    D0 52 53
    (pop a1)                    00043E4B    D0 8E 52
    (jmp cons)                  00043E4E    17 D8 C8 3A
    (endl)                      00043E52
= ()
?
? (dlq 'a '(b a c a b))
= (b c b)
?
? ; Cet exemple montre la manipulation des chaînes de caractères Lisp
?
? (defvar llap2 '(
?       (fentry screat subr2)
?       (push a1)
?       (push a2)
?       (mov '6 a1)
?       (mov '#/X a2)
?       (jcall makestring)
?       (hbmovx '#/a a1 (& 1))
?       (hbmovx '#/b a1 (& 0))
?       (adjstk '2)
?       (return)))
= llap2
?
? (loader llap2 t)
    (fentry screat subr2)       00043E52    01 01
    (push a1)                   00043E54    DD 52
    (push a2)                   00043E56    DD 53
    (mov '6 a1)                 00043E58    D0 06 52
    (mov '88 a2)                00043E5B    D0 8F 58 00 00 00 53
    (jcall makestring)          00043E62    16 D8 68 50
    (hbmovx '97 a1 (& 1))       00043E66    D0 62 51 D0 AE 04 50 C0 08 50 C0
                                00043E71    50 51 D0 8F 61 00 00 00 50 90 50
```

```
                                00043E7C   61
    (hbmovx '98 a1 (& 0))       00043E7D   D0 62 51 D0 6E 50 C0 08 50 C0 50
                                00043E88   51 D0 8F 62 00 00 00 50 90 50 61
    (adjstk '2)                 00043E93   C0 08 5E
    (return)                    00043E96   05
    (endl)                      00043E97
= ()
?
? (screat 2 4)
= "XXaXbX"
?
? ; Ce dernier exemple montre la manipulation des vecteurs
?
? (defvar llap3 '(
?       (fentry listovect nsubr)
?       (push a4)
?       (mov a4 a1)
?       (mov nil a2)
?       (jcall makevector)
?       (pop a4)
?       (bra 1005)
?     1002
?       (pop a2)
?       (hpmovx a2 a1 a4)
?     1005
?       (sobgez a4 1002)
?       (return)))
= llap3
?
? (loader llap3 t)
    (fentry listovect nsubr)    00043E97   01
    (push a4)                   00043E98   DD 55
    (mov a4 a1)                 00043E9A   D0 55 52
    (mov nil a2)                00043E9D   D0 58 53
    (jcall makevector)          00043EA0   16 D8 C8 52
    (pop a4)                    00043EA4   D0 8E 55
    (bra 1005)                  00043EA7   11 00
1002                            00043EA9
    (pop a2)                    00043EA9   D0 8E 53
    (hpmovx a2 a1 a4)           00043EAC   D0 62 51 9C 02 55 50 C0 08 50 C0
                                00043EB7   50 51 D0 53 61
1005                            00043EBC
    (sobgez a4 1002)            00043EBC   B7 55 18 E9
    (return)                    00043EC0   05
    (endl)                      00043EC1
= ()
?
? (listovect 1 3 5 7 2 4 6 8)
= #[1 3 5 7 2 4 6 8]
```

CHAPITRE 13

Les Compilateurs et les Modules

Bernard Serpette

Ce chapitre décrit les deux compilateurs présents dans la version 15.2 : le compilateur standard et le compilateur modulaire *Complice*. Ces deux compilateurs possèdent les mêmes fonctions d'appel et sont donc exclusifs l'un de l'autre.

Le compilateur standard assure la *totale compatibilité* avec l'interprète en particulier il fabrique toujours les mêmes blocs d'activation dynamiques et lexicaux que l'interprète. Ses seules limitations sont l'impossibilité de traiter des programmes auto-modifiants bien évidemment, de ne pas provoquer l'erreur ERRUDV en cas de variable indéfinie et de ne pas tenir compte de l'aspect totalement dynamique de la forme FLET. Sa taille est réduite, ses performances sont moyennes (entre 3 et 6 fois plus rapide que l'interprète) mais il permet d'enlever les fonctions de la zone liste pour les transférer dans la zone code avec un gain en place (dépendant du jeu d'instructions des machines hôtes).

Le compilateur modulaire Complice engendre un code nettement plus performant (en temps mais pas nécessairement en place!) et permet surtout la compilation séparée. Le code engendré est de 2 à 5 fois plus rapide qu'avec le compilateur standard mais nécessite certaines précautions décrites dans une section suivante. Tout en assurant la compatibilité avec l'interprète, Complice essaie, dans la mesure du possible, d'utiliser un schéma lexical pour lier les variables des fonctions. Ce trait permet d'utiliser Complice pour vérifier si un programme Le_Lisp peut être compiler par un compilateur lexical, tel que sera ISO LISP.

Ces deux compilateurs utilisent le même chargeur mémoire (décrit au chapitre précedent).

13.1 Les Fonctions d'Appel des Compilateurs

COMPILER **[Feature]**

ce trait indique que l'un des compilateurs est présent en mémoire.

COMPLICE **[Feature]**

ce trait indique que le compilateur Complice est présent en mémoire.

(COMPILER <l1> <l2> <i1> <i2>) [SUBR à 4 arguments]

compile la fonction, ou liste de fonctions, <l1>. <i1> est un indicateur qui, s'il est positionné, fera imprimer dans le flux de sortie courant la liste des instructions LLM3 engendrées par le compilateur. <i2> est un indicateur qui sera passé au chargeur. <l2> indique le statut des fonctions utilisées à l'intérieur des fonctions de <l1>. Si <l2> est une liste, elle représentera la liste des fonctions à ne pas compiler durant la compilation des fonctions de <l1>. Si <l2> n'est pas une liste toutes les fonctions utilisées à l'intérieur des fonctions de <l1> seront compilées. Dans le compilateur standard si <l2> vaut () aucune des fonctions appelées par les fonctions de <l1> ne seront compilées.
Ceci n'est pas réalisable par le compilateur Complice. Si l'une des fonctions de <l2> est une SUBR, le compilateur Complice, engendrera un appel à l'évaluateur lors la compilation d'un appel de cette fonction. Le compilateur standard ignore les SUBR dans la liste <l2>.

(COMPILE <l1> <l2> <i1> <i2>) [FSUBR]

est la forme FSUBRée de la fonction précédente, c'est-à-dire qu'aucun argument n'est évalué. De plus les arguments <l2>, <i1> et <i2> sont optionnels. Ils prennent la valeur () par défaut.

(COMPILE-ALL-IN-CORE <i1> <i2>) [SUBR à 0, 1 ou 2 arguments]

compile toutes les fonctions de l'OBLIST qui sont de type EXPR, FEXPR, MACRO et DMACRO. Les arguments <i1> et <i2> sont identiques à ceux de la fonction COMPILER.

COMPILE-ALL-IN-CORE peut être défini en Lisp de la manière suivante :

```
(DE COMPILE-ALL-IN-CORE i
    (LET ((i1 (CAR i)) (i2 (CADR i)))
         ; d'abord les EXPR/FEXPR (si les macros les utilisent)
         (COMPILER (MAPLOBLIST (LAMBDA (x)
                      (AND (NULL (GETPROP x 'DONT-COMPILE))
                           (MEMQ (OR (CAR (GETPROP x 'RESETFN))
                                     (TYPEFN x))
                                 '(EXPR FEXPR)))))
                   T
                   i1
                   i2)
         ; puis toutes les macros sans risque.
         (COMPILER (MAPLOBLIST (LAMBDA (x)
                      (AND (NULL (GETPROP x 'DONT-COMPILE))
                           (MEMQ (OR (CAR (GETPROP x 'RESETFN))
                                     (TYPEFN x))
                                   '(MACRO DMACRO)))))
                   T
                   i1
                   i2)))
```

(COMPILEFILES <lfi> <fo>) [SUBR à 2 arguments]

compile toutes les fonctions qui se trouvent dans le fichier, ou la liste de fichiers, <lfi> et range le résultat de cette compilation dans le fichier de nom <fo>. L'extension #:SYSTEM:LELISP-EXTENSION est automatiquement ajoutée aux noms de fichiers <lfi> qui ne la possèdent pas. Le fichier <fo> peut être chargé sous Lisp au moyen des fonctions classiques : LOAD, LOADFILE, ^L .. etc.
Pour le compilateur modulaire Complice l'argument <fo> est optionnel. Il est par défaut identique au premier élément de la liste <lfi>. De plus Complice rajoutera l'extension #:SYSTEM:OBJ-EXTENSION au nom de fichier <fo> si celui-ci ne la possède pas. Le fichier devra alors être chargé par la fonction LOADOBJECTFILE.

(PRECOMPILE <e1> <l> <e2> <op>) [FSUBR]

sous interprète, PRECOMPILE retourne la valeur de l'évaluation de <e1> en ignorant les 2 autres arguments. Le compilateur quant à lui va supposer que la liste <l> est le résultat de la compilation de <e1> et va charger les instructions LLM3 de la liste <l>. Après ce chargement le compilateur va exécuter l'expression <e2> *dans l'environnement du compilateur.* <op> est l'opérande LLM3 qui contiendra le résultat (par défaut A1).
Cette fonction permet d'interfacer facilement des précompilateurs au compilateur existant.

(DONT-COMPILE <s1> ... <sN>) [FSUBR]

indique au compilateur de ne jamais compiler les fonctions de nom <s1> ... <sN>. Pour forcer la compilation il faut utiliser la fonction COMPILE explicitement.

13.2 Les Macros du Compilateur

Avant compilation, les formes Lisp subissent une macro-expansion qui utilise non seulement les macros utilisateurs (fonctions de type MACRO, DMACRO, MSUBR ou DMSUBR) mais aussi deux nouveaux types de macro-fonctions propres au compilateur : les macros ouvertes et les macros fermées.

Certaines fonctions du système possèdent des définitions de macros fermées ou de macros ouvertes. Les fonctions possédant une définition de macro fermée ne doivent pas être redéfinies. Les définitions des macros du compilateurs sont données dans le fichier CPMAC de la bibliothèque standard.

13.2.1 Les macros fermées

Les macros fermées sont toujours expansées pendant la phase de compilation. Cette macro expansion permet d'engendrer un code extrêmement efficace pour certaines fonctions définies sous forme de SUBR ou de FSUBR pour des raisons d'efficacité de l'interprète. La fonction MAPC est par exemple une SUBRN dans l'interprète et une macro fermée du compilateur, qui engendre une simple boucle.

Voici la liste des macros fermées standard du compilateur.

```
ANY          CATCHERROR  COND          DECR          DEFPROP
```

DESETQ	ERR	ERRSET	EVERY	IFN
INCR	LETVQ	LOGNOT	LOOP	MAP
MAPC	MAPCAN	MAPCAR	MAPCON	MAPL
MAPLIST	NEQ	NEQUAL	NEWL	NEWR
NEXTL	NULL	PROG2	PSETQ	RETURN
SETQQ	UNLESS	UNTILEXIT	WHEN	

13.2.2 Les macros ouvertes

Les macros ouvertes ne sont expansées durant la phase de compilation que lorsque l'indicateur #:COMPILER:OPENP est positionné.
En standard les combinaisons des primitives CAR et CDR sont des macros ouvertes du compilateur. L'utilisateur peut donner des définitions de macros ouvertes à toutes les fonctions qu'il désire, sauf à celles qui possèdent une définition de macro fermée. Il lui appartient de vérifier la cohérence de la définition de la fonction et de sa définition sous forme de macro ouverte, afin de préserver la compatibilité entre le code interprété et le code compilé.

#:COMPILER:OPEN-P [Variable]

indique si les macros ouvertes du compilateur doivent être expansées. Au chargement des compilateurs ces variables sont toujours positionnées à T. Spécifie aussi si l'on veut compiler efficacement les accès aux divers champs des objets Le_Lisp et les primitives de bas niveau, dans ce cas les tests de type ne sont plus effectués.
Ceci concerne les fonctions suivantes :

ADD	ADD1	CAR	CDR	COMMENT
DIV	FADD	FDIV	FMUL	FSUB
LOGAND	LOGOR	LOGSHIFT	LOGXOR	MUL
OBJVAL	PACKAGECELL	PLIST	REM	RPLACA
RPLACD	SET	SLEN	SREF	SSET
SUB	SUB1	SYMEVAL	VLENGTH	VREF
VSET				

Le code engendré est bien évidemment plus dense, et plus rapide, mais ne réalise plus les tests de l'interprète (par exemple, un appel à la fonction CAR ne vérifie plus si son argument est bien une liste). Il est donc possible, en cas de programme incorrect, de provoquer des erreurs de la machine.

```
ex : ? (DE TEST1 (L)
     ?     (CONS (CAR L) (CDR L)))
     = TEST1
     ? (DE TEST2 (L)
     ?     (CONS (CAR L) (CDR L)))
     = TEST2
     ?
     ? (DEFVAR #:COMPILER:OPEN-P T)
     = #:COMPILER:OPEN-P
     ? ; Expansion directe des accès aux champs CAR et CDR
     ? (COMPILE TEST1 T T)
               fentry    test1,subr1
               mov       cdr(a1),a2
               mov       car(a1),a1
```

```
            jmp         cons
= (TEST1)
?
? ; Appel des fonctions CAR et CDR
? (DEFVAR #:COMPILER:OPEN-P ())
= #:COMPILER:OPEN-P
? (COMPILE TEST2 T T)
            fentry      test2,subr1
            push        a1
            jcall       car
            push        a1
            mov         &1,a1
            jcall       cdr
            mov         a1,a2
            pop         a1
            adjstk      '1
            jmp         cons
= (TEST2)
```

(DEFMACRO-OPEN <nom> . <fval>) [FSUBR]

donne une définition de macro ouverte à la fonction <nom>. Les arguments de cette fonction sont exactement les même que ceux de la fonction DEFMACRO.

(MAKE-MACRO-OPEN <nom> <fval>) [SUBR à 2 arguments]

est la version SUBRée de la fonction précédente. Elle est particulièrement utile pour engendrer des macros ouvertes par programme.

(MACRO-OPENP <nom>) [SUBR à 1 argument]

retourne une fonction ayant la définition de macro ouverte associée à la fonction de nom <nom>. Retourne () si cette fonction ne possède pas de définition de macro ouverte. On peut réaliser l'expansion de la macro ouverte associée à une fonction de la manière suivante :

```
ex : ? (DE EXPAND-OPEN (call)
     ?     (LET ((open-macro (MACRO-OPENP (CAR call))))
     ?          (WHEN open-macro
     ?               (APPLY open-macro (CDR call)))))
     = EXPAND-OPEN
     ? (DE DEUZE (v)
     ?     (VREF v 2))
     = DEUZE
     ? (DEFMACRO-OPEN DEUZE (v)
     ?    `(VREF ,v 2))
     = DEUZE
     ? (EXPAND-OPEN '(DEUZE #[1 2 3]))
     = (VREF #[1 2 3] 2)
```

(REMOVE-MACRO-OPEN <nom>) [SUBR à 1 argument]

enlève la définition de macro ouverte de la fonction de nom <nom>.

13.3 Les Modules

Un module est un atome Lisp référant un fichier de description, d'extension #:SYSTEM:MOD-EXTENSION, dans le système de catalogues #:SYSTEM:PATH. Un module décrit un ensemble de fichiers contenant du code Le_Lisp et ses interactions avec d'autres modules Le_Lisp.

13.3.1 Format des fichiers de description

Le fichier de description de module doit être écrit par l'utilisateur selon la syntaxe suivante :

```
DEFMODULE <#:sys-package:colon>
     <clef1>          <valeur1>
     ...                   ...
     <clefN>          <valeurN>
```

<#:SYS-PACKAGE:COLON> indique le package dans lequel doivent être lus les symboles du fichier de description de module qui commencent par le caractère ":".

```
ex :  DEFMODULE pretty
        FILES          (pretty)
        EXPORT     (pretty prettyf pprint pprin prettyend)
```

Ces fichiers de description seront utilisés voire complétés par des utilitaires tels que chargeurs, mises au point, documentation et compilateurs.

13.3.2 Les fonctions sur les modules

(PROBEPATHM <module>) [SUBR à 1 argument]

retourne, dans le sytème de catalogue #:SYSTEM:PATH, le nom complet du fichier de description du module <module>, retourne () si ce module n'existe pas.

PROBEPATHM peut être défini en Lisp de la manière suivante :

```
(DE PROBEPATHM (mod)
    (SEARCH-IN-PATH
        #:SYSTEM:PATH
        (CATENATE mod #:SYSTEM:MOD-EXTENSION)))
ex : ? (PROBEPATHM 'pretty)   -> /usr/local/lelispv15.2/llmod/pretty.lm
```

(READDEFMODULE <module>) [SUBR à 1 argument]

retourne une structure Lisp contenant les informations du fichier de description du module <module>. Cette structure sera appelée par la suite une définition de module.

(GETDEFMODULE <defmod> <clef>) [SUBR à 2 arguments]

retourne la valeur définie sous la clef <clef> dans la définition de module <defmod>.

(SETDEFMODULE <defmod> <clef> <valeur>) [SUBR à 3 arguments]

rajoute la valeur <valeur> définie par la clef <clef> dans la définition de module <defmod>. Si la clef existe déja l'ancienne valeur est écrasée.

(PRINTDEFMODULE <defmod> <file>) [SUBR à 2 arguments]

imprime dans le fichier <file> une description de module représentée par la définition <defmod>.

13.3.3 Clefs utilisées par le système

L'interprète utilise les clefs **FILES** (liste des fichiers sources) et **IMPORT** (liste des modules importés) lors du chargement d'un module par la fonction LOADMODULE.

Le compilateur Complice utilise les clefs **FILES**, **IMPORT**, **EXPORT** (liste des fonctions exportées du module), **CPEXPORT** (description interne des fonctions exportées) et **CPIMPORT** (description interne des fonctions définies mais non exportées).

Il appartient à l'utilisateur de définir les clefs FILES, IMPORT et EXPORT d'un module. Si la clef EXPORT n'est pas définie, le module ne sera jamais compilé par la fonction COMPILEMODULE. Les autres clefs sont maintenues à jour automatiquement par le compilateur Complice.

13.3.4 Chargement et compilation des modules

(LOADMODULE <module> <indicateur>) [SUBR à 1 ou 2 arguments]

Charge le module <module> et tous les modules importés par ce module.
Le chargement des modules importés est réalisé en premier, par la même fonction LOADMODULE; on charge donc récursivement les modules importés par les modules importés. Seul les modules qui n'ont pas été déjà chargés dans le système sont chargés. L'indicateur permet cependant de forcer le (re)chargement de tous les modules importés.
Si le module n'a pas été compilé, LOADMODULE charge ensuite les fichiers définis sous la clef FILE, par la fonction LIBLOADFILE. Si le module a été compilé (par la fonction COMPILEMODULE) LOADMODULE charge le fichier de nom <module> par la fonction LOADOBJECTFILE. On charge ainsi le code compilé des fichiers définis sous la clef FILE.

Les deux variables suivantes sont mises à jour par la fonction LOADMODULE.

#:MODULE:INTERPRETED-LIST [Variable]

contient l'ensemble des modules dont les fichiers ont été chargés par la fonction LIBLOADFILE.

#:MODULE:COMPILED-LIST [Variable]

contient l'ensemble des modules dont les fichiers ont été chargés par la fonction LOADOBJECTFILE.

(COMPILEMODULE <module> <ind>) [SUBR à 1 ou 2 arguments]

compile le module <module> avec le compilateur modulaire Complice. Le résultat de la compilation est stocké dans le fichier de nom <module> et d'extension #:SYSTEM:OBJ-EXTENSION, qui est éventuellement créé s'il n'existait pas. Si l'indicateur <ind> est positionné, les modules importés par <module> sont compilés récursivement au moyen de la même fonction COMPILEMODULE.
Le fichier résultat de la compilation peut être chargé au moyen de la fonction LOADOBJECTFILE. Le module compilé, et ses modules importés peuvent être chargés par la fonction LOADMODULE.

Lorsque l'indicateur n'est pas fourni la définition de COMPILEMODULE se rapproche de :

```
(DE COMPILEMODULE (mod)
    (COMPILEFILES
        (GETDEFMODULE (READDEFMODULE mod) 'FILES)
        mod))
```

mais le fichier de description du module est mis à jour pour refléter la compilation du module (clefs EXPORT, CPEXPORT et CPIMPORT). De plus les fonctions non exportées du module n'auront plus d'existence après la compilation. Les droits d'accès en écriture sur le fichier de description de module sont nécessaires pour pouvoir effectuer la compilation.

13.3.5 Exemples

Soit le fichier **f1**:

```
(DE EXEMPLE1 (l) (TMP1 l))
(DE TMP1 (l) (CONS (CAR l) (CDR l)))
```

et le fichier **f2**:

```
(DE EXEMPLE2 (l) (TMP2 l))
(DE TMP2 (l) (CONS (CAR l) (EXEMPLE1 l)))
```

Les fichiers de descriptions de module sont pour **f1** le fichier **m1**:

```
DEFMODULE USER
        FILES  (F1)
        EXPORT (EXEMPLE1)
```

et pour **f2** le fichier **m2**:

```
DEFMODULE USER
        FILES  (F2)
        EXPORT (EXEMPLE2)
        IMPORT (M1)
```

Exemple d'utilisation de modules sous forme interprétée et exemple de compilation de modules :

```
? (LOADMODULE 'm2 t)
= m2
? (PRETTY EXEMPLE1)
(DE EXEMPLE1 (l) (TMP2 l))
= EXEMPLE1
? (COMPILEMODULE 'm2 t)
```

```
= ()
? #:MODULE:INTERPRETED-LIST
= (M1 M2 ...)
```

Exemple d'utilisation de modules sous forme compilée :

```
? (LOADMODULE 'm2 t)
= m2
? (PRETTY EXEMPLE1)
(DS EXEMPLE1 SUBR1 10440)
= ()
? (PRETTY TMP2)
()
= ()
? (EXEMPLE2 '(1 2 3))
= (1 1 2 3)
```

13.4 Les Fichiers Objets

Le compilateur Complice, qui est utilisé pour réaliser la compilation des modules, construit des fichiers objets, c'est-à-dire des fichiers contenant du code LLM3 sous format LAP. Ces fichiers ont toujours l'extension #:SYSTEM:OBJ-EXTENSION.

#:SYSTEM:OBJ-EXTENSION [Variable]

contient l'extension par défaut des fichiers objets Le_Lisp.

(PROBEPATHO <file>) [SUBR à 1 argument]

retourne, dans le système de catalogue #:SYSTEM:PATH, le nom complet du fichier objet <file>. L'extension #:SYSTEM:OBJ-EXTENSION est éventuellement rajoutée au nom <file> si elle est absente. Retourne () si le fichier ne peut pas être trouvé.

PROBEPATHO peut être défini en Lisp de la manière suivante :

```
(DE PROBEPATHO (mod)
    (SEARCH-IN-PATH
        #:SYSTEM:PATH
        (CATENATE mod #:SYSTEM:OBJ-EXTENSION)))
```

```
ex : ? (PROBEPATHO 'pretty)
     = /usr/local/lelispv15.2/llobj/pretty.lo
```

(LOADOBJECTFILE <file>) [SUBR à 1 argument]

charge dans le système de catalogue #:SYSTEM:PATH le fichier objet <file>.

LOADOBJECTFILE peut être défini en Lisp de la manière suivante :

```
(DE LOADOBJECTFILE (file)
    (LET ((real-file (PROBEPATHO file)))
         (IFN real-file
              (ERROR 'LOADOBJECTFILE "fichier inconnu" file)
              (LOADFILE real-file t)))
```

13.5 Complice

Le compilateur Complice est toujours utilisé pour réaliser la compilation des modules. Cette section décrit les traits spécifiques de Complice, ses messages d'erreur et ses avertissements.

13.5.1 Indicateurs de compatibilité

Complice compile lexicalement, dans la mesure de ses moyens, les accès aux variables; dans ce cas les variables n'ont plus d'existence propre en dehors de la portée du bloc où est déclarée la variable. Ceci peut s'avérer incompatible avec le comportement qu'a un programme exécuté en mode interprété. Pour pallier ces problèmes Complice utilise un indicateur #:COMPLICE:PARANO-FLAG pour, si il est positionné et dans le cas d'appels explicites à l'évaluateur, compiler les accès aux variables de manière dynamique (c.à.d. de la même manière que l'évaluateur).

#:COMPLICE:PARANO-FLAG [Variable]

indique si l'on veut rester compatible dans tous les cas de figures avec l'interprète. Au chargement de Complice #:COMPLICE:PARANO-FLAG vaut T.

```
ex : (DE TEST1 (l) (FUNCALL 'TEST2))
     (DE TEST2 () l)

     (TEST1 'local)                    -> local
     (COMPILE TEST1)
     (TEST1 'local)                    -> local

     (de TEST1 (l) (FUNCALL 'TEST2))))
     (DEFVAR #:COMPLICE:PARANO-FLAG ())
     (COMPILE TEST1 T T)
               fentry    test1,subr1
               push      @test2
               mov       '1,a4
               jmp       funcall
     (LET ((l 'global))
        (TEST1 'local))                -> global
```

Les fonctions modifiant le comportement du compilateur en fonction de la valeur de #:COMPLICE:PARANO-FLAG sont:

```
APPLY      DESET      EVAL       EVLIS      FLAMBDA
FLET       FUNCALL    LAMBDA     LOCK       SET
SYMEVAL    TIME       UNEXIT     UNWIND
```

13.5.2 Erreurs et avertissements

Les erreurs et avertissements de Complice sont édités sur le canal de sortie courant sous l'un des deux formats :

```
E.<n>.<fnt>..<msg>:          pour une erreur
       <arg>

W.<n>.<fnt>..<msg>:          pour un avertissement
     <arg>
```

Où

<n> est le numéro de l'erreur ou de l'avertissement,
<fnt> est la fonction en cours de compilation,
<msg> est un message expliquant l'erreur en clair,
<arg> est l'argument faisant défaut.

Dans le cas d'une erreur la fonction <fnt> n'est pas compilée. Dans le cas d'un avertissement le code produit peut être incompatible avec l'interprète (voir cependant l'indicateur #:COMPLICE:PARANO-FLAG).

#:COMPLICE:WARNING-FLAG [Indicateur]

permet, si cette variable n'est pas positionnée, de supprimer l'impression des avertissements. Au chargement de Complice #:COMPLICE:WARNING-FLAG vaut T.

#:COMPLICE:NO-WARNING [Variable]

contient la liste des numéros d'avertissement que l'on ne désire pas voir afficher à l'écran. Au chargement de Complice #:COMPLICE:NO-WARNING contient la liste (7 8).

E.0 [Message d'erreur de Complice]

E.0.<fnt>..Erreur interne:
(<f> <m> <b>)

CAUSE : Pour des raisons d'efficacité (ou de cas non prévu) Complice n'analyse pas systématiquement la bonne contruction du code à compiler, par exemple (add1 . 2), dans ce cas on aboutit généralement à une erreur de la machine (<m> vaut errmac).

ARGUMENTS : <f>, <m>, <b> sont les arguments du traitement d'erreur standard (voir la fonction syserror).

REMEDE : Vérifier que la fonction <fnt> fonctionne en mode interprété, vérifier la cohérence du code Lisp de <fnt>.

E.1 [Message d'erreur de Complice]

E.1.<fnt>..Fonction calculée:
<expression-Lisp>

CAUSE : Cette erreur est produite lorsque le premier élément d'une forme évaluable n'est ni un symbole ni une lambda-expression, par exemple ((if x '1+ '1-) 8).

ARGUMENT : Pour l'exemple précédent l'<expression-Lisp> est (if x '1+ '1-).

REMEDE : Mettre (funcall (if x '1+ '1-) 8).

E.2 [Message d'erreur de Complice]

E.2.<fnt>..Application d'une mlambda:
<lambda-expression>

CAUSE : Complice ne sait pas compiler une forme telle que :
((mlambda (l x) `(ncons ,x)) x).

ARGUMENT : Pour l'exemple précédent (mlambda (l x) `(ncons ,x)).

REMEDE : Si la macrogénération peut être faite durant la compilation utilisez une macro-fonction intermediaire.

E.3 [Message d'erreur de Complice]

E.3.<fnt>..Erreur durant macroexpansion:
<macro-fonction>

CAUSE : Toutes les macroexpansions (MACRO et DMACRO) sont faites dans l'environnement du compilateur, les macro-fonctions utilisant l'environnement dynamique de votre programme provoquent cette erreur.
Par exemple :

```
(DE F (a)
    (BCLOS a (+ a a)))

(DMD BCLOS (v . pg)
     `(LET ((,v ',(SYMEVAL v)))
           ,@pg))
```

ARGUMENT : Pour l'exemple précédent: bclos.

REMEDE : Si la macrogénération ne doit être faite qu'une seule fois dans l'environnement de votre programme, faites ces macrogénérations avant la compilation avec un jeu d'essai.

E.4 [Message d'erreur de Complice]

E.4.<fnt>..Fonction non définie:
<fonction>

CAUSE : La fonction <fnt> appelle une fonction qui n'est pas définie au moment de la compilation. Par exemple :

```
(DE f () (G))
```

ARGUMENT : Pour l'exemple précédent : g.

REMEDE : Si la fonction incriminée ne provient pas d'une faute de frappe, effectuez l'appel au moyen de funcall : (de f () (funcall 'g))

E.5 [Message d'erreur de Complice]

E.5.<fnt>..Ne peut pas compiler un flet:
<expression-lisp>

CAUSE : Lorsque l'indicateur #:COMPLICE:PARANO-FLAG n'est pas positionné Complice refuse de compiler une forme commençant par (flet ...

REMEDE : Soit utilisez le système d'interruption programmable (#:SYSTEM:ITSOFT) si vous désirez modifier temporairement le comportement de fonctions telles que SYSERROR, BOL, EOL, etc .. Soit positionnez l'indicateur #:COMPLICE:PARANO-FLAG.

E.7 [Message d'erreur de Complice]

E.7.<fnt>..Ne peut pas compiler un letv:
<expression-lisp>

CAUSE : Complice ne sait compiler les constructions dynamiques d'environnement que lorsque l'arbre des variables est explicitement fourni. Par exemple :

```
(DE goodf (vals) (LETV '(a b) vals (+ a b)))
(DE badf (env vals) (LETV env vals (+ a b)))
```

ARGUMENT : Pour l'exemple précédent : env.

REMEDE : Aucun.

W.0 [Message d'avertissement de Complice]

W.0.<fnt>..Variable globale non déclarée:
<variable>

CAUSE : Dans la fermeture transitive de <fnt> la variable <variable> peut être utilisée sans avoir été instanciée. Par exemple:

```
(DE f () xyz)
```

ou

```
(DE last (l) (IFN l xyz (LET ((xyz l)) (LAST (cdr l)))
```

ARGUMENT : Pour les exemples précédents : xyz

REMEDE : Si ce n'est ni une erreur de frappe ni une erreur logique, donnez une valeur globale à votre variable au moyen de la fonction DEFVAR. L'appel à DEFVAR doit se trouver nécessairement dans un fichier.

W.2 [Message d'avertissement de Complice]

W.2.<module>..Fonction non utilisée:
<fonction>

CAUSE : Cet avertissement ne se produit qu'avec la fonction COMPILEMODULE. Il apparaît si la fonction <fonction> est créée dans le module <module> mais n'est utilisée par aucune des fonctions exportées.

REMEDE : Si vous avez besoin de cette fonction la rajouter dans la description du module <module> sous la clef XPORT.

W.3 [Message d'avertissement de Complice]

W.3.<fnt>..Fonction de mauvais type:
(<fonction> <type>)

CAUSE : Si <fnt> est le même symbole que <fonction> c'est que vous demandez de compiler une fonction non compilable, par exemple (compile car),

sinon c'est que <fonction> est indéfinie (<type> vaut ()) et il y aura appel explicite à l'évaluateur.

REMEDE : Utiliser la fonction FUNCALL qui pourra, elle, compiler les arguments de <fonction>.

W.5 [Message d'avertissement de Complice]

W.5.<fnt>..Mauvais nombre d'arguments
- <fonction>

CAUSE : La séquence d'appel de la fonction <fonction> à l'intérieur de la fonction <fnt> ne correspond pas à sa définition. Par exemple:

```
(DE f () (CONS 1 2 3))
```

ou

```
(DE f () (g 1 2 3))
(DE g (x y) (CONS x y))
```

W.6 [Message d'avertissement de Complice]

W.6.<module>..Fonction externe au module:
<fonction>

CAUSE : Cet avertissement ne se produit qu'avec la fonction COMPILEMODULE. Il apparaît si la fonction <fonction> est utilisée dans le module <module> mais n'est créée par aucun des modules importés.

REMEDE : Si vous connaissez le module correspondant à <fonction> le rajouter dans la description du module <module> sous la clef **IMPORT**, sinon il faut s'arranger pour que <fonction> soit déjà compilée au moment du COMPILEMODULE et s'assurer qu'elle soit présente au moment de l'exécution.

W.7 [Message d'avertissement de Complice]

W.7.<fnt>..Fonction calculée:
<expression-lisp>

CAUSE : La fonction <fnt> utilise une fonction du type FUNCALL, APPLY, etc... l'<expression-lisp> est une valeur fonctionnelle. Par exemple :

```
(DE tw (f a) (FUNCALL f (FUNCALL f a)))
```

ARGUMENT : Pour l'exemple précédent : f.

REMEDE : Si il n'y a pas d'interaction entre la valeur fonctionnelle appelée et la fonction appelante, le code généré avec l'indicateur #:COMPLICE:PARANO-FLAG non positionné est performant.

W.8 [Message d'avertissement de Complice]

W.8.<fnt>..Appel explicite à l'évaluateur:
<expression-lisp>

CAUSE : La fonction <fnt> utilise une fonction du type EVAL, SET, etc... Par exemple :

```
(DE symev (v env) (OR (CASSQ v env) (SYMEVAL v)))
```

ARGUMENT : Pour l'exemple précédent : v.

REMEDE : Si il n'y a pas d'interaction entre l'évaluation demandée et l'environnement de l'appelant, le code généré avec l'indicateur #:COMPLICE:PARANO-FLAG non positionné est performant.

W.9 [Message d'avertissement de Complice]

W.9.compilefiles..Fonction externe au module:
‹fonction›

CAUSE : Cet avertissement ne se produit qu'avec la fonction COMPILEFILES. Il apparaît si la fonction ‹fonction› est utilisée dans un des fichiers argument de COMPILEFILES mais n'est créée par aucun de ces fichiers.

REMEDE : Il faut s'arranger pour que ‹fonction› soit deja compilée au moment du COMPILEFILES et s'assurer qu'elle soit présente au moment de l'exécution.

13.5.3 Remarques générales et exemples

13.5.3.1 utilisation de DEFVAR

Dans certains cas particuliers, Complice ne trouve pas les variables devant rester dynamiques (même avec #:COMPLICE:PARANO-FLAG à t), par exemple :

```
(DE LINE-COUNT ()
    (LET ((#:SYS-PACKAGE-ITSOFT 'count) (#:count:n 0))
         (UNTILEXIT EOF (READ))
         #:count:n))

(DE #:COUNT:BOL ()
    (SETQ #:count:n (ADD1 #:count:n))
    (BOL))
```

Dans ce cas la variable #:COUNT:N est utilisée de manière dynamique mais sera compilée de manière lexicale. Pour pallier ces problèmes, une variable déclarée globale par DEFVAR sera toujours compilée de manière dynamique.
Le programme précédent s'écrit :

```
(DEFVAR #:count:n)

(DE LINE-COUNT ()
    (LET ((#:sys-package-itsoft 'count) (#:count:n 0))
         (UNTILEXIT EOF (READ))
         #:count:n))

(DE #:count:bol ()
    (SETQ #:count:n (ADD1 #:count:n))
    (BOL))
```

Il est donc déconseillé d'utiliser DEFVAR pour des variables telle que **l, n, at, etc...**

13.5.3.2 utilisation des macros

Une macro-fonction instanciée dans un module mais non exportée n'aura plus d'existence après la compilation.

Le fichier de module **test** :

```
defmodule user
   files  (test)
   export (test)
```

Le fichier de code lisp **test** :

```
(DE test (n) (test-macro n))

(DMD test-macro (n) `(ADD1 ,n))
```

Une fois le module compilé la fonction TEST-MACRO n'existe plus.

13.5.3.3 générateurs de fonctions

Dans le cas des fonctions COMPILEFILES et COMPILEMODULE, Complice détermine les fonctions à compiler par la présence des primitives DE, DF, DEFUN, SETFN, DM, DMD, DEFMACRO. Pour pouvoir se définir de nouveaux générateurs de fonction on peut utiliser deux traits du compilateur.
Le premier est qu'il y a macro-expansion des formes lues dans les fichiers, il est donc possible de se définir des macro-fonctions générant des appels aux primitives standards vues plus haut.

```
(DMD DEFSYS (name . fval)
     `(DE ,(symbol 'system name) ,@fval) )

(DEFSYS ob () (OBLIST 'system))
```

Par contre dans les fichiers de description de module vous devez spécifier les noms complets des fonctions à exporter.

```
defmodule system
   files  (foo)
   export (#:system:ob)
```

Le deuxième trait est qu'une forme commençant par PROGN est elle-même parcourue pour déterminer d'éventuelles définitions de fonctions.

```
(DMD newtype (type make pred)
     `(PROGN
         (DE ,make (l)
             (LET ((v (APPLY 'VECTOR l)))
                  (TYPEVECTOR v ',type)))
         (NEWL list-type ',type)
         (DE ,pred (v) (EQ (TYPEVECTOR v) ',type))
         (DEFMACRO-OPEN ,pred (v) `(EQ (TYPEVECTOR ,v) ',,type))))

(DEFVAR list-type ())
(newtype cell makecell cellp)
```

13.5.3.4 utilisation de la fonction PRECOMPILE

La fonction PRECOMPILE permet entre autres de ne définir des données qu'au moment du chargement.

```
(DE static-stream ()
    (LET ((stream '#.(CONS -1 (CIRLIST 0 1))))
         (PROG1 (CADR stream)
                (RPLACD stream (CDDR stream)))))
```

Cette fonction est correctement compilée par les fonctions COMPILE et COMPILE-ALL-IN-CORE mais le code LAP correspondant ne pourra pas être déchargé dans un fichier à cause de la circularité de la donnée. Dans ce cas on peut utiliser la fonction PRECOMPILE de la manière suivante :

```
(DMD reconstruct (data)
     `(PRECOMPILE ',(EVAL data) () () (EVAL (KWOTE ,data))))

(DE static-stream ()
    (LET ((stream (reconstruct (CONS -1 (cirlist 0 1)))))
         (PROG1 (CADR stream)
                (RPLACD stream (CDDR stream)))))
```

13.5.3.5 les mauvais traits de Complice

La fonction COMPILEMODULE crée toujours le fichier objet dans le catalogue de travail courant.

Si un module utilise des macro-fonctions définies dans un autre module il faut que ce dernier soit chargé durant la compilation.

Les fonctions de contrôle lexicales (TAGBODY, BLOCK, GO, RETURN ..) font appel systématiquement aux routines spécialisées de l'interprète.

La fonction UNLOADMODULE ne marche pas.

CHAPITRE 14

Les Interfaces Externes

Matthieu Devin

Un première section décrit les fonctions permettant d'interfacer Le_Lisp avec des modules externes écrits dans d'autres langages de programmation. Les sections suivantes décrivent les détails du processus consistant à interfacer Le_Lisp avec différents langages sur différents systèmes. Seule la section concernant le système UNIX est décrite ici. Pour les interfaces avec le système VMS voir [Dana 86], pour celles sur Multics voir [Grimm 86].

14.1 Les Fonctions de l'Interface

Le_Lisp permet d'appeler des modules externes écrits dans un langage autre que Lisp. Ces modules doivent être intégrés au système Lisp au moyen de l'éditeur de liens propre au système d'exploitation utilisé.

Ces fonctions ne sont pas standard dans Le_Lisp, assurez-vous toujours de leur existence avant utilisation.

(GETGLOBAL <strg>) [SUBR à 1 argument]

retourne l'adresse du symbole global <strg>. Ce symbole doit exister dans la table des symboles engendrée par l'éditeur de liens à la création du système Le_Lisp. Le résultat de cette fonction peut être utilisé comme premier argument de la fonction suivante.

La notion de *symbole* dépend bien sûr des systèmes. Sur les systèmes UNIX les symboles correspondant à des programmes écrits en C sont généralement précédés du caractère _ (exemple _getenv). Sur le système Multics, les symboles sont fait de deux parties précisant un nom de segment et un point d'entrée dans le segment (exemple ioa_$nnl).

```
ex : (getglobal "cond")          ->   -5812
```

(CALLEXTERN <adr> <typ> <v1> <t1> .. <vN> <tN>) [SUBR à N args]

permet d'appeler un module externe rangé à l'adresse <adr>. Ce module doit retourner un objet de type <typ>. Ce type est codé comme suit :

```
0     pointeur
1     nombre entier
2     nombre flottant
3     chaîne de caractères
4     adresse de vecteur
5     nombre entier passé par référence (FORTRAN)
```

```
6       nombre flottant passé par référence (FORTRAN)
```

Les arguments de ce module sont <v1> ... <vN> et leurs types respectifs <t1> .... <tN> codés d'une manière identique.

(DEFEXTERN <symb> <ltype> <type>) [FSUBR]

permet d'associer dynamiquement un module externe avec une fonction Le_Lisp. <symb> est le nom d'un module externe qui va devenir une nouvelle fonction Le_Lisp. <ltype> est la liste des types des arguments d'entrée et <type> le type de la valeur retournée. Ces types sont l'un des symboles suivants :

T	pour des pointeurs Le_Lisp
EXTERNAL	pour des pointeurs externes
FIX	pour des nombres entiers
FLOAT	pour des nombres flottants
STRING	pour des chaînes de caractères
VECTOR	pour des adresses de vecteurs
RFIX	pour des nombres entiers passés par référence
RFLOAT	pour des nombres flottants passés par référence

Le type EXTERNAL correspond à un pointeur externe à l'espace mémoire Lisp qui est représenté sous la forme d'un CONS de deux entiers ou d'un flottant selon les systèmes. En général on recevra de tels pointeurs comme résultats de procédures externes et on se contente de les stocker pour pouvoir les passer à d'autres procédures le moment venu.

DEFEXTERN peut être défini en Lisp de la manière suivante :

```
(dmd defextern (nom ltype . type)
     (buildextern nom (getglobal nom)
                  ltype (or (car type) 'fix)))

(de buildextern (nom adr ltype type)
   ; définit une procédure externe
   (let* ((n -1)
          (lvar (mapcar (lambda (l)
                          (symbol '#:system:callext
                                  (concat "arg" (incr n))))
                        ltype))
          (body `(callextern
                  ,(if (numberp adr) adr `',adr)
                  ,(convtypextern type)
                  ,@(mapcan (lambda (type var)
                              (if (eq type 'external)
                                `((vag ,var) ,(convtypextern type))
                                `(,var ,(convtypextern type))))
                            ltype lvar))))
         (when (eq type 'external)
               (setq body `(loc ,body)))
         (if (and (numberp adr) (zerop adr))
             (error 'defextern 'errudf nom)
             `(de ,nom ,lvar ,body))))

(de convtypextern (type)
    (selectq type
        (external 0)
        (fix 1)
```

```
            (float 2)
            (string 3)
            (vector 4)
            (rfix 5)                    ; FIX par référence (FORTRAN)
            (rfloat 6)                  ; FLOAT par référence (FORTRAN)
            ((t) 0)                     ; T arrête les clauses!!
            (t (error 'convtypextern 'erroob type))))
```

ex : si la procédure suivante est définie en C

```
double ctest (strg,nf,ni,vect)
      char *strg; double nf; int ni; int *vect;{
      int i;
      printf("la chaine est %s ", strg);
        printf("le flottant est %e ", nf);
        printf("l'entier est %d ", ni);
        printf("le vecteur contient vect[0]=%8x, vect[1]=%8x ",
                        vect[0], vect[1]);
        i = vect[0]; vect[0] = vect[1]; vect[1] = i;
      return(nf*ni);
}
```

; sous Le_Lisp

```
(defextern _ctest (string float fix vector) float)    -> _ctest
(defvar vec #[(a b) #[1 2 3]])                        -> vec
(_ctest "Fou Bare" 123.45 12 vec)                     -> 1481.4
        la chaine est Fou Bare
        le flottant est 1.234500e+02
        l'entier est 12
        le vecteur contient vect[0]=  9ed30, vect[1]=  63ae4
vec    ->   #[#[1 2 3] (a b)]
```

14.2 Liens avec des Modules Externes sous UNIX

Nous expliquons ici comment interfacer Lisp avec des modules écrits en C (DEFEXTERN), et comment rappeler Lisp depuis ces modules (Lispcall).

Ces traits sont les moins portables du système et concernent ici uniquement les systèmes Unix: Vax, Sm90, SUN, Sps9 et Perkin-Elmer.

La possibilité de rappeler l'interprète Lisp depuis C est aujourd'hui disponible sur Sm90, SUN, Sps7 et VAX, et sera étendue ultérieurement aux autres systèmes.

Sur le système Multics, il est possible d'appeler des programmes écrits en PL/1, en Pascal ou en FORTRAN et rappeler l'interprète Le_Lisp depuis un programme écrit en PL/1 ou en Pascal par des méthodes très proches de celles décrites ici. Cette interface est due à José Grimm.

Nous tenons a remercier Laurent Fallot pour sa contribution au lien dynamique des modules C. Nous remercions particulièrement Dominique Clément, Laurence Gallot et Gilles Kahn, qui ont aidé à concevoir et mettre au point ces interfaces.

14.2.1 Philosophie

Le langage Lisp n'est pas adapté à toutes les tâches que l'on désire faire faire à une machine. Il est notamment peu efficace pour ce qui concerne les longs calculs numériques (résolution d'équations différentielles, maillages, etc..) ou les opérations bit à bit sur des écrans (rasters-ops).

Pour ces tâches on préfère généralement utiliser des langages très performants dans leurs spécialités: FORTRAN, C, Assembleur.

Le_Lisp fournit le moyen d'interfacer des sous-programmes écrits en d'autres langages avec l'interprète : les sous-programmes deviennent accessibles par des fonctions Lisp.

Il est possible de passer des arguments de tous types à ces sous-programmes (entiers, flottants, chaînes de caractères, vecteurs d'objets Lisp, Objets Lisp quelconques) et de recevoir des résultats de ces sous-programmes (entiers, flottants, chaînes de caractères, pointeurs quelconques).

Le_Lisp peut donc être utilisé pour piloter des sous-programmes très performants écrits en d'autres langages. Le_Lisp est alors à considérer comme un outil d'interface et de dialogue avec ces sous-programmes.

Exemples de réalisations :

Interface avec le gestionnaire d'écran bitmap ASH sur Sm90 : les fonctions de gestions de fenêtres et les commandes graphiques deviennent des fonctions Lisp classiques [M. Devin, L. Gallot 85]. Cette interface a été portée sur Sps9 et SUN par Laurence Gallot.

Interface avec un interprète Prolog écrit en C sur Sm90 et Sps9 : on a accès aux fonctions permettant de construire des clauses Prolog et de lancer une inférence. Cette manière de faire du Prolog devrait être plus efficace que l'utilisation d'un interprète Prolog écrit en Lisp [D.Clément]. Certains prédicats Prolog permettent de rappeler l'interprète Lisp. Ces prédicats sont principalement utilisés pour créer des objets Lisp depuis Prolog.

Interface avec des routines graphiques sur Sps9 : un système expert de génération de coupes de digues portuaires dessine les solutions engendrées sur un écran bitmap [P. Haren et. al. 85].

Interface avec le gestionnaire d'écran SUNVIEW sur SUN.

14.2.2 Appel de fonctions écrites en C

Nous décrivons ici le moyen d'interfacer Lisp avec des programmes écrits en C. Tout ce que nous décrivons est aussi valable pour FORTRAN, qui respecte les conventions d'appel de C sous Unix, et pour les programmes assembleurs s'ils respectent aussi ces conventions.

Il peut y avoir quelques problèmes pour interfacer des programmes en Pascal qui manipulent des fichiers. En Pascal les fichiers doivent en effet être déclarés dans la déclaration PROGRAM : il est difficile dans ce cas de lier des routines Pascal hors d'un contexte Pascal. Dans ce cas on doit écrire un programme Pascal appelant le système Le_Lisp comme une procédure externe [D. Clément, L. Gallot 85].

Pour pouvoir appeler une fonction C depuis l'interprète Lisp il faut :
1- *lier* le code compilé de la fonction et le noyau de l'interprète en un même

programme.
2- *associer* à une fonction Lisp le point d'entrée de la fonction C. L'appel de la fonction Lisp permet alors le lancement de la fonction C.

14.2.2.1 Lien des modules

L'opération de *lien* consiste à réunir dans un même programme le code de plusieurs sous-programmes. Il s'agit ici de lier l'interprète Lisp et un module C contenant le code des fonctions C que l'on désire appeler.

Ce lien peut être réalisé statiquement, lors de la compilation de l'interprète, ou dynamiquement, à tout instant sous le contrôle de l'interprète.

L'opération de lien statique consiste en la création d'un nouveau binaire exécutable contenant le noyau de l'interprète Le_Lisp et les modules C. Ce nouveau binaire peut alors être utilisé pour engendrer un nouveau système Lisp. Elle peut être réalisée sur toutes les machines Unix, mais est assez lourde car il est nécessaire de recompiler le système à chaque modification des modules C.

L'opération de lien dynamique peut être uniquement réalisée sur les machines disposant d'un éditeur de lien permettant un lien incrémental (option -A de la commande Unix *ld*). Elle est actuellement uniquement implantée sur Sm90, SUN, Sps7, Vax et Sps9. Elle remplace avantageusement l'opération de lien statique car il n'est pas nécessaire de recompiler le système à chaque modification des modules C. Les modules liés dynamiquement sont de plus conservés dans les images mémoires, le lien dynamique est donc aussi durable que le lien statique.

14.2.2.1.1 Le lien dynamique

(CLOAD <string>) [SUBR1]

L'argument de cette fonction est le nom d'un module compilé qui doit être lié à l'interprète. Cette fonction appelle l'éditeur de lien *ld* pour lier le module avec l'interprète utilisé. Le code compilé du module C est chargé dans la zone Lisp CODE et peut donc être sauvegardé dans une image mémoire.

Exemple :

```
$ lelisp
******* Le_Lisp (by INRIA) version 15.2 (20/Mai/86) [sm90]
= Systeme standard compile avec environnement avec compilateur
?
? ; On compile un module C foo.c en créant le binaire foo.o
? !cc -c foo.c
= t
? ; On lie ensuite le module foo.o à l'interprète
? (cload "foo.o")
= foo.o
```

La chaîne de caractères argument sert de base pour construire un appel à l'éditeur de lien *ld*. Cette chaîne peut donc contenir certaines options de l'éditeur de lien permettant le chargement de bibliothèques, une trace lors de la résolution des liens, ou le lien de plusieurs modules à la fois. Voyez la documentation Unix ld(1) sur votre machine pour plus de détails.

Voici quelques exemples d'utilisation de la fonction CLOAD :

L'appel suivant permet de charger le module *ash.o* et la bibliothèque *raster.a.*

```
(cload "ash.o -lraster")
```

L'appel suivant permet de charger le point d'entrée *sinh* (sinus hyperbolique) de la bibliothèque mathématique.

```
(cload "-u _sinh -lm")
```

L'appel suivant permet de charger les trois modules *foo.o, gee.o* et *bar.o.*

```
(cload "foo.o  gee.o  bar.o")
```

DETAILS DU PROCESSUS

La chaîne de caractères argument sert de base à la construction d'un appel à l'éditeur de lien *ld,* de la manière suivante (cf doc Unix ld(1)).

L'appel

```
(CLOAD <string>)
```

engendre l'appel :

```
ld -A <lelispbin> -N -x -T <bout> -o <temporaire> <string> -lc
```

Sur SPS9 on engendre l'appel :

```
ld -C -A <lelispbin> -N -x -T <bout> -o <temporaire> <string> -lc
```

où *<lelispbin>* est un path permettant de trouver le binaire Le_Lisp en train de tourner, *<bout>* est la première adresse disponible dans la zone Lisp réservée au code compilé, *<temporaire>* est un fichier temporaire de nom unique (dans /tmp).

Après lien du module, le fichier *<temporaire>* contient le code du module C. Il est chargé dans la zone Lisp CODE. C'est le fichier *<temporaire>* qui sera utilisé à la place de *<lelispbin>* pour un autre appel de *CLOAD.* Ceci permet d'enchaîner plusieurs appels à CLOAD.

14.2.2.1.2 Le lien statique

Le lien statique est décrit dans la documentation d'installation du système, au début du manuel de référence.

Il consiste à lier une fois pour toutes le code de l'interprète Le_Lisp, *lelispbin,* et les modules C pour créer un nouveau binaire. Il faut ensuite configurer le système (chargement de l'environnement et création d'une image mémoire) comme décrit dans la documentation d'installation.

En général on crée un point d'entrée dans le makefile permettant de lier et configurer le nouveau système Lisp.

Exemple :

Voici par exemple les déclarations du makefile permettant de lier le module *ash.o* et la bibliothèque *raster.a* au noyau *lelispbin,* et de configurer un système de nom *ashlelisp.* Voir la documentation d'installation de Le_Lisp sur Unix pour plus de détails.

```
ashlelisp :     ashlelispbin ash.ll
        config ashlelisp ashlelispbin ash.ll
```

```
ashlelispbin : ash.o
        cc -x -n $(CFLAGS) lelispbin.o ../common/lelisp.c \
                           ash.o -lraster -lm -lc -o ashlelispbin
```

Un inconvénient du lien statique est la nécessité de reconfigurer le système complet à chaque modification des modules C ce qui est souvent assez long.

14.2.2.2 Déclarations des fonctions C

Une fois que les modules sont liés à l'interprète on doit associer des fonctions Lisp aux fonctions C. La fonction Lisp DEFEXTERN (voir plus haut) réalise cette association.

Le tableau ci dessous décrit les conversions qui ont lieu lors du passage des arguments à C et lors du retour du résultat.

Type Lisp	Type C	Argument	Résultat
fix	int	extension du signe à 32 bits	conversion à 16 bits
float	double	passage de la valeur	allocation d'un flottant Lisp
string	char *	passage d'un pointeur sur les caractères de la chaîne	Allocation d'une chaîne Lisp et recopie de la chaîne C dans la chaîne Lisp
vector	any * []	passage d'un vecteur de pointeurs Lisp.	non implémenté
t	any *	aucune modification : C reçoit le pointeur Lisp* tel quel.	aucune modification : Lisp reçoit l'objet C tel quel. Attention : Le_Lisp risque de faire n'importe quoi si on lui donne n'importe quel pointeur.
external	any *	L'objet Lisp doit être un CONS de deux entiers. Ces deux entiers servent à former une adresse (cf fct Lisp VAG) qui est passée à C.	Allocation d'un CONS contenant en CAR et CDR les parties hautes et basse de l'objet C (cf fct Lisp LOC). Le CONS est le résultat de la fonction.

Les pointeurs Lisp sont soit des pointeurs dans les zones de données Lisp, soit des petits entiers 16 bits. Le fichier *lelisp.h* donne les déclarations C des structures sur lesquelles pointent les pointeurs Lisp. Nous détaillons ce point plus loin.

Le type *external* permet de manipuler des pointeurs C en Lisp. Les pointeurs C quelconques ne sont pas manipulables par l'interprète, ils doivent être convertis en un

CONS de deux entiers représentant la partie haute et la partie basse de l'adresse. Le type external permet la conversion Pointeur C<->Doublet Lisp à chaque appel/retour de fonctions C.

14.2.2.3 Exemples

Nous donnons ci dessous des exemples montrant l'utilisation de la fonction DEFEXTERN.

Pour chaque exemple nous donnons d'abord le programme C à appeler, la déclaration DEFEXTERN permettant de construire la fonction Lisp correspondante, et des exemples d'utilisation de cette fonction.

Voici une fonction recevant trois arguments (entier, chaîne, flottant) et rendant un nombre flottant.

```
double
foo (x, s, f)
   int x; char *s; double f;{
   printf("Entier : %d, Chaine : %s, Flottant : %g\r\n", x, s, f);
   return(x * strlen(s) * f);
}
```

```
? (defextern _foo (fix string float) float)
= _foo
?
? (_foo 10 "hello" 1.2e+10)
Entier : 10, Chaine : hello, Flottant : 1.2e+10
= 6.e+11
?
```

Voici une fonction recevant une chaîne et rendant une chaîne :

```
char *
getenv(s)
  char *s; {
    ...        /* rend une chaîne C */
}
```

```
? (defextern _getenv (string) string)
= _getenv
?
? (_getenv "HOME")
= /udd/lelisp/devin       ; c'est maintenant une chaîne Lisp
?
```

Voici une fonction recevant un vecteur. L'argument *v* de cette fonction est déclaré de type *"int v[]"*. On pourra lui passer un vecteur d'entiers Lisp (entiers 16 bits), qui sera reçu comme un vecteur d'entiers C (32 bits signés) tous contenus dans l'intervalle 0..65536. Il est en général commode de passer la longueur du vecteur en argument.

```
voir (v, l)
   int v[], l;{     /* l est la longueur du vecteur */
   int i;
   for (i = 0; i < l; i++)
     printf("%d ", v[i]);
```

```
   printf("\r\n");
}

? (defextern _voir (vector fix))
= _voir
?
? (_voir #[1 2 3 4 -4 -3 -2 -1] 8)
1 2 3 4 65532 65533 65534 65535
= 0
```

Voici un exemple d'utilisation du type *external.* Il montre comment manipuler un pointeur quelconque reçu de C (ici un pointeur sur une fenêtre allouée en C).

```
window *
make_window (x, y, l, h)
  int x, y, l, h; {
    ...    /* crée un objet structuré fenêtre et
              rend un pointeur dessus */
}

select_window (w)
  window *w; {
    ....   /* recoit un pointeur sur un objet structuré fenêtre */
}

? (defextern _create_window (fix fix fix fix) external)
= _create_window
? (defextern _select_window (external))
= _select_window
?
? (setq a (_create_window 10 10 400 400))
= (12 . 2348)    ; le pointeur manipulable en Lisp.
?
? (_select_window a) ; on peut repasser le pointeur à C.
= 0
```

Les arguments correspondant au type *t* sont passés aux fonctions C sans aucune modification. Ces fonctions reçoivent donc des objets Lisp, en général des pointeurs sur des structures (le cas des entiers et des flottants est différent, voir plus loin). Le fichier *lelisp.h* de la distribution standard fournit les déclarations des structures C correspondant aux objets Lisp.

Voici par exemple la déclaration des Cons, structures à deux champs (*ll_car* et *ll_cdr*) contenant chacun un objet Lisp.

```
struct LL_CONS {
     LL_OBJECT  ll_car;
     LL_OBJECT  ll_cdr;
};
```

Le type *LL_OBJECT* est un type polymorphe regroupant tous les types d'objets Lisp (LL_SYMBOL, LL_CONS, etc...). C n'ayant pas de notion de type polymorphe, le type *LL_OBJECT* est déclaré de la manière suivante :

```
typedef char *LL_OBJECT;
```

Exemple :

La fonction ci-dessous extrait le *CAR* d'un doublet Lisp reçu en argument.

```
#include "lelisp.h"

LL_OBJECT
car (o)
  struct LL_CONS *o;{
  return (o->ll_car);
}

? (defextern _car (t) t)
= _car
?
? (_car '((a) (b) (c)))
= (a)
?  ; Attention il est parfois dangereux de prendre le CAR de n'importe quoi!
? (_car 1)
signal 3
OUPPS! J'ai failli faire un core
$
```

Les symboles se manipulent de la même manière que les Cons. Les champs *ll_alink* et *ll_pname* sont utilisés par Le_Lisp pour lier les symboles dans l'oblist, il ne faut absolument pas toucher à leur contenu.

```
struct LL_SYMBOL {
     LL_OBJECT  ll_cval;
     LL_OBJECT  ll_plist;
     LL_OBJECT  ll_fval;
     LL_OBJECT  ll_alink;
     LL_OBJECT  ll_pkgc;
     LL_OBJECT  ll_oval;
     char       ll_ftype;
     char       ll_ptype;
     short      ll_pad;
     LL_OBJECT  ll_pname;
};
```

Les chaînes de caractères et les vecteurs sont des pointeurs sur des pointeurs sur des structures de taille variable. Cette double indirection est nécessaire pour le récupérateur. Les structures de taille variable n'existant pas en C, les déclarations suivantes correspondent à des objets de taille 1 (chaîne de un caractère, vecteur à un seul élément). Le champ *ll_XXXsiz* indique la taille réelle de l'objet (nombre de caractères de la chaîne ou nombre d'objets Lisp du vecteur). Le premier élément de l'objet est dans le champ *ll_XXXfil,* les autres éléments sont à sa suite dans la mémoire. Les champs *ll_XXXtyp* contiennent les types de vecteurs et de chaînes.

```
struct LL_STRING {
     struct {
        struct LL_STRING *ll_strarr;
        int               ll_strsiz;
```

```
        char              ll_strfil;
    } *ll_strobj;
    LL_OBJECT ll_strtyp;
};
struct LL_VECTOR {
    struct{
        struct LL_VECTOR *ll_vecarr;
        int               ll_vecsiz;
        LL_OBJECT         ll_vecfil;
    } *ll_vecobj;
    LL_OBJECT ll_vectyp;
};
```

Voici par exemple une fonction qui imprime un par un les caractères d'une chaîne Lisp :

```
un_par_un (chaine)
  struct LL_STRING *chaine; {
  int i;
  int size;
  char *s;
  size = (chaine->ll_strobj)->ll_strsiz;   /* la taille de la chaîne */
  s = &((chaine->ll_strobj)->ll_strfil);   /* l'adresse du premier caractere */
  for (i = 0; i < size; i++)
     printf("caractere %d :%c\r\n", i, *(s+i));
}
```

Les objets Lisp de type FIX ne sont pas des pointeurs, mais des valeurs immédiates. Les entiers sont des nombres 16 bits. Un objet Lisp de type Entier est donc un pointeur (32 bits) dont les 16 premiers bits sont toujours à zéro.

Les flottants peuvent être implantés de plusieurs manières différentes selon les systèmes. La macro C, *LL_C_FLOAT* permet de transformer un flottant Lisp en flottant C. Il n'est en général pas possible de transformer un flottant C en flottant Lisp. La fonction C suivante permet par exemple de recopier un vecteur de flottants Lisp dans un tableau de flottants C.

```
/* le tableau de flottants */
double table[128];

C_float_vector (vector)
  struct LL_VECTOR *vector; {
  int i;
  int size;
  LL_FLOAT *v;
  size = (vector->ll_vecobj)->ll_vecsiz; /* la taille du vecteur */
  v = &((vector->ll_vecobj)->ll_vecfil); /* l'adr. du premier flottant Lisp *.Lb
  for(i = 0; i < size; i++)
     table[i] = LL_C_float(*(v+i));
}
```

Cette dernière fonction permet de calculer la somme point à point de deux vecteurs en

stockant le résultat dans un troisième vecteur.

```
plus_vector(v1, v2, v3)
  struct LL_VECTOR *llv1, *llv2, *llv3; {
  int i;
  int size;
  LL_FLOAT *v1, *v2, *v3;

  size = (v1->ll_vecobj)->ll_vecsiz;    /* la taille du vecteur */

  /* l'adresse du premier flottant du vecteur llv1 */
  v1 = &((llv1->ll_vecobj)->ll_vecfil);

  v2 = &((llv2->ll_vecobj)->ll_vecfil);
  v3 = &((llv3->ll_vecobj)->ll_vecfil);

  for(i = 0; i < size; i++)
     *(v3+i) = C_LL_FLOAT(LL_C_FLOAT(*(v1+i))+LL_C_FLOAT(*(v2+i)));
}
```

AVERTISSEMENT

La manipulation des objets Lisp en C est en général un sport d'assez haut vol. Il faut faire très attention lorsque l'on modifie le contenu des objets Lisp reçus en arguments.

On doit notamment ne *jamais* toucher au contenu des champs : ll_pname, ll_alink, ll_strobj, ll_vecobj, ll_strarr, ll_vecarr, ll_strtyp, ll_vectyp, au risque de provoquer une erreur système de l'interprète.

Nous n'avons pas encore d'expérience de dialogue Lisp/FORTRAN, notamment en ce qui concerne le passage de tableaux de flottants entre Lisp et FORTRAN. Il paraît souhaitable d'utiliser C pour recopier le vecteur Lisp dans un tableau FORTRAN, et de recopier le tableau FORTRAN résultat dans un vecteur Lisp passé en argument.

14.2.3 Rappel de Lisp depuis C

Sur Sm90, VAX, Sps7 et SUN il est aujourd'hui possible de rappeler l'interprète Lisp depuis une fonction C qui a elle-même été appelée par Lisp. Le schéma de ce dialogue est le suivant :

1- Appel de l'interprète Lisp.
2- Appel de fonctions C définies par DEFEXTERN.
3- Rappel de l'interprète depuis ces fonctions C.

Les appels Lisp/C /Lisp/C /Lisp/.. peuvent être imbriqués à n'importe quel niveau.

14.2.3.1 Mise en oeuvre

Ce dialogue est réalisé au niveau Lisp par l'utilisation de la primitive DEFEXTERN décrite plus haut et au niveau C par trois fonctions : *getsym, pusharg* et *lispcall.* Ces fonctions utilisent les définitions des objets Lisp du fichier lelisp.h et sont définies de la manière suivante :

```
struct LL_SYMBOL *
getsym(s)
  char *s; {
```

```
  ...  /* Rend le symbole Lisp ayant pour Pname la chaîne s */
}

pusharg(type, val)
  int type; any val; {
  ...  /* Empile l'argument val de type type */
}

LL_OBJECT
lispcall(typeres, narg, fonction)
  int typeres;                /* le type du résultat de la fonction Lisp */
  int narg;                   /* le nombre d'arguments empilés par pusharg */
  struct LL_SYMBOL *fonction;  /* La fonction lisp à lancer */
  {
  ...  /* Appelle la fonction Lisp avec les narg arguments empilés.
          Rend le résultat de cette fonction. */
}
```

Les types des arguments sont à choisir parmi les types symboliques : LLT_FIX, LLT_FLOAT, LLT_STRING, LLT_VECTOR et LLT_T (pointeur quelconque). Comme pour l'appel Lisp->C il y a conversion automatique des arguments lors du passage de C à Lisp, comme décrit dans le tableau plus haut.

14.2.3.2 Exemples :

Voici quelques exemples de définitions de fonctions C appelant l'interprète Lisp, et de leur utilisation.

14.2.3.2.1 Appel de la fonction CONS

Voici une fonction C qui reçoit deux objets Lisp quelconques et rend un CONS de ces deux objets.

```
struct LL_CONS *
cons_en_c (a, b)
  LL_OBJECT a, b; {
  struct LL_SYMBOL *lisp_cons;
  lisp_cons = getsym("cons");
  pusharg(LLT_T, a);
  pusharg(LLT_T, b);
  return( (struct LL_CONS *) lispcall(LLT_T, 2, lisp_cons));
  }
```

```
? (defextern _cons_en_c (t t) t)
= _cons_en_c
?
? (_cons_en_c 1 2)
= (1 . 2)
? (trace cons)
= cons
? (_cons_en_c 1 2)
```

```
cons ---> 1 2       ; la fonction lisp cons est bien appelée
cons <--- (1 . 2)
= (1 . 2)
?
```

14.2.3.2.2 Fonctions N-aires

On peut bien sûr appeler des fonctions Lisp N-aires. Voici par exemple une fonction qui transforme un vecteur en liste, en appelant la fonction Lisp *list*.

Notez qu'en général on n'effectue pas un appel à getsym à chaque appel de la fonction C (c'est la partie la plus longue de l'appel) mais une seule fois dans une partie d'initialisation.

```
#define NULL (struct LL_SYMBOL *) 0

struct LL_SYMBOL *lisp_list = NULL;

struct LL_CONS *
vect_to_list (v, l)
  int v[], l; {
  int i;

  if(lisp_list == NULL)
    lisp_list = getsym("list");

  for (i = 0; i < l; i++)
    pusharg(LLT_T, v[i]);

  return( (struct LL_CONS *) lispcall(LLT_T, l, lisp_list));
 }
```

```
? (defextern _vect_to_list (vector fix) t)
= _vect_to_list
?
? (_vect_to_list #[1 2 3 4 -4 -3 -2 -1] 8)
= (1 2 3 4 -4 -3 -2 -1)
?
```

14.2.3.2.3 Récursion

Rappelons que les appels Lisp/C peuvent s'emboiter récursivement. On peut, par exemple, écrire la fonction Fibonnacci sous la forme d'une fonction Lisp et d'une fonction C mutuellement récursives.

```
? !cat fib.c
struct LL_SYMBOL *lisp_fib;

init_fib(){
  lisp_fib = getsym("fib");
}

int
fib (n)
  int n;{
  int x;

  if (n==1) return(1);
```

```
  else
  if (n==2) return(1);
  else{
  pusharg(LLT_FIX, n-1);
  x = (int) lispcall(LLT_FIX, 1, lisp_fib);
  pusharg(LLT_FIX, n-2);
  return(x+ (int) lispcall(LLT_FIX, 1, lisp_fib));
  }
}

= t
?  ; lien (dynamique) du module fib.o avec l'interprète
? !cc -c fib.c
= t
? (cload "fib.o")
= fib.o
?
? (defextern _fib (fix) fix)
= _fib
? (defextern _init_fib ())
= _init_fib
? (_init_fib)
= 0
?  ; Voici la fonction Lisp qui appelle la fonction C lors de la récursion :
? (de fib (n)
?     (cond ((eq n 1) 1)
?           ((eq n 2) 1)
?           (t (+ (_fib (1- n))       ; le premier appel (C) récursif
?                 (_fib (- n 2))))))) ; le second ...
= fib
?
? (fib 4)
= 3
?
? (trace fib _fib)
= (fib _fib)
?
? (fib 4)
fib ---> n=4
_fib ---> arg106=3
fib ---> n=2
fib <--- 1
fib ---> n=1
fib <--- 1
_fib <--- 2
_fib ---> arg106=2
_fib <--- 1
fib <--- 3
= 3
?
? (time '(fib 20))
= 9.0
```

```
? ; soit du même ordre de grandeur que la fonction Lisp récursive
? ; interprétée (8.75 sur cette machine).
```

CHAPITRE 15

Le Terminal Virtuel

Pour libérer l'utilisateur de la gestion explicite de tous les types de terminaux vidéo alphanumériques, Le_Lisp a introduit la notion de *terminal virtuel* qui se compose d'un ensemble de variables et de fonctions décrivant les fonctionnalités de ce genre de terminaux. Un grand nombre de terminaux physiques possèdent une description sous forme de terminal virtuel Le_Lisp. Ce terminal virtuel est initialisé par la fonction INITTY qui, le cas échéant, chargera un fichier de la bibliothèque des terminaux virtuels Le_Lisp. En cas d'absence de description d'un terminal, cette fonction lancera la compilation de cette description à partir d'autres bases de données de description de terminaux, les bases *termcap* et *terminfo*.

(INITTY <symb>) [SUBR à 0 ou 1 argument]

initialise le terminal virtuel pour un terminal de type <symb>. Si l'argument est absent, le type du terminal physique est donné d'abord par l'appel de (GETENV "TERM") puis par le nom du système (SYSTEM). En cas d'échec d'initialisation du terminal virtuel, l'erreur ERRVIRTTY se déclenche dont le libellé par défaut est :

```
** <fnt> : terminal inconnu : <symb>     ou bien
** <fnt> : unknown terminal : <symb>
```

dans lequel <fnt> est le nom de la fonction ayant provoqué l'erreur (TERMCAP ou TERMINFO) et <symb> est le nom du terminal inconnu. Voici les messages obtenus en cas d'absence d'un terminal :

```
? (INITTY 'alogon)
; On crée le fichier virtty à partir de termcap pour : alogon
** termcap : terminal inconnu : alogon
; On crée le fichier virtty à partir de terminfo pour : alogon
** terminfo : terminal inconnu : alogon
= alogon
```

#:SYSTEM:VIRTTY-DIRECTORY [Variable]

contient le nom du catalogue renfermant les fichiers de description Le_Lisp des terminaux virtuels. L'extension par défaut de ces fichiers est la valeur de #:SYSTEM:LELISP-EXTENSION.

#:SYSTEM:TERMCAP-FILE [Variable]

contient le nom du fichier contenant les descriptions des terminaux en format *termcap*. Si la variable d'environnement TERMCAP possède une valeur (obtenue par (GETENV "TERMCAP")), cette valeur remplace la variable globale #:SYSTEM:TERMCAP-FILE.

#:SYSTEM:TERMINFO-DIRECTORY [Variable]

contient le nom du catalogue contenant les descriptions des terminaux en format *terminfo*. Si la variable d'environnement TERMINFO possède une valeur (obtenue par (GETENV "TERMINFO")), cette valeur remplace la variable globale #:SYSTEM:TERMINFO-DIRECTORY.

TERMCAP [Feature]

ce trait indique que le traducteur des descriptions des terminaux virtuels en format *termcap* est chargé en mémoire. Ce traducteur se trouve dans la bibliothèque standard sous le nom de TERMCAP.

TERMINFO [Feature]

ce trait indique que le traducteur des descriptions des terminaux virtuels en format *terminfo* est chargé en mémoire. Ce traducteur se trouve dans la bibliothèque standard sous le nom de TERMINFO.

15.1 Les Fonctions sur le Terminal Virtuel

Les fonctions du terminal virtuel se divisent en trois groupes : les fonctions standard, les fonctions obligatoires et les fonctions facultatives.

Les fonctions standard permettent de réaliser l'interaction minimale avec le système Le_Lisp, c'est-à-dire l'écriture de caractères à l'écran et la lecture de caractères au clavier. Ces fonctions sont utilisées par les fonctions BOL, EOL, et FLUSH pour gérer le dialogue sur le terminal.

Les fonctions obligatoires sont utilisées par les outils plein écran (l'éditeur PEPE, l'environnement multi-fenêtres, les jeux, etc...).

Les fonctions facultatives permettent de réaliser des effets vidéos plus fins mais ne sont pas utilisées par les outils fournis dans la bibliothèque standard.

Le comportement des fonctions décrites ici est paramétré par le type du terminal effectivement connecté au système. Si ce terminal ne peut pas réaliser l'effet désiré, la fonction TYERROR est appelée. Cette fonction retourne la valeur () par défaut. Elle peut, bien sûr, être redéfinie pour générer une erreur ou bien pour essayer de réaliser l'opération désirée d'une autre manière.

(TYERROR <list>) [SUBR à 1 argument]

Cette fonction est automatiquement appelée par les fonctions du terminal virtuel lorsqu'il n'est pas possible d'obtenir un effet donné sur le terminal connecté au système. L'argument <list> est l'appel qu'il n'a pas été possible de réaliser. Le traitement standard de cette fonction retourne la valeur (), par compatibilité avec la version 15.

15.1.1 Les fonctions standard

(TYI) [SUBR à 0 argument]

lit un caractère au clavier et le retourne sous forme de code interne.

(TYS) [SUBR à 0 argument]

consulte le clavier. Si un caractère est en attente d'être lu celui ci est lu comme par la fonction TYI. Sinon la fonction retourne () immédiatement.

```
ex :  incrémenter un compteur jusqu'à la frappe d'un caractère :
      (UNTIL (TYS)
             (INCR compteur))

      TYS permet de définir TYI
      (DE TYI ()
          (LET ((c))
               (UNTIL (SETQ c (TYS)))))
```

(TYINSTRING <string>) [SUBR à 1 argument]

lit une ligne sur le clavier et stocke les caractères lus dans la chaîne <string>. Retourne en valeur le nombre de caractères lus. Une ligne plus longue que la chaîne sera tronquée à la longueur de la chaîne. Cette fonction doit être utilisée pour lire les caractères au clavier lorsque l'indicateur #:SYSTEM:LINE-MODE-FLAG est positionné (voir la définition de la fonction BOL par exemple).

(TYCN <cn>) [SUBR à 1 argument]

affiche le caractère de code interne <cn> sur l'écran. Le caractère est envoyé immédiatement sur l'écran.

(TYSTRING <string> <longueur>) [SUBR à 2 arguments]

envoie les <longueur> premiers caractères de la chaîne <string> à l'écran.

TYSTRING peut être défini en Lisp de la manière suivante :

```
(DE TYSTRING (s l)
    (FOR (i 0 1 (SUB1 (MIN (SLEN s) l)))
         (TYCN (SREF s i))))
```

(TYNEWLINE) [SUBR à 0 argument]

envoie une marque de fin de ligne sur l'écran.

(TYBACK <cn>) [SUBR à) argument]

efface le caractère de code interne <cn> qui est immédiatement avant le curseur sur l'écran et recule le curseur. Si le caractère qui précède le curseur n'est pas le caractère <cn> l'effet à l'écran est imprévisible. Cette fonction est utilisée par la fonction BOL pour effacer un caractère lorsque l'utilisateur utilise une touche d'effacement (DELETE, BACKSPACE). Le caractère passé en argument permet de faire fonctionner l'éditeur de ligne avec des caractères à espacement proportionnel.

```
ex : ; la séquence suivante laisse l'écran inchangé
```

```
(TYCN #/a)
(TYBACK #/a)
```

En général la définition suivante correspond à l'effet désiré :

```
(DE TYBACK (cn)
    (TYCN #\bs)
    (TYCN #\sp)
    (TYCN #\bs))
```

15.1.2 Les fonctions obligatoires

Les fonctions d'affichage de cette section éditent des caractères dans le canal de sortie du terminal virtuel. Ce canal est distinct du canal terminal et porte le "numéro" T. Lorsque le canal courant est le canal T, les impressions (fonctions PRIN, PRINT, PRINCN, etc.) et les émissions de caractères (fonctions TYO, TYOD, etc.) ont lieu dans le même tampon.

Le canal T est un canal de sortie au même titre que le canal (). Sa marge droite est positionnée en colonne 257, juste après la fin du tampon. Ainsi il n'y a jamais d'interruption programmable EOL sur le canal T, mais uniquement l'interruption FLUSH, lorsque la position d'écriture atteint la fin du tampon (256 caractères). La fonction TYFLUSH permet de déclencher à tout moment l'interruption FLUSH sur le canal du terminal virtuel. Le tampon est de plus vidé par le système avant chaque lecture par l'une des fonctions TYI, TYS ou TYINSTRING.

(TYO <o1> ... <oN>) [SUBR à N arguments]

édite dans le tampon du terminal virtuel, les caractères des objets <o1>...<oN>. Chaque objet <oI> peut être :

- un caractère sous la forme d'un code interne
- une chaîne de caractères
- une liste de codes internes

TYO peut être défini en Lisp de la manière suivante :

```
(DE TYO objs
    (WITH ((OUTCHAN T))
          (WHILE objs (TYO1 (NEXTL objs)))))

(DE TYO1 (o1)
    (COND ((CONSP o1) (MAPC 'PRINCN o1))
          ((STRINGP o1) (PRIN o1))
          ((FIXP o1) (PRINCN o1))))
```

(TYFLUSH) [SUBR 1 à 0 argument]

vide le tampon du terminal virtuel en produisant l'interruption programmable FLUSH sur le canal T. La fonction FLUSH, invoquée par l'interruption programmable, imprime le contenu du tampon à l'écran.

TYFLUSH peut être défini en Lisp de la manière suivante :

```
(DE TYFLUSH ()
    (WITH ((OUTCHAN T))
          (ITSOFT 'FLUSH ())))
```

(TYOD n nc) [SUBR à 2 arguments]

Edite dans le tampon du terminal virtuel la représentation du nombre <n> sur <nc> caractères en base 10.

Les fonction suivantes permettent de manipuler le terminal en mode plein-écran. Elles sont nécessaires pour utiliser les outils fournis avec le système (PEPE par exemple). Si ces fonctions doivent envoyer des caractères de contrôle au terminal (séquences d'échappement) elles doivent utiliser le canal de sortie du terminal virtuel (en utilisant la fonction TYO) pour permettre une synchronisation correcte des effets plein écran.

(TYPROLOGUE) [SUBR à 0 argument]

active le terminal en mode vidéo (pleine page). Cette fonction doit impérativement être appelée avant tout appel aux fonctions suivantes.

(TYEPILOGUE) [SUBR à 0 argument]

repasse en mode normal (rouleau).

(TYXMAX) [SUBR à 0 argument]

retourne l'indice maximal des colonnes du terminal, compté à partir de 0. En général (TYXMAX) vaut 79 (pour les terminaux qui ont 80 colonnes).
Cette fonction retourne en fait la valeur de la variable globale XMAX du package #:SYS-PACKAGE:TTY.

TYXMAX peut être défini en Lisp de la manière suivante :

```
(DE TYXMAX ()
    (SYMEVAL (GETSYMB #:SYS-PACKAGE:TTY 'XMAX ())))
```

(TYYMAX) [SUBR à 0 argument]

retourne l'indice maximal des lignes du terminal, compté à partir de 0. En général (TYYMAX) vaut 23 (pour les terminaux qui ont 24 lignes).

(TYCURSOR <x> <y>) [SUBR à 2 arguments]

positionne le curseur en colonne <x> à la ligne <y>. La position en haut à gauche de l'écran correspond aux indices 0, 0.

(TYBS <cn>) [SUBR à 1 argument]

recule d'un caractère sans rien effacer sur l'écran. L'argument précise le caractère qui précède le curseur sur l'écran. Ceci permet de reculer sur des caractères à espacement proportionnel.
En général la définition suivante produit l'effet désiré :

```
(DE TYBS (cn)
    (TYO #\bs))
```

(TYCR) [SUBR à 0 argument]

positionne le curseur au début de la ligne courante.
En général la définition suivante produit l'effet désiré :

```
(DE TYCR ()
    (TYO #\cr))
```

(TYCLS) [SUBR à 0 argument]

efface tout l'écran.

(TYBEEP) [SUBR à 0 argument]

déclenche l'alarme (clochette, buzzer, ...) sur le terminal.

(TYLEFTKEY) [SUBR à 0 argument]

retourne le code clavier associé à la touche *flèche à gauche.* Par défaut ce code est #^B. Cette fonction, ainsi que les trois fonctions qui suivent, ont une définition analogue à la fonction TYXMAX.

(TYRIGHTKEY) [SUBR à 0 argument]

retourne le code clavier associé à la touche *flèche à droite.* Par défaut ce code est #^F.

(TYUPKEY) [SUBR à 0 argument]

retourne le code clavier associé à la touche *flèche en haut.* Par défaut ce code est #^P.

(TYDOWNKEY) [SUBR à 0 argument]

retourne le code clavier associé à la touche *flèche en bas.* Par défaut ce code est #^N.

15.1.3 Les fonctions facultatives

(TYCLEOL) [SUBR à 0 argument]

efface l'écran de la position du curseur à la fin de la ligne.

(TYCLEOS) [SUBR à 0 argument]

efface l'écran de la position du curseur à la fin de l'écran.

(TYINSCN ‹cn›) [SUBR à 1 argument]

(TYINSCH ‹cn›) [SUBR à 1 argument]

insère le caractère de code interne ‹cn› à la position courante du curseur. Il n'y a absolument aucune différence entre les fonctions TYINSCN et TYINSCH.

(TYDELCN <cn>) [SUBR à 1 argument]
(TYDELCH) [SUBR à 0 argument]

détruit le caractère à la position courante du curseur. La fonction TYDELCN reçoit de plus en argument le code interne du caractère qui doit être détruit. Cette information permet de détruire les caractères à espacement proportionnel. Si le caractère présent à l'emplacement du curseur n'est pas le caractère de code interne <cn> l'effet à l'écran est imprévisible.

TYDELCN peut être défini en Lisp de la manière suivante :

```
(DE TYDELCN (cn)
    (TYDELCH))
```

(TYINSLN) [SUBR à 0 argument]

insère une nouvelle ligne à la position courante du curseur. Les lignes sous la ligne courante sont toutes décalées d'une ligne vers le bas, la dernière ligne de l'écran disparaît.

(TYDELLN) [SUBR à 0 argument]

détruit la ligne à la position courante du curseur. Les lignes sous la ligne courante sont toutes décalées d'une ligne vers le haut, la dernière ligne de l'écran est effacée.

(TYATTRIB <i>) [SUBR à 0 ou 1 argument]

si l'indicateur <i> est vrai (c'est-à-dire différent de ()), active le mode attribut, s'il est faux (c'est-à-dire égal à ()), l'enlève. Sans argument TYATTRIB retourne la valeur courante du mode attribut. Le mode attribut, qui sert à mettre en valeur un texte sur l'écran, dépend du terminal : il peut s'agir du soulignement, du clignotement (attention au mal de crâne) ou de l'affichage inversé.

Cette fonction peut être utilisée avec la structure de contrôle WITH pour modifier l'attribut localement.

```
ex :   (WITH ((TYATTRIB T))
             (TYO "Hello"))
```

```
(DE TYATTRIB x
    (IFN x
         (SYMEVAL (GETSYMB #:SYS-PACKAGE:TTY 'TYATTRIB ()))
         (FUNCALL (GETFN #:SYS-PACKAGE:TTY 'TYATTRIB ()) (CAR x))
         (SET (GETSYMB #:SYS-PACKAGE:TTY 'TYATTRIB ()) (CAR x))))
```

(TYSHOWCURSOR <i>) [SUBR à 0 ou 1 argument]

si l'indicateur <i> est vrai (c'est-à-dire différent de ()), active l'affichage du curseur sur le terminal, s'il est faux (c'est-à-dire égal à ()), l'enlève. Sans argument, TYSHOWCURSOR retourne la valeur courante de l'indicateur d'affichage du curseur.

Cette fonction peut être utilisée avec la structure de contrôle WITH pour modifier l'indicateur d'affichage du curseur localement.

```
ex :   (WITH ((TYSHOWCURSOR ()))
             (VDT))
```

(TYCO <x> <y> <cn1> ... <cnN>) [SUBR à N arguments]

correspond à l'appel :

```
(PROGN (TYCURSOR x y) (TYO cn1 ... cnN))
```

c'est-à-dire se positionne en <x>, <y> puis édite dans le tampon du terminal les codes cn1 ... cnN en utilisant la fonction TYO.

(TYCOT <x> <y> <cn1> ... <cnN>) [SUBR à N arguments]

correspond à l'appel :

```
(WITH ((TYATTRIB T))
      (TYCO x y cn1 ... cnN))
```

c'est-à-dire se positionne en <x>, <y>, passe en mode attribut, édite dans le tampon du terminal les codes cn1 ... cnN avec la fonction TYO et repasse dans le mode précédent.

15.2 Les Fonctions sur les Ecrans

Un *écran* est une matrice de caractères possédant une largeur <w> et une hauteur <h>, stockée sous la forme d'une chaîne de caractères. Les fonctions suivantes vont permettre d'une part d'envoyer les différences entre deux écrans permettant de réaliser des réaffichages asynchrones et d'autre part de déplacer un sous-écran dans un écran.

(REDISPLAYSCREEN <sn> <so> <w> <h>) [SUBR à 4 ou 12 args]

suppose que <sn> et <so> sont des chaînes de caractères contenant des contenus d'écrans, de taille <w> caractères de large et <h> caractères de haut. REDISPLAYSCREEN va envoyer sur le terminal des positionnements de curseur (TYCURSOR) et des écritures de caractères (TYO) représentant les différences entre l'ancien écran <so> et le nouvel écran <sn>. A la fin du traitement, la chaîne <so> a été physiquement modifiée et contient tous les caractères de <sn> pour pouvoir enchaîner les appels à cette fonction. Il est possible de rajouter 8 autres arguments qui ont la même signification que pour la fonction suivante.

(BLTSCREEN <sd> <ss> <w> <h>) [SUBR à 4 ou 12 args]

déplace les caractères de l'écran <ss> dans l'écran <sd>. Les tailles de ces écrans sont de <w> caractères de large et de <h> caractères de haut. Si cette fonction ne possède que 4 arguments, les tailles de ces écrans sont identiques. Si cette fonction possède 12 arguments, elle permet de déplacer un sous-écran quelconque d'un écran dans une position quelconque d'un autre écran.

Voici la signification de ces 12 arguments :
<sd> écran destination
<ss> écran source
<wd> largeur de l'écran destination
<hd> hauteur de l'écran destination
<ws> largeur de l'écran source
<hs> hauteur de l'écran source

<xd> abcisse du sous-écran destination
<yd> ordonnée du sous-écran destination
<xs> abcisse du sous-écran source
<ys> ordonnée du sous-écran source
<wt> largeur du sous-écran transféré
<ht> hauteur du sous-écran transféré

Voici une démonstration de l'utilisation conjuguée des deux fonctions manipulant des écrans :

```
(DE BLTDEMO ()
    (LET ((NSCREEN (MAKESTRING (* 80 24) #\SP))
          (OSCREEN (MAKESTRING (* 80 24) #\SP)))
         (FOR (I 0 80 240)
              (BLTSTRING NSCREEN I         "*******")
              (BLTSTRING NSCREEN (+ I 12)  "------")
              (BLTSTRING NSCREEN (+ I 24) "......"))
         (TYCLS)
         (WHILE T
                (REDISPLAYSCREEN NSCREEN OSCREEN 80 24)
                (FOR (I 0 6 30)
                     (BLTSCREEN NSCREEN NSCREEN
                          80 24 80 24
                          (RANDOM 0 80) (RANDOM 4 19)
                          I 0 6 4)))))
```

15.3 Utilisation du Terminal Virtuel

Il existe 3 programmes de démonstration du terminal virtuel :
- HANOI qui simule le jeu du même nom
- WHANOI identique avec un réaffichage asynchrone
- VDT qui simule la dure vie d'un ver-de-terre.

Ils se trouvent respectivement dans les fichiers HANOI, WHANOI et VDT de la bibliothèque standard. Leur lecture est facile et permet de bien comprendre l'utilisation du terminal simple, du terminal virtuel et du réaffichage asynchrone.

(HANOI <n>) [EXPR à 1 argument]

anime le jeu des tours de Hanoi avec <n> disques. <n> doit être un nombre compris entre 3 et 9. Cette fonction est chargée automatiquement à son premier appel.

(HANOIEND) [SUBR à 0 argument]

permet de récupérer la place occupée par les fonctions du jeu précédent.

(WHANOI <n>) [EXPR à 1 argument]

est identique à la fonction précédente mais utilise le réaffichage asynchrone REDISPLAYSCREEN.

(WHANOIEND) [SUBR à 0 argument]

permet de récupérer la place occupée par les fonctions du jeu précédent.

(VDT) [EXPR à 0 argument]

lance un jeu d'animation qui simule la croissance d'un ver de terre. Utilisez les flèches de votre terminal pour contrôler le mouvement du ver. Cette fonction est chargée automatiquement à son premier appel.
It's more fun to compete! (for amusement only).

(VDTEND) [EXPR à 0 argument]

permet, quand le travail sérieux reprend, de récupérer l'espace occupé par les fonctions du jeu précédent.

15.4 Définition d'Un Terminal Virtuel

#:SYS-PACKAGE:TTY [Variable]

Toutes les fonctions du terminal virtuel appellent en fait des fonctions dont le nom est calculé de la manière suivante :

```
(GETFN #:SYS-PACKAGE:TTY <nom de la fontion> ())
```

La fonction TYCN est, par exemple, définie de la manière suivante :

```
(DE TYCN (cn)
    (FUNCALL (GETFN #:SYS-PACKAGE:TTY 'TYCN ()) cn))
```

A l'initialisation du système la variable #:SYS-PACKAGE:TTY vaut TTY. Les fonctions réalisant l'interface standard sont donc recherchées dans le package TTY. Les fonctions de ce package (#:TTY:TYCN, #:TTY:TYI, #:TTY:TYO, etc.) sont les fonctions qui réalisent l'interface physique avec le terminal de l'utilisateur.

La définition d'un terminal virtuel devra donc comporter :
- l'affectation de la variable globale #:SYS-PACKAGE:TTY
- la définition des fonctions dans un package qui est la valeur de cette variable.

La variable #:SYS-PACKAGE:TTY doit *toujours* figurer dans un sous-package du package TTY, pour permettre l'utilisation des comportements par défaut du terminal virtuel.
Ces comportements par défaut concernent les fonctions suivantes :
- TYI, TYS, TYINSTRING : vidage du tampon du terminal avant lecture.
- TYO, TYOD, TYCO, TYCOT : utilisation du tampon du terminal virtuel.
- TYINSCH, TYDELCH, et autres fonctions facultatives : appel de la fonction TYERROR.
- TYERROR : retour de la valeur ()

- REDISPLAYSCREEN : calcul de la différence des deux écrans et envoi de codes à l'écran en utilisant les fonctions TYO et TYCURSOR.

Usuellement il suffira de redéfinir les fonctions spécifiques à un terminal donné (insertion/effacement de lignes et caractères). Voir cependant l'exemple du terminal virtuel de type #:TTY:WINDOW dans le fichier VIRBITMAP de la bibliothèque standard, qui définit des comportements particuliers pour les fonctions TYI, TYS, TYCN, TYNEWLINE et REDISPLAYSCREEN.

Voici à titre d'exemple la définition du terminal virtuel pour le terminal H19

```
; Le_Lisp version 15.2 : compilation du terminal virtuel : h19

(SETQ #:SYS-PACKAGE:TTY '#:TTY:H19)

(DEFVAR #:TTY:H19:XMAX 79)
(DEFVAR #:TTY:XMAX 79) ;   compatibilité V15

(DEFVAR #:TTY:H19:YMAX 23)
(DEFVAR #:TTY:YMAX 23) ;   compatibilité V15

(DE #:TTY:H19:TYCURSOR (col line)
    (#:TTY:TYO 27 89 (+ 32 line) (+ 32 col)))

(DE #:TTY:H19:TYCLS ()
    (#:TTY:TYO 27 69))

(DE #:TTY:H19:TYCLEOL ()
    (#:TTY:TYO 27 75))

(DE #:TTY:H19:TYCLEOS ()
    (#:TTY:TYO 27 74))

(DE #:TTY:H19:TYDELCH ()
    (#:TTY:TYO 27 78))

(DE #:TTY:H19:TYINSLN ()
    (#:TTY:TYO 27 76))

(DE #:TTY:H19:TYDELLN ()
    (#:TTY:TYO 27 77))

(DE #:TTY:H19:TYATTRIB (X)
    (IF X (#:TTY:TYO 27 112) (#:TTY:TYO 27 113)))

(DEFVAR #:TTY:H19:TYATTRIB ())
(DEFVAR #:TTY:TYATTRIB ()) ;   compatibilité V15

(DE #:TTY:H19:TYINSCH (ARG)
    (#:TTY:TYO 27 64 ARG 27 79))

(DE #:TTY:H19:TYPROLOGUE ()
    (#:TTY:H19:TYCLS))

(DE #:TTY:H19:TYEPILOGUE ()
    (TYCURSOR 0 (SUB1 #:TTY:H19:YMAX)))

(DEFVAR #:TTY:H19:TYSHOWCURSOR T)
(DEFVAR #:TTY:TYSHOWCURSOR T) ;   compatibilité V15
```

CHAPITRE 16

Les Editeurs Vidéo

16.1 PEPE

L'une des applications immédiates d'un terminal vidéo virtuel (voir le chapitre 15) est la réalisation d'un éditeur vidéo (appelé également éditeur *pleine page* (comme PEPE)). Cela est facilement réalisé en Lisp. La lecture du fichier PEPE.LL de la bibliothèque initiale vous convaincra très certainement. L'utilisation de Lisp comme langage d'implantation d'un tel éditeur permet, en plus de la facilité d'écriture et de correction, d'avoir un éditeur *extensible.*

16.1.1 Les fonctions d'appel de PEPE

PEPE [Feature]

ce trait indique si l'éditeur PEPE est chargé en mémoire.

(PEPE <f>) [FEXPR]

appelle l'éditeur sur l'objet <f> non évalué. Il existe une autre écriture plus facile pour appeler l'éditeur, le macro-caractère ^E :

^Ef correspond à l'appel (PEPE f)

Si l'objet <f> est (), l'édition reprend sur le même tampon. Si l'objet <f> est T, l'édition commence avec un nouveau tampon de nom TMP. Si l'objet est un atome, l'édition a lieu sur le fichier de même nom. Si l'objet est une liste, elle est évaluée et l'édition s'effectue dans un tampon de nom TMP qui contient tout ce qui a été imprimé par l'évaluation précédente.

; pour éditer un fichier disque de nom FOO.LL

```
^Efoo.ll
```

; pour éditer la représentation paragraphée des fonctions FOO et BAR

```
^E(pretty foo bar)
```

; pour éditer la liste triée des fonctions du système

```
^E(mapc 'print (sortl (maploblist 'typefn)))
```

(PEPEFILE <f>) [EXPR à 1 argument]

cette fonction est équivalente à la fonction précédente mais évalue son argument.

(PEPEND) [EXPR à 0 argument]

permet de récupérer l'espace occupé par les fonctions de l'éditeur. Le trait PEPE disparaît et tout appel ultérieur de PEPE rechargera l'ensemble de ces fonctions. De plus le tampon est également perdu.

16.1.2 Les Commandes de PEPE

```
^A       va au début de la ligne
^B  <-   recule d'un caractère
^C       sort de PEPE; retour par ^E
^D       détruit le caractère courant
^E       va en fin de ligne
^F  ->   avance d'un caractère
^G       annule la commande en cours
^K       détruit la ligne pointée par le curseur
^L       ré-affiche tout l'écran
^M  RC   brise la ligne à la position du curseur
^N  v    passe à la ligne suivante
^O       brise la ligne au niveau du curseur
^P  ^    passe à la ligne précédente
^S       recherche une chaîne
^V       passe à l'écran suivant
^Y       force à la position du curseur la dernière
         ligne détruite par ^K
DEL      détruit le caractère à gauche du curseur
ESC E    exécute le tampon courant
ESC F    change le nom du fichier courant
ESC I    insère un autre fichier
ESC R    lecture dans le tampon d'un nouveau fichier
ESC S    sauve le tampon courant dans le fichier courant
ESC V    passe à l'écran précédent
ESC W    écriture du tampon courant dans le fichier
ESC X    appel direct d'une fonction de PEPE
ESC Z    sauve sur disque le tampon et charge ce tampon
         en mémoire
ESC )    se positionne sur la ) fermante suivante (à la Lisp)
ESC ]    se positionne sur le ] fermant suivant (à la Lisp)
ESC >    va en fin du tampon
ESC <    va au début du tampon
ESC ?    affiche un aide mémoire de PEPE
```

16.1.3 Les extensions de PEPE

Un des avantages les plus importants de PEPE est d'être extensible. Sa simplicité lui permet d'être très aisément compris à la lecture de son code. Deux fonctions permettent d'associer à des frappes de caractères des expressions Le_Lisp.

(DEFKEY ‹n› ‹e1› ... ‹eN›) [FEXPR]

associe à la valeur de la clef ‹n› les expressions ‹e1› ... ‹eN›.

(DEFESCKEY ‹n› ‹e1› ... ‹eN›) [FEXPR]

associe à la valeur de la suite des clefs ‹ESC› ‹n› les expressions ‹e1› ... ‹eN›.

16.2 Emacs

Cet éditeur qui fonctionnait avec la version 14 n'a pas encore été remis en service officiellement pour la version 15.2 mais des versions expérimentales circulent.

Voici à titre de rappel les commandes de cet éditeur.

```
@           kill-to-beginning-of-line
^A          go-to-beginning-of-line
^B          backward-char
^C          re-execute-command
^D          delete-char
^E          go-to-end-of-line
^F          forward-char
^G          command-quit
^H          rubout-char
^I          tab-command
^J          null
^K          kill-lines
^L          redisplay-command
^M          new-line
^N          next-line-command
^O          open-space
^P          prev-line-command
^Q          quote-char
^R          reverse-string-search
^S          string-search
^T          twiddle-chars
^V          next-screen
^Y          yank
‹del›       rubout-char
‹esc› ›     go-to-end-of-buffer
‹esc› ‹     go-to-beginning-of-buffer
‹esc› #     rubout-word
‹esc› %     query-replace
```

```
<esc> ^     delete-line-indentation
<esc> ^A    begin-defun
<esc> ^E    end-defun
<esc> ^F    forward-sexp
<esc> ^O    split-line
<esc> ^R    move-defun-to-screen-top
<esc> ^Z    eval-top-level-form
<esc> B     backward-word
<esc> C     capitalize-initial-word
<esc> D     delete-word
<esc> F     forward-word
<esc> G     go-to-line-number
<esc> L     lower-case-word
<esc> R     move-to-screen-edge
<esc> U     upper-case-word
<esc> V     previous-screen
<esc> X     extended-command
<esc> <del>rubout-word
<esc> ~     unmodify-buffer
<esc> <esc>eval-lisp-line
^X (        begin-macro-collection
^X )        end-macro-collection
^X *        show-last-or-current-macro
^X =        linecounter
^X ^C       quit-the-editor
^X ^F       find-file
^X ^R       read-file
^X ^S       save-file
^X ^T       toggle-redisplay
^X ^W       write-file
^X E        execute-last-editor-macro
^X I        insert-file
^X Q        macro-query
^Z ^Y       show-kills

**** les commandes etendues ****

          eval-loop
          eval-buffer
          eval-lisp-line
          french-mode
          rem-char-in-word
          add-char-in-word
          assemble-buffer
          show-value
          edit-value
          edit-print
          insert-function
          edit-function
          look-function
          make-wall-chart
          print-wall-chart
          print-key
```

set-permanent-key
set-tab-pos

CHAPITRE 17

L'éditeur de Ligne en Entrée Clavier

Olivier Guillaumin

Edlin est un éditeur vidéo de lignes de commandes, activable uniquement à partir d'un clavier, et donc en mode interactif. Son but est de vous éviter au maximum de retaper intégralement des commandes parfois longues ou compliquées, de vous donner toutes les possibilités d'édition sur les lignes que vous entrez, ainsi qu'une aide parfois précieuse sous forme de recherches automatiques de symboles, fonctions, paramètres de fonctions, etc.

Edlin n'est pas dans sa version actuelle un éditeur plein écran, il opère exclusivement sur une ligne, sans utiliser les fonctions d'adressage du curseur, et en laissant l'écran défiler le cas échéant. Les primitives du terminal virtuel qu'il utilise sont l'insertion et suppression de caractères, l'effacement de fin de ligne, et les codes ASCII de déplacement du curseur Backspace, Carriage-return, et Line-feed. Certains terminaux (en particulier VT100) ne disposant pas de toutes ces primitives font l'objet d'une version spécifique quelque peu modifiée d'Edlin.

Les fonctions qui vont être décrites se trouvent dans un fichier de la bibliothèque standard de nom EDLIN

17.1 Chargement d'Edlin

EDLIN [Feature]

ce trait indique que l'éditeur de ligne EDLIN est chargé en mémoire.

(EDLIN) [EXPR à 0 argument]

charge, s'il ne l'était pas l'éditeur EDLIN.

Et on dispose immédiatement des ressources de l'éditeur qui supervise dès lors le lecteur Le_Lisp. Il est ainsi possible de conserver les possibilités d'édition tout au long de la session, quelle que soit l'application qui fait les lectures.

Edlin dispose d'un aide-mémoire en ligne pour ses commandes, que l'on obtient en tapant successivement la touche escape (notée dorénavant ESC) et la touche ? (/ pour les anglicistes).

17.2 Les Commandes d'Edition

Les commandes d'édition d'Edlin sont les mêmes que celles d'emacs, ou pepe :

^F (forward) déplace le curseur d'un caractère vers la droite.

ESC f place le curseur à la fin du mot.

^B (backward) déplace le curseur d'un caractère vers la gauche.

ESC b place le curseur au début du mot.

^A met le curseur sur le premier caractère de la ligne.

^E (end) positionne le curseur derrière le dernier caractère de la ligne.

^D (delete) supprime le caractère sous le curseur.

ESC d ou ESC ^D supprime la fin du mot.

^H ^? (backspace rubout) suppriment le caractère à gauche du curseur.

ESC ^H ou ESC ^? supprime le mot précédant le curseur.

^K (kill) supprime la fin de la ligne (curseur compris).

^X ou ^U supprime tous les caractères situés à gauche du curseur.

^T (transpose) échange les deux caractères situés à gauche du curseur.

ESC (positionne le curseur sur la parenthèse ouvrante correspondante.

ESC) fait de même pour la parenthèse fermante.

ESC ' insère le caractère ' (quote) devant le symbole courant.

Les caractères alphanumériques sont insérés à l'emplacement du curseur.

De plus toutes ces commandes, ainsi que d'autres qui vont être explicitées par la suite peuvent être affectées d'un coefficient multiplicatif quelconque : il suffit de taper ESC suivi du nombre suivi de la commande.

17.3 Les Commandes de Rappel

Edlin conserve dans une variable interne la liste de toutes les lignes différentes entrées au terminal. Des commandes spécifiques permettent de manipuler cette liste :

ESC h (history) donne la liste des lignes entrées en ordre chronologique décroissant.

TAB ou ^I insère successivement à la position du curseur les différentes lignes dans l'ordre de la liste précédente.

^N permet de redescendre chronologiquement suivant le même principe (la commande insérée est remplacée par la suivante jusqu'à ce que le curseur bouge).

^W permet d'éviter que la liste des commandes conservées ne s'étende démesurément (il garde tout de même les 100 dernières).

ESC ESC cherche parmi les lignes connues celles qui contiennent les caractères situés à gauche du curseur.

17.4 Les Commandes de Recherche

On a déjà vu le résumé en ligne qui s'obtient par ESC ? : en fait c'est cette même séquence qui sert dans d'autres cas de recherche :

ESC ? sur une ligne vide donne le résumé, et derrière un mot cherche dans l'oblist tous les symboles contenant son pname (lhoblist). Chaque symbole trouvé est explicité sous un format clair à l'écran.

ESC / réalise le même genre de recherche avec la restriction que les symboles doivent commencer par le mot courant.

ESC p donne en plus la P-list des symboles trouvés.

si le mot courant est spécifié avec un package, la recherche sera limitée à ce package, en particulier s'il est de la forme #:package on obtiendra la liste de tous les symboles du package. La liste est triée par ordre de packages de profondeur croissante et par ordre alphabétique avant impression.

ESC SP (espace) essaie de compléter le mot courant avec les règles de ESC / en un symbole, les différentes possibilités sont proposées successivement.

17.5 Les Autres Commandes

**** (backslash) est le caractère de forc/age pour insérer les caractères ayant une signification pour Edlin. Il faut noter que les macro-caractères ^D ^P ^E ^R sont gérés par Edlin qui a tendance à les insérer effectivement quand ils sont tapés sur une ligne vide.

^ est le caractère permettant de quitter Edlin (il doit être précédé du caractère de forc/age ^V sur les systèmes supportant UN*X et csh)

^Z est un caractère d'échappement vers le shell, son comportement varie avec le système hôte.

**ESC ** permet de sortir de l'évaluation courante (par exemple en cours de saisie pour revenir directement au toplevel) c'est plus propre que break pour un résultat identique.

ESC ^M (return) permet de fermer toutes les parenthèses ouvertes et lance l'évaluation.

^M ou ^J (return line-feed) valident la ligne courante, où qu'ils soient tapés (et non pas seulement en fin de ligne).

Le plus sûr moyen d'apprécier, c'est d'essayer...

CHAPITRE 18

Le Fenêtrage Virtuel

Matthieu Devin
Laurence Gallot
Jean-Marie Hullot

Ce chapitre décrit le système de fenêtrage virtuel disponible sous Le_Lisp. L'interface avec un dispositif de pointage (souris, tablette, crayon optique, ...) est décrit dans le chapitre 19. Les primitives graphiques portables sont décrites dans le chapitre 20. Les menus portables ne sont pas encore décrits.

Le système de fenêtrage est normalement relié à un dispositif d'affichage haute résolution (bitmap) mais fonctionne également sur un terminal alphanumérique classique.

Cette interface permet de réaliser un environnement de programmation multi-fenêtres *portable,* qui peut être utilisé sur tous les systèmes Le_Lisp possédant un écran haute résolution et/ou un dispositif de pointage.

Ce système a déjà été implanté sur :
- SM90 avec écran Numelec et ASH.
- SUN avec ASH et suntools.
- SPS9 avec écran monochrome et système ASH ou le gestionnaire de fenêtres de ROS.
- Apple MacIntosh

Il est en cours de réalisation sur :
- Vax Station II, sous MicroVMS.
- CADMUS

Une version fonctionnant sur terminaux alphanumériques classiques, en appui sur le terminal virtuel, est aussi disponible. Cette version a des possibilités très limitées en raison de sa faible résolution; elle n'est intéressante que pour vérifier la portabilité des logiciels multi-fenêtres.

18.1 Avertissement

Le système de fenêtrage virtuel décrit dans ce chapitre permet uniquement l'affichage de texte dans les fenêtres. Cette restriction permet l'utilisation de l'environnement multi-fenêtres sur des terminaux alphanumériques classiques.

Les fonctions qui permettent d'afficher du texte font partie des primitives graphiques du système, décrites au chapitre 20. Le chapitre 20 donne une description propre et complète du système graphique et de ses relations avec le système de fenêtrage.

18.2 Implantation du Fenêtrage Virtuel

La gestion de l'écran haute résolution est bâtie sur le même modèle que la gestion du terminal virtuel : les fonctions indépendantes des écrans (génériques) recherchent et invoquent, à chaque appel, les fonctions spécifiques à l'écran utilisé. Ces fonctions spécifiques ont le même nom que les fonctions génériques, mais résident dans un package spécialisé, dépendant de l'écran effectivement utilisé. La variable globale #:SYS-PACKAGE:BITMAP indique dans quel package Lisp sont définies les fonctions de gestion de l'écran connecté au système. Ce package doit être un sous-package du package BITMAP.

Les fonctions génériques sont toutes définies sur le modèle de la fonction BITPROLOGUE :

```
(DE BITPROLOGUE ()
    (FUNCALL (GETFN #:SYS-PACKAGE:BITMAP 'BITPROLOGUE)))
```

Les fonctions spécifiques à l'écran SUNVIEW, par exemple, sont définies dans le package #:BITMAP:SUNVIEW.

```
(DE #:BITMAP:SUNVIEW:BITPROLOGUE ()
    ...)
```

#:SYS-PACKAGE:BITMAP [Variable Globale]

Contient le package dans lequel sont définies les fonctions de gestion de l'écran connecté au système.

#:BITMAP:NAME [Variable Globale]

Contient le nom symbolique du gestionnaire d'écran utilisé pour l'implantation du fenêtrage virtuel. Par exemple ASH sur une SM90 ou un SUN si l'on utilise ASH, SUNVIEW sur un SUN si l'on utilise SUNVIEW.

WINDOW [Feature]

Le trait WINDOW indique que le fenêtrage virtuel a été chargé dans le système. Un appel de la fonction CURRENT-WINDOW sans argument (voir cette fonction) permet de savoir si le fenêtrage est actif, c'est-à-dire si l'utilisateur travaille effectivement dans une fenêtre.

(INIBITMAP <symb>) [SUBR à 0 ou 1 argument]

Cette fonction permet de charger le fichier de définition des fonctions spécifiques à l'écran dont le nom est passé en argument. Ce fichier a pour nom <symb> suffixé par l'extension Lisp #:SYSTEM:LELISP-EXTENSION. Il est recherché dans le catalogue #:SYSTEM:VIRBITMAP-DIRECTORY. Si l'argument n'est pas fourni il prend pour valeur la valeur de la variable d'environnement BITMAP (voir fonction GETENV) si cette variable d'environnement est définie, VIRTTY sinon. Dans ce dernier cas c'est l'implantation du fenêtrage virtuel sur terminal virtuel qui est chargée.

Le chargement du fichier doit positionner la variable globale #:SYS-PACKAGE:BITMAP. La fonction INIBITMAP positionne la variable globale #:BITMAP:NAME et ajoute le trait WINDOW au système (voir fonction ADD-FEATURE).

18.3 Les Fonctions de l'Ecran Haute Résolution

(BITXMAX) [SUBR à 0 argument]
(BITYMAX) [SUBR à 0 argument]

Ces fonctions retournent les dimensions du dispositif d'affichage haute résolution. BITXMAX est la largeur de l'écran moins un, BITYMAX la hauteur moins un. L'unité employée est le point élémentaire affichable par le dispositif.

Exemple : Pour un écran NUMELEC (sur une SM90) ces fonctions retournent les valeurs suivantes :

```
(BITXMAX) --> 1023
(BITYMAX) --> 779
```

(BITPROLOGUE) [SUBR à 0 argument]

BITPROLOGUE initialise l'écran haute résolution attaché au système. Cette fonction est automatiquement appelée lors de la création de la première fenêtre si elle n'a pas été appelée avant.

(BITEPILOGUE) [SUBR à 0 argument]

BITEPILOGUE termine une session de travail sur l'écran haute résolution. On doit l'appeler avant de sortir du système pour terminer proprement l'utilisation de l'écran.

(BITMAP-REFRESH) [SUBR à 0 argument]

Cette fonction réaffiche le contenu des fenêtres Lisp sur l'écran haute résolution. Elle est utile pour nettoyer l'écran des scories que peuvent y afficher des programmes extérieurs au système Le_Lisp (messages du système hôte, traces de programmes tournant parallèlement au système Le_Lisp, ...).

18.3.1 Coordonnées globales

On appelle *coordonnées bitmap globales* un couple d'entiers (x, y) permettant de repérer un point sur l'écran haute résolution.

Les origines de ce système de coordonnées sont définies comme suit :
- en haut à gauche : x=0, y=0
- en bas à gauche : x=0, y=(BITYMAX)
- en haut à droite : x=(BITXMAX), y=0
- en bas à droite : x=(BITXMAX), y=(BITYMAX)

18.4 Les Fenêtres

Les fenêtres sont des objets structurés typés (voir les Structures), qui ont des représentations graphiques sur l'écran haute résolution, sous la forme de cadres rectangulaires munis d'un titre. Un environnement graphique (voir le graphique virtuel) est attaché à chaque fenêtre. Il permet diverses opérations d'affichage dans la fenêtre : texte, lignes, zones.

Certains systèmes, comme le fenêtrage virtuel sur terminal virtuel, ne disposent pas du graphique virtuel. Pour ces systèmes un environnement graphique minimum, permettant uniquement l'affichage de texte est néanmoins présent. C'est cet environnement graphique minimum qui est décrit dans ce chapitre, l'environnement graphique complet est décrit au chapitre 20.

Le système connaît un environnement graphique courant sur lequel agissent les primitives d'affichage. La fenêtre courante est la fenêtre attachée à l'environnement graphique courant.

A l'initialisation du système de fenêtrage, c'est-à-dire après l'appel de la fonction BITPROLOGUE, la fenêtre courante peut être :
- Une fenêtre de dialogue, ou Lisp-Listener (cf Sec. 15.5) comme sur le MacIntosh ou le SUN avec SUNVIEW.
- Une fenêtre recouvrant tout l'écran haute résolution attaché au système, le dialogue avec l'interprète ayant lieu sur cet écran ou bien sur un terminal alphanumérique classique relié au système, comme sur SM90 et SPS9 avec ASH.
- La fenêtre spéciale (), qui indique que les fonctions d'affichage de texte sont sans effet, comme pour le multifenêtrage sur écran alphanumérique. Ceci permet de désarmer les fonctions spéciales d'affichage de texte lorsque le multifenêtrage n'est pas utilisé.

18.4.1 Coordonnées locales

L'environnement graphique attaché à une fenêtre y définit un système de coordonnées locales. Dans la définition actuelle du fenêtrage virtuel ce système de coordonnées locales est figé. Dans une version future il sera possible de le redéfinir.

Le système de coordonnées locales est défini comme suit :
- coin supérieur gauche de la fenêtre : x=0, y=0
- coin inférieur droit de la fenêtre : x=hauteur fenêtre moins un, y=largeur fenêtre moins un.

Les points du cadre et du titre de la fenêtre sont à l'extérieur de la fenêtre mais peuvent être représentés dans ce système de coordonnées. Le point de coordonnées locales (-1, -1) est par exemple un point du cadre de la fenêtre.

18.4.2 Messages

Le fenêtrage virtuel est implanté selon une méthodologie orientée objet : les fenêtres sont des objets structurés, de type WINDOW.

Ces objets structurés peuvent recevoir des *messages* (voir les structures) :
- Les messages CURRENT-WINDOW et UNCURRENT-WINDOW permettent de gérer la fenêtre courante;
- Les messages MODIFY-WINDOW, KILL-WINDOW, POP-WINDOW et MOVE-BEHIND-WINDOW mettent à jour la représentation de la fenêtre sur l'écran.

Les fonctions CURRENT-WINDOW, MODIFY-WINDOW, KILL-WINDOW, POP-WINDOW et MOVE-BEHIND-WINDOW permettent de transmettre ces messages aux fenêtres.

Cette implantation des fenêtres permet de réaliser très facilement des extensions au multi-fenêtrage virtuel. Voir la section 15.5 par exemple.

18.4.3 Création des fenêtres

(CREATE-WINDOW <type> <left> <top> <wth> <hgt> <ti> <hi> <vi>) [SUBR 8]

Cette fonction crée une fenêtre.
<type> est le type de la fenêtre. Les seuls types existant au lancement du système sont les types WINDOW et #:WINDOW:TTY.
<left> et <top> sont des nombres qui indiquent les coordonnées du coin supérieur gauche de la fenêtre.
<wth> et <hgt> sont des nombres qui indiquent la largeur et la hauteur de la zone utile de la fenêtre (hors cadre et titre) en nombre de points sur le dispositif d'affichage.
<ti> est une chaîne de caractères qui devient le titre de la fenêtre.
<hi> est un nombre indiquant le mode de mise en valeur de la fenêtre. Si cet argument vaut 0 la fenêtre n'est pas mise en valeur. S'il est différent de 0 sa valeur indique de quelle manière la fenêtre est mise en valeur (titre rehaussé, fenêtre colorée, etc.). Les différents modes de mise en valeur d'une fenêtre dépendent des systèmes; en standard un seul mode de mise en valeur existe, de code 1.
<vi> est un petit entier indiquant le degré de visibilité de la fenêtre sur l'écran. Si cet indicateur vaut 0 la fenêtre est invisible. Si l'indicateur est différent de 0, sa valeur indique son degré de visibilité (entièrement visible, semi-visible, transparente, etc.). Les différents degrés de visibilité sont dépendants des systèmes; en standard un seul degré de visibilité existe de code 1. Le degré de visibilité 1 indique que la fenêtre cache les fenêtres qui sont derrière elle.
La fenêtre est créée au premier plan de l'écran. Elle est éventuellement affichée si la valeur de <vi> est différente de 0. Elle est positionnée de telle sorte que le point de coordonnées locales (0, 0) soit au point de coordonnées globales (<top>, <left>).

Cette fonction ne modifie pas la fenêtre courante. La valeur rendue par cette

fonction est un objet structuré de type <type>.

```
ex : ? (CREATE-WINDOW 'window 10 10 400 400 "Lisp" 0 1)
     = #<WINDOW Lisp>
```

La fonction CREATE-WINDOW utilise la fonction de création spécifique aux objets de type <type>.

peut être défini en Lisp de la manière suivante :

```
(DE CREATE-WINDOW (type le to wi he ti hi vi)
    (LET ((create (GETFN1 type 'create ())))
         (UNLESS create
                 (ERROR 'CREATE-WINDOW 'ERROOB type))
         (FUNCALL create le to wi he ti hi vi)))
```

La composition exacte des structures de type WINDOW, peut varier d'un système à l'autre. Sur tous les systèmes néanmoins, le type WINDOW comporte les champs *left, top, width, height, title, hilighted, visible* et *extend.* L'usage du champ extend est réservé à l'implantation du fenêtrage virtuel. Il ne doit pas être utilisé hors de ce cadre.

On peut lire uniquement le contenu des champs avec les fonctions suivantes :

(#:WINDOW:LEFT <win>) [SUBR à 1 argument]
(#:WINDOW:TOP <win>) [SUBR à 1 argument]
(#:WINDOW:WIDTH <win>) [SUBR à 1 argument]
(#:WINDOW:HEIGHT <win>) [SUBR à 1 argument]
(#:WINDOW:TITLE <win>) [SUBR à 1 argument]
(#:WINDOW:HILITED <win>) [SUBR à 1 argument]
(#:WINDOW:VISIBLE <win>) [SUBR à 1 argument]
(#:WINDOW:EXTEND <win>) [SUBR à 1 argument]

Ces fonctions retournent en valeur le contenu des champs de la fenêtre argument.

18.4.4 Les fonctions sur les fenêtres

Les fonctions qui suivent permettent d'agir sur les fenêtres créées par la fonction CREATE-WINDOW. Le rôle de ces fonctions est uniquement de transmettre un message à la fenêtre argument : ce sont les fonctions primitives sur les fenêtres, décrites dans la section suivante, qui agissent réellement sur les fenêtres. En général on utilisera uniquement les fonctions de cette section.

(CURRENT-WINDOW [<win>]) [SUBR à 0 ou 1 argument]

La fenêtre argument devient la fenêtre courante et son environnement graphique devient l'environnement graphique courant. Toutes les opérations graphiques ont donc désormais lieu dans l'environnement graphique attaché à la fenêtre <win>. Si l'argument n'est pas fourni, CURRENT-WINDOW retourne la fenêtre courante.

Si l'argument est (), la fenêtre courante devient () : les opérations d'affichage sont alors sans effet.

Si l'argument est une fenêtre qui a été tuée le système déclenche l'erreur ERRNWA de libellé :

```
** CURRENT-WINDOW : L'argument n'est pas une fenêtre : xxxx
```

Dans tous les cas où l'argument est fourni, on envoie le message UNCURRENT-WINDOW à la fenêtre courante précédente avant de changer de fenêtre courante.

Cette fonction permet d'utiliser la structure de contrôle WITH pour sélectionner temporairement une fenêtre. Voici par exemple un moyen de lancer l'éditeur PEPE dans une fenêtre :

```
(WITH ((CURRENT-WINDOW PEPEWINDOW))
      (PEPE ()))
```

(MODIFY-WINDOW <win> <left> <top> <wth> <hgt> <ti> <hi> <vi>) [SUBR 8]

L'argument <win> est une fenêtre, précédemment créée par la fonction CREATE-WINDOW. Les arguments <left>, <top>, <wth>, <hgt>, <ti>, <hi>, et <vi> sont similaires aux arguments de même nom de la fonction CREATE-WINDOW. La fonction MODIFY-WINDOW permet de modifier les paramètres de définition de la fenêtre argument : c'est cette fonction qui permet de déplacer une fenêtre, d'en changer la taille ou le titre, de la mettre en valeur, de la rendre visible ou invisible. MODIFY-WINDOW retourne la fenêtre modifiée en valeur. Par convention, si l'un des arguments <left>, <top>, <wth>, <hgt>, <ti>, <hi> ou <vi> a pour valeur (), le paramètre correspondant n'est pas modifié.

Exemples : Création d'une fenêtre de nom Lisp et d'une fenêtre de nom Foo :

```
(DEFVAR wlisp (CREATE-WINDOW 'WINDOW 10 10 400 400 "Lisp" 0 1)
        wfoo  (CREATE-WINDOW 'WINDOW 30 30 500 500 "Foo"   0 1))
```

Déplacement de la fenêtre wlisp :

```
(MODIFY-WINDOW wlisp 20 20 () () () () ())
```

Diminution de la taille de la fenêtre de nom Foo :

```
(MODIFY-WINDOW wfoo () () 50 100 () () ())
```

Modification du titre de la fenêtre de nom Lisp :

```
(MODIFY-WINDOW wlisp () () () () "Le Nouveau Nom" () ())
```

Rehaussement du titre de la fenêtre wfoo :

```
(MODIFY-WINDOW wfoo () () () () () 1 ())
```

Effacement de l'écran de la fenêtre wlisp (la fenêtre est rendue invisible) :

```
(MODIFY-WINDOW wlisp () () () () () () 0)
```

Réapparition de cette même fenêtre :

```
(MODIFY-WINDOW wlisp () () () () () () 1)
```

(KILL-WINDOW <win>)

Supprime la fenêtre passée en argument. La fenêtre est effacée du dispositif d'affichage si elle était visible. Toute utilisation ultérieure de la fenêtre window provoquera l'erreur ERRNWA.

Si lors de ce processus on tue la fenêtre courante, la fenêtre courante redevient la fenêtre (). La fenêtre courante reçoit le message UNCURRENT-WINDOW avant d'être tuée.

Les deux fonctions suivantes permettent de gérer l'ordre d'affichage des fenêtres sur l'écran.

(POP-WINDOW <win>) [SUBR à 1 argument]

Ramène la fenêtre <win> au premier plan de l'écran.

(MOVE-BEHIND-WINDOW <win1> <win2>) [SUBR à 1 argument]

positionne la fenêtre <win1> juste derrière la fenêtre <win2> sur l'écran. La position de <win2> n'est pas modifiée.

Les deux fonctions suivantes permettent de faire le lien entre un dispositif de pointage sur l'écran et les fenêtres.

(FIND-WINDOW <x> <y>) [SUBR à 2 arguments]

Cette fonction retourne en valeur la fenêtre Lisp à laquelle appartient le point de coordonnées globales (<x>, <y>) sur l'écran. Si le point n'appartient à aucune fenêtre cette fonction retourne ().

Les points du cadre et du titre de la fenêtre sont supposés appartenir à la fenêtre pour cette opération : si le point désigne le titre d'une fenêtre FIND-WINDOW retournera cette fenêtre.

(MAP-WINDOW <win> <x> <y> <symbx> <symby>) [SUBR à 5 arg.]

<win> est une fenêtre.
<X> et <y> sont deux nombres désignant un point en coordonnées globales sur l'écran.
<Symbx> et <symby> sont deux symboles. Cette fonction permet de calculer les coordonnées locales à la fenêtre <win> du point (<x>, <y>). Au retour de cette fonction les symboles <symbx> et <symby> ont pour valeur les coordonnées locales du point dans la fenêtre.

Cette fonction rend des résultats significatifs uniquement pour les points (<x>, <y>) qui sont dans la fenêtre, dans son cadre ou dans son titre. Pour les points du cadre et du titre les coordonnées locales rendues sont simplement en dehors du rectangle intérieur de la fenêtre.

```
ex : ? (DEFVAR w (CREATE-WINDOW 'WINDOW 10 10 400 400 "w" 0 1)
     ?          x 0
     ?          y 0)
     = 0
     ? (EQ w (FIND-WINDOW 200 200))
     = T
     ? (MAP-WINDOW w 200 200 'x 'y)
     ? x
     = 190
     ? y
     = 190
```

18.4.5 Les fonctions primitives sur les fenêtres

Ces fonctions sont les fonctions primitives du multi-fenêtrage virtuel. Ce sont les méthodes des objets de type WINDOW pour les messages CURRENT-WINDOW, UNCURRENT-WINDOW, MODIFY-WINDOW, KILL-WINDOW, MOVE-BEHIND-WINDOW et MAP-WINDOW.

Elles ne seront à priori jamais utilisées directement par l'utilisateur et sont ici à titre de documentation.

(#:WINDOW:CURRENT-WINDOW <win>) [SUBR à 1 arg.]
(#:WINDOW:UNCURRENT-WINDOW <win>) [SUBR à 1 arg.]
(#:WINDOW:MODIFY-WINDOW <win> <left> <top> <w> <h> <ti> <hi> <vi>) [SUBR 8]
(#:WINDOW:KILL-WINDOW <win>) [SUBR à 1 arg.]
(#:WINDOW:POP-WINDOW <win>) [SUBR à 1 arg.]
(#:WINDOW:MOVE-BEHIND-WINDOW <win1> <win2>) [SUBR à 1 arg.]
(#:WINDOW:MAP-WINDOW <win> <x> <y> <symbx> <symby>) [SUBR à 5 arg.]

18.4.6 L'environnement graphique minimum

Les quatre fonctions ci-dessous sont les fonctions existant sur toutes les implantations du fenêtrage virtuel. Elle forment un ensemble minimum de fonctions nécessaires pour construire des fenêtres qui émulent des terminaux.

(CLEAR-GRAPH-ENV) [SUBR à 0 argument]

Efface l'environnement graphique courant.

(DRAW-CURSOR <x> <y> <i>) [SUBR à 3 arguments]

Cette fonction permet d'afficher et d'effacer un curseur dans l'environnement graphique courant au point de coordonnées (<x>, <y>). <i> est un indicateur qui peut prendre les valeurs () ou t : si <i> vaut () le curseur est effacé, sinon il est affiché. Le curseur affiché dépend bien sûr des systèmes.

Exemple : Ce curseur est généralement utilisé pour avertir l'utilisateur qu'il doit donner un caractère au système. Voici la manière standard de lire un caractère au clavier et d'en faire l'écho à une position (x, y) passée en argument :

```
(DE WTYI-ECHO (x y)
    (DRAW-CURSOR x y t)
    (LET ((c (TYI)))
         (DRAW-CURSOR x y ())
         (DRAW-CN x y c)))
```

(DRAW-CN <x> <y> <cn>) [SUBR à 3 arguments]

Ecrit le caractère de code interne <cn> à la position <x>, <y> dans l'environnement graphique courant.

DRAW-CN peut être défini en Lisp de la manière suivante :

```
(DE DRAW-CN (x y cn)
    (DRAW-SUBSTRING x y (STRING (ASCII cn)) 0 1)
```

(DRAW-STRING <x> <y> <s>) [MACRO à 3 arguments]
(DRAW-SUBSTRING <x> <y> <s> <start> <length>) [SUBR à 5 arg.]

La fonction DRAW-SUBSTRING écrit <length> caractères de la chaîne <s> pris à partir de la position <start> au point de coordonnées <x>, <y> dans l'environnement graphique courant.

La macro DRAW-STRING permet d'écrire la chaîne <s> complète au point <x>, <y>.

Exemple : Ecrire la chaîne "otheb" au point de coordonnées locales 10 10 dans la fenêtre courante.

```
(DRAW-SUBSTRING 10 10 "foothebar" 2 5)
```

DRAW-STRING peut être défini en Lisp de la manière suivante :

```
(DMD DRAW-STRING (x y s)
     `(LET ((s ,s))
           (DRAW-SUBSTRING ,x ,y s 0 (slength s))))
```

18.4.7 Les chaînes de caractères

Sur l'écran, une chaîne de caractères est caractérisée par un rectangle d'affichage, un point de base et un point d'incrément.

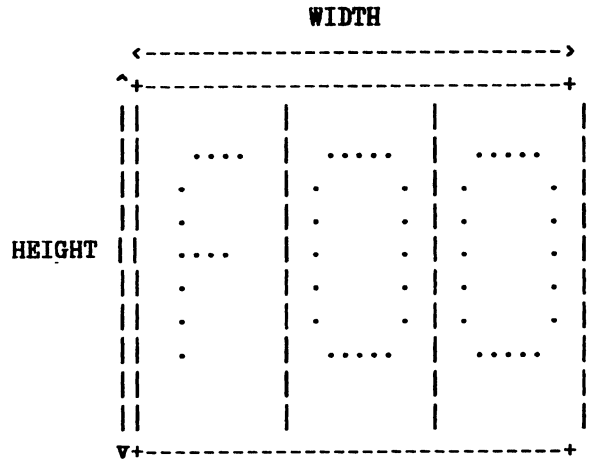
Le rectangle d'affichage est la zone occupée par la chaîne sur l'écran.

Le point de base est le point du rectangle d'affichage qui est positionné au point <x>, <y> donné en paramètre à la fonction DRAW-STRING.

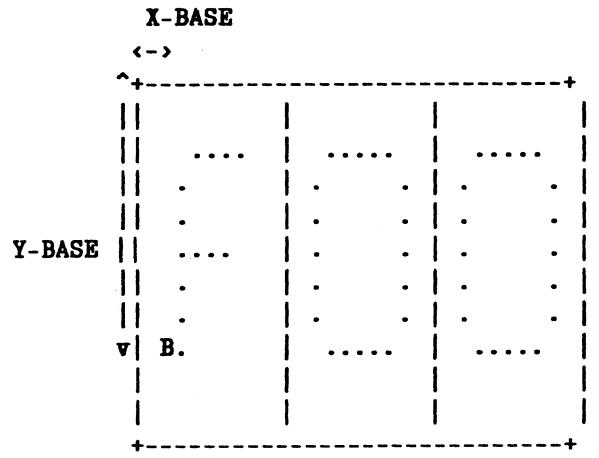
Le point d'incrément est un point du rectangle d'affichage qui doit servir de point de base pour l'affichage d'une nouvelle chaîne à la suite de la chaîne affichée.

Les schémas suivants permettent de visualiser ces différentes quantités.

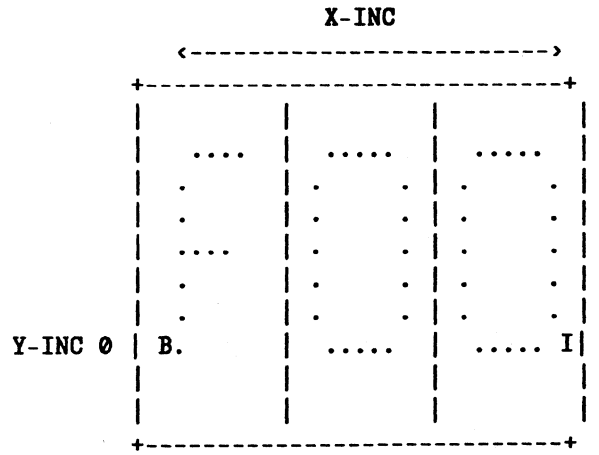
Le rectangle d'affichage (largeur et hauteur) :



Le point de base (coordonnées locales au rectangle d'affichage) :



Le point d'incrément (relativement au point de base) : ici le point d'incrément et le point de base ont la même coordonnée Y. La valeur de Y-INC est donc 0.



Ces six quantités caractéristiques d'une chaîne dépendent évidemment de l'attribut d'affichage (corps, graisse, police, ...). Elles peuvent être calculées par les fonctions suivantes.

(WIDTH-SUBSTRING <string> <start> <length>)
(HEIGHT-SUBSTRING <string> <start> <length>)
(X-BASE-SUBSTRING <string> <start> <length>)
(Y-BASE-SUBSTRING <string> <start> <length>)
(X-INC-SUBSTRING <string> <start> <length>)
(Y-INC-SUBSTRING <string> <start> <length>)

Ces fonctions retournent diverses informations sur la taille de la représentation d'une chaîne sur l'écran.

Exemples :

La fonction suivante permet par exemple de déterminer si le point de coordonnées <px>, <py> est dans la représentation de la chaîne "foo" en <x>, <y>.

```
(DE IN-STRINGP (px py s x y)
    (LET ((lx (- x (x-base-substring s 0 (slength s))))
          (ty (- y (y-base-substring s 0 (slength s)))))
         (AND (> px lx)
              (> py ty)
              (< (- px lx) (width-substring s 0 (slength s)))
              (< (- py ty) (height-substring s 0 (slength s))))))
```

La fonction suivante permet d'afficher n chaînes, à la suite les unes des autres sur l'écran.

```
(DE DRAW-N-STRINGS (x y string . strings)
    (DRAW-STRING x y string 1)
    (WHILE strings
       (DRAW-STRING
           (INCR x (X-INC-SUBSTRING string 0 (slength string)))
           (INCR y (Y-INC-SUBSTRING string 0 (slength string)))
           (NEXTL strings string))))
```

(WIDTH-SPACE) [SUBR à 0 argument]
(HEIGHT-SPACE) [SUBR à 0 argument]

Ces deux fonctions permettent de calculer la largeur et la hauteur du caractère d'espacement. Elles sont particulièrement utiles pour la manipulation de texte en chasse fixe.

WIDTH-SPACE peut être défini en Lisp de la manière suivante :

```
(DE WIDTH-SPACE ()
    (WIDTH-SUBSTRING " " 0 1))
```

18.5 Définition de Nouveaux Types de Fenêtres

L'implantation *orientée objet* du fenêtrage virtuel rend son extension aisée. Il est ainsi possible de définir de nouveaux types de fenêtres ayant des comportements spécialisés. Nous expliquons ici comment définir des fenêtres émulant des terminaux alphanumériques. L'utilisateur désirant étendre le fenêtrage virtuel s'inspirera de cette description. Le code Lisp correspondant est dans le fichier VIRBITMAP.LL de la bibliothèque Le_Lisp.

18.5.1 Spécifications

Nous définissons ici le type de fenêtres #:WINDOW:TTY. Ces fenêtres émulent un terminal alphanumérique de la manière suivante :

- lors de la sélection d'une telle fenêtre toutes les impressions Lisp sur le canal terminal et sur le canal TTY sont dirigées vers cette fenêtre.
- ces fenêtres gèrent un curseur.
- ces fenêtres sont capables de réaliser toutes le opérations plein écran du terminal virtuel.
- ces fenêtres modifient automatiquement les paramètres XMAX, YMAX et RMARGIN lorsque leur taille est modifiée.

18.5.2 Réalisation

La redirection des impressions Lisp se fait en définissant un nouveau type de terminal virtuel. Nous définissons le type de terminal virtuel #:TTY:WINDOW. Pour ce type de terminal, les fonctions d'affichage (TYCN, TYNEWLINE, etc..) utilisent les fonctions d'affichage du graphique virtuel minimum. Ce terminal virtuel est défini dans le fichier VIRBITMAP.LL.

Lors de la sélection d'une fenêtre de type #:WINDOW:TTY, le terminal virtuel sera positionné à ce type de terminal. Il faut donc que les fenêtres de ce type aient des réactions spéciales aux messages CURRENT-WINDOW et UNCURRENT-WINDOW.

La mise à jour des paramètres de taille du terminal virtuel lors de la modification de la taille de la fenêtre est réalisée par une réaction spéciale au message MODIFY-WINDOW.

La gestion d'un curseur nécessite deux champs supplémentaires aux fenêtres de type #:WINDOW:TTY, qui permettent de stocker sa position courante. Nous avons aussi besoin de champs supplémentaires permettant de stocker diverses informations (voir le source Lisp).

Nous définissons donc d'abord le type de fenêtres #:WINDOW:TTY de la manière suivante :

```
; Le type #:WINDOW:TTY
(defstruct #:window:tty          ; Un sous type du type WINDOW
    (cx 0)               ; position curseur en x en pixel
    (cy 0)               ; position curseur en y en pixel
    (font 0)             ; la police utilisée
    (lmargin)                 ; ancienne marge gauche
    (rmargin)                 ; ancienne marge droite
    (tyshowcursor t)          ; on voit/ne voit pas le curseur
    (tty '#:tty:window)         ; type de terminal virtuel
    (itsoft ())               ; le package d'itsoft
```

```
    )

; Avec sa fonction de création spécifique
(de #:window:tty:create (to le wi he ti hi vi)
    (let ((win1 (create-window 'window to le wi he ti hi vi))
          (win2 (#:window:tty:make)))
        (exchvector win1 win2)
        (typevector win1 '#:window:tty)
        (bltvector win1 0 win2 0)
          (#:window:tty:cx win1 (x-base-space))
          (#:window:tty:cy win1 (y-base-space))
        (#:window:tty:lmargin win1 0)
        (#:window:tty:rmargin win1 (quo wi (width-space)))
        (#:window:tty:itsoft win1 #:sys-package:itsoft)
        win1))
```

Nous définissons ensuite les comportements spécifiques à la réception des messages CURRENT-WINDOW, UNCURRENT-WINDOW et MODIFY-WINDOW. Ceux-ci consistent à mettre à jour la marge droite et les variables définissant la taille de l'écran. Notez l'utilisation du SEND-SUPER pour activer les comportements des fenêtres de type WINDOW après ces mises à jour.

```
(de #:window:tty:current-window (win)
    (:flip-tty win)           ; le terminal devient #:tty:window
    (:adjust win)             ; on calcule la taille de l'écran
    (send-super '#:window:tty
                'current-window win)) ; le comportement par défaut

(de #:window:tty:uncurrent-window (win)
    (itsoft 'flush ())        ; on vide les tampons
    (tyflush)
    (when (eq (outchan) ())   ; on repositionne les marges
        (#:window:tty:lmargin win (lmargin))
        (#:window:tty:rmargin win (rmargin)))
    (:flip-tty win)           ; le terminal retrouve son ancien type
    (send-super '#:window:tty
                'uncurrent-window win)) ; le comportement par défaut

(de #:window:tty:modify-window (win x y w h ti hi vi)
    (send-super '#:window:tty
                'modify-window win x y w h ti hi vi) ; le défaut
    (when (and (eq win (current-window)) (or w h))
        (:adjust win)))     ; mise à jour des tailles
```

Les fenêtres de type #:WINDOW:TTY sont maintenant utilisables. En voici deux utilisations :

```
; Pepe dans une fenêtre
(defvar pepewindow ())

(de pepe-in-window (f)
    (unless pepewindow
        (setq pepewindow
            (create-window '#:window:tty 10 10 500 500 "pepe" 0 1)))
    (with ((current-window pepewindow))
        (pepefile f)))

; Démonstrations vidéo
```

```
(de demo ()
    (ifn (featurep 'window)
         (error 'demo "Pas de fenetres" ())
         (let ((w1) (w2) (w3))
              (protect (progn (setq w1 (create-window '#:window:tty
                                              1 1  76 21 "Hanoi 1" 0 1)
                                    w2 (create-window '#:window:tty
                                              1 25 76 21 "Hanoi 2" 0 1)
                                    w3 (create-window '#:window:tty
                                              1 49 76 20 "Hanoi 3" 0 1))
                              (bitmap-refresh)
                              (progn (with ((current-window w3))
                                           (hanoi 8))
                                     (with ((current-window w2))
                                           (hanoi 7))
                                     (with ((current-window w1))
                                           (whanoi 9))))
                       (kill-window w1)
                       (kill-window w2)
                       (kill-window w3)))))
```

CHAPITRE 19

Le Dispositif de Pointage

Matthieu Devin
Laurence Gallot
Jean-Marie Hullot

Le système Le_Lisp permet de gérer un dispositif permettant de désigner des points sur un écran haute résolution. Ce dispositif, appelé *souris virtuelle*, peut être une souris (avec un nombre quelconque de boutons), un crayon optique, une tablette graphique, un écran tactile, ou tout autre dispositif de pointage.

Un *événement souris* est une action physique sur la souris qui permet de désigner un point de l'écran (enfoncement d'un bouton parmi plusieurs, relachement, pression courte ou longue, double pression, déplacement, etc.). Ces événements sont encodés par des nombres entiers supérieurs ou égaux à 256. Le nombre d'événements différents disponibles dépend bien évidemment des systèmes. On dispose en général de deux événements différents.

AVERTISSEMENT

Le système de pointage décrit ici est un système minimum. Il peut, et doit(!), être étendu dans plusieurs directions. Par exemple pour la détection de zones sensibles dans les fenêtres, et la manipulation de types d'événements plus structurés (événements "modification de fenêtre", événements systèmes, etc.). Les implémenteurs sont invités à faire toutes les extensions qu'ils jugent souhaitables sur le système. Le souci de portabilité éventuelle vers d'autres systèmes Le_Lisp doit néanmoins être présent lors de chaque extension. Nous donnons un exemple d'extension dans la dernière section de ce chapitre. Cet exemple décrit la manipulation de zones sensibles dans les fenêtres.

#:EVENT:MOVE-EVENT [Variable]
#:EVENT:CLICK-EVENT [Variable]

Ces deux variables contiennent deux codes d'événements différents disponibles sur le système. Ces événements sont des petits entiers supérieurs ou égaux à 256.

Les événements sont stockés dans une queue qu'un programme peut consulter à tout moment. Il est possible de demander l'ajout d'événements *clavier* lors de la frappe de caractères, et le déclenchement d'une interruption de nom EVENT, lors de l'ajout de tout événement dans la queue.

L'asservissement du curseur associé à la souris sur l'écran est toujours fait par le système hôte.

19.1 Les Modes de Fonctionnement de la Souris

La souris peut fonctionner selon plusieurs modes. Ces modes sont la combinaison de trois indicateurs indépendants : événement clavier, mode raccourci, interruption souris.

Si l'indicateur *événement clavier* est positionné, les caractères frappés au clavier sont insérés dans la queue d'événements. Les événements correspondant ont pour code le code interne du caractère, qui est un petit entier compris entre 0 inclus, et 256 exclus.

Si l'indicateur *mode raccourci* est positionné, l'ajout d'un nouvel événement de code #:EVENT:MOVE-EVENT dans la queue suit la procédure suivante : si le dernier événement de la queue est aussi un événement de code #:EVENT:MOVE-EVENT le nouvel événement remplace l'ancien, sinon il est ajouté normalement. Cette opération est empruntée au système APIO de l'Apollo. Elle permet de diminuer la taille de la queue d'événements.

Si l'indicateur *interruption souris* est positionné, une interruption programmable de nom EVENT est déclenchée lors de l'ajout de tout événement dans la queue.

(EVENT-MODE <mode>) [SUBR à 0 ou 1 argument]

<Mode> devient le mode de fonctionnement de la souris. Sans argument EVENT-MODE retourne le mode courant de la souris. On peut ainsi utiliser la structure de contrôle WITH pour modifier temporairement le mode de fonctionnement de la souris.

Les codes sont définis comme suit :

mode	interruption souris	mode raccourci	événement clavier
0			
1			X
2		X	
3		X	X
4	X		
5	X		X
6	X	X	
7	X	X	X

A l'initialisation du système le mode de la souris est 0 : les interruptions EVENT ne sont pas déclenchées, les événements clavier ne sont pas stockés dans la queue, et la queue n'utilise pas le mode raccourci

De manière interne le système stocke toujours les événements claviers dans la queue d'événements. Les fonctions de lecture de la queue ignorent ces événements si l'indicateur *événement clavier* n'est pas positionné : la lecture d'un événement clavier provoque la lecture de l'événement suivant dans la queue, l'événement clavier est retiré de la queue.

Les fonctions TYI et TYS recherchent les caractères dans la queue d'événements. Elles lisent donc la queue d'événements jusqu'à y trouver un événement clavier. On peut schématiser le fonctionnement de la fonction TYI à partir de la fonction READ-EVENT de la manière suivante :

```
(DE TYI ()
    (WITH ((EVENT-MODE (LOGOR (EVENT-MODE) #%001))) ; événement clavier
          (READ-EVENT)
```

```
(UNTIL (< #:EVENT:CODE 256)
       (READ-EVENT))
#:EVENT:CODE))
```

19.2 La Queue d'Evénements

Les événements stockés dans la queue sont des triplets indiquant un lieu (deux entiers) et un événement encodé (un entier).

La longueur de la queue d'événements dépend des systèmes. Si la queue déborde, les événements les plus anciens sont perdus.

(EVENTP <levent>) [SUBR à 0 ou 1 argument]

Sans argument, ce prédicat est vrai s'il y a au moins un événement dans la queue. Si l'argument est fourni, ce doit être une liste de codes d'événements. Le prédicat est vrai s'il y a un événement dans la queue dont le code est dans la liste <levent>. Cette fonction ne modifie jamais le contenu de la queue.

(READ-EVENT) [SUBR à 0 argument]
(PEEK-EVENT) [SUBR à 0 argument]

Ces fonctions permettent de lire et de consulter le prochain événement de la queue. La fonction READ-EVENT extrait le premier événement de la queue, la fonction PEEK-EVENT se contente de consulter cet événement. Le lieu de l'événement (en coordonnées globales), et son code sont stockés dans les variables globales #:EVENT:X, #:EVENT:Y et #:EVENT:CODE. Si la queue d'événements est vide, ces fonctions sont bloquées en attente d'un événement.

La fonction suivante imprime une trace des événements de la souris :

```
(DE TRACE-EVENTS ()
    (LOOP
        (READ-EVENT)
        (PRINT "Lieu " #:EVENT:X ", " #:EVENT:Y
               " Evenement " #:EVENT:CODE))))
```

La fonction suivante démontre l'utilisation des événements claviers :

```
(DE LETTERS ()
    (CLEAR-GRAPH-ENV)
    (FLUSH-EVENT)
    (READ-EVENT)
    (UNTIL (EQ #:EVENT:CODE #:EVENT:CLICK-EVENT)
           (READ-EVENT))
    (WITH ((EVENT-MODE 3))
          (LET ((C #/a))
               (UNTIL (EQ #:EVENT:CODE #/q)
                      (LOCAL-READ-EVENT)
                      (WHEN (LT #:EVENT:CODE 256)
                            (SETQ C #:EVENT:CODE))
                      (DRAW-CN #:EVENT:X #:EVENT:Y C)))))
```

(LOCAL-READ-EVENT) [SUBR à 0 arguments]

Lit le prochain événement de la queue et convertit les coordonnées globales de cet événement en coordonnées locales à l'environnement graphique de la fenêtre courante.

LOCAL-READ-EVENT peut être défini en Lisp de la manière suivante :

```
(DE LOCAL-READ-EVENT ()
    (READ-EVENT)
    (MAP-WINDOW '#:EVENT:X '#:EVENT:Y '#:EVENT:X '#:EVENT:Y))
```

(FLUSH-EVENT) [SUBR à 0 argument]

Vide la queue d'événements.

(ADD-EVENT <x> <y> <event>) [SUBR à 3 arguments]

Ajoute un nouvel événement dans la queue d'événements souris. Cet événement est décrit par le triplet <x>, <y>, <event>. L'événement est rajouté à la fin de la queue d'événements. Cette fonction est particulièrement utile pour permettre d'étendre les types d'événements : rajout d'événements MENUS, FENETRES, etc. Les arguments <x> et <y> doivent êtres des coordonnées globales d'un point de l'écran, l'argument <event> peut être n'importe quel objet Lisp (entier, liste, vecteur, etc.), ce qui laisse toute latitude à l'utilisateur pour étendre les types d'événements.

#:EVENT:X, #:EVENT:Y, #:EVENT:CODE [Variables]

Ces trois variables contiennent les coordonnées globales et le code du dernier événement lu par les fonctions READ-EVENT, PEEK-EVENT et LOCAL-READ-EVENT.

19.3 L'Interruption Programmable EVENT

EVENT [Interruption programmable]

Cette interruption programmable est déclenchée lors de l'ajout d'un événement dans la queue, si l'indicateur *interruption souris* est positionné. Cette interruption a le même statut que les interruptions BREAK et CLOCK : elle est inhibée par la fonction WITH-NO-INTERRUPTS, et lors de son traitement les autres interruptions sont inhibées par un appel implicite à cette même fonction.

La queue d'événements est automatiquement vidée après le traitement de l'interruption EVENT, par la fonction FLUSH-EVENT. Ceci garantit que le premier appel à la fonction READ-EVENT dans une fonction invoquée par l'interruption EVENT lit bien l'événement qui a causé cette interruption.

(EVENT) [SUBR à 0 argument]

Cette fonction est la fonction standard de traitement de l'interruption EVENT. Elle ne fait absolument rien.

EVENT peut être défini en Lisp de la manière suivante :

```
(DE EVENT ()
    )
```

19.4 Consultation Asynchrone de la Souris

(READ-MOUSE) [SUBR à 3 arguments]

Cette fonction permet de consulter la position de la souris indépendamment de la queue d'événements. Les coordonnées globales de la souris sont stockées dans les deux variables #:MOUSE:X et #:MOUSE:Y, l'état de la souris dans la globale #:MOUSE:STATE.

L'état de la souris est encodé par un nombre. La notion d'état est différente de la notion d'événement et dépend des systèmes. Dans le système standard le seul état du dispositif de pointage est encodé par le nombre 0. Sur certains systèmes la souris peut être en différents états: des boutons sont restés enfoncés, la souris ne touche plus la table, elle a les pattes en l'air, etc..

#:MOUSE:STATE [Variable]

Cette variable contient l'état de la souris lu par le dernier appel à la fonction READ-MOUSE.

19.5 Exemple d'Extension du Système

Pour permettre la détection facile de zones dans les fenêtres (boutons, menus, etc.) il peut être utile de définir des *zones sensibles*. Le passage de la souris dans une zone sensible génèrera un évenement de type *zone sensible* décrivant la zone et l'endroit désigné par la souris dans cette zone. Les fonctions décrites ci-dessous constituent uniquement des guides pour les implémenteurs désirant étendre le systèmne de pointage.

SENSIBLE-ZONE [STRUCTURE]

Cette structure décrit une zone sensible. Une zone sensible est une zone rectangulaire, qui peut engendrer des événements lors du passage de la souris, ou de la désignation d'un point, dans le rectangle. Les conditions de création de l'événement (au passage, à la désignation, à la sortie de la zone, etc.) dépendent du type de la zone sensible.
Pour les événements engendrés par une zone sensible, la variable #:EVENT:CODE contiendra la zone sensible concernée, et les variables #:EVENT:X et #:EVENT:Y contiendront les coordonnées de la souris lors de la génération de l'événement.
La structure des zones sensibles correspond à la définition suivante :

```
(DEFSTRUCT SENSIBLE-ZONE
           X Y W H  ; coin supérieur gauche du rectangle
           TYPE)    ; type de la zone
```

(CREATE-ZONE <x> <y> <w> <h> <type>) [SUBR à 5 arguments]

Crée et active une zone sensible. <type> est un petit entier qui décrit le type de zone. La création d'une zone sensible n'a aucun effet surl'environnement graphique attaché à la fenêtre. Un action avec la souris dans un zone sensible n'a a priori aucun autre effet que d'engendrer un événement *zone sensible* dans la queue d'événements.

Les types sont définis comme suit :

0 - zone sensible au passage de la souris.
1 - zone sensible à la désignation d'un point.
2 - zone sensible à la sortie de la souris de la zone.

Il peut y avoir beaucoup plus de types différents de zones sensibles, dépendant des systèmes. Certains système peuvent proposer des types de zones qui réalisent un écho à la sélection d'une zone sensible (pour implanter facilement des menus par exemple).

(KILL-ZONE <zone>) [SUBR à 1 argument]

rend insensible la zone <zone> et récupère la place occupée par <zone>.

CHAPITRE 20

Les Primitives Graphiques

Matthieu Devin
Jean-Marie Hullot

Ce chapitre décrit les primitives graphiques portables. Ces primitives constituent une interface graphique virtuelle qui permet le dessin et le remplissage de figures, et l'affichage de texte sur des écrans bitmap noir et blanc.

Ces primitives ont déja implantées sur :
- SM90 interfacé avec ASH
- SUN interfacé avec ASH et avec SUNTOOLS
- SPS9 interfacé avec ASH et avec le système graphique de ROS 3.3.
- MacIntosh interfacé avec la ROM du système

Elles sont en projet sur :
- MicroVAX II sous micro VMS
- Amiga interfacé avec la ROM du système
- Colorix

AVERTISSEMENT

Les primitives graphiques constituent un ensemble minimum permettant de réaliser des graphiques élémentaires sur un écran. Pour obtenir des effets plus élaborés (couleurs, animations, déplacement de segments, etc.), il est nécessaire d'ajouter de nouvelles primitives. Il faut donc considérer que le système graphique portable de Le_Lisp est ouvert et extensible.
La dernière section de ce chapitre donne un exemple d'extension permettant de manipuler des mémoires de points (bitmaps). Cette section doit être lue comme un exemple d'écriture d'extensions au système graphique portable Le_Lisp.

20.1 Les Environnements Graphiques

Un *environnement graphique* est un objet capable de réaliser certaines opérations graphiques sur un plan d'affichage muni d'un système de coordonnées.

Sous le fenêtrage virtuel un environnement graphique est attaché à chaque fenêtre. Une fenêtre constitue une *vue* sur le plan d'affichage, définie par les coordonnées des coins supérieur gauche et inférieur droit de la fenêtre dans ce plan. Cette vue est définie au chapitre 18, le fenêtrage virtuel. Il n'est pas possible actuellement de modifier la vue que donne la fenêtre sur l'environnement graphique qui lui est associé.

L'environnement graphique courant, est l'environnement graphique attaché à la fenêtre courante. Les primitives graphiques agissent sur cet environnement graphique. Elles sont implantées selon une méthodologie orientée objet bâtie sur le même modèle que l'implantation des fenêtres. Chaque primitive transmet un message à

l'environnement graphique courant. C'est la méthode associée à ce message pour le type de l'environnement graphique courant qui réalise effectivement l'opération graphique.

Par exemple la fonction DRAW-POLYLINE est définie de la manière suivante :

```
(DE DRAW-POLYLINE (n vx vy)
    (SEND 'DRAW-POLYLINE (CURRENT-GRAPH-ENV) n vx vy))
```

20.1.1 Structure des environnements graphiques

Chaque environnement graphique est caractérisé par 5 paramètres : une police (FONT) un type de lignes (LINE-STYLE), un motif de remplissage (PATTERN), un mode de combinaison (MODE) et un rectangle de découpe (CLIP). Ceci correspond à la définition de structure suivante :

```
(DEFSTRUCT GRAPH-ENV
           FONT
           LINE-STYLE
           PATTERN
           MODE
           CLIP)
```

Les fonctions suivantes permettent de consulter et modifier les champs de l'environnement graphique courant.

(CURRENT-FONT ‹font›) [SUBR à 0 ou 1 argument]

‹font› devient la police de l'environnement graphique courant. Sans argument, CURRENT-FONT retourne la police de l'environnement graphique courant. ‹Font› est un petit entier positif qui désigne une police de caractère. Le nombre de polices utilisables dépend des systèmes. Les polices 0 et 1 sont cependant présentes sur tous les systèmes. Leur aspect à l'écran dépend des systèmes.

(FONT-MAX) [SUBR à 0 argument]

retourne le plus grand argument admissible par la fonction CURRENT-FONT, c'est-à-dire le nombre de polices différentes moins un. Ce nombre est toujours supérieur à 1.

(LOAD-FONT ‹string›) [SUBR à 1 argument]

Charge la police de caractère décrite par la chaîne de caractères ‹string› dans le système. Retourne en valeur un petit entier qui peut être donné en paramètre à la fonction CURRENT-FONT. Le résultat d'un appel à la fonction FONT-MAX reflètera le chargement de cette nouvelle police dans le système.
Le contenu de la chaîne ‹string› dépend des systèmes et n'est absolument pas portable.

(CURRENT-LINE-STYLE ‹line-style›) [SUBR à 0 ou 1 argument]

‹line-style› devient le style de ligne de l'environnement graphique courant. Sans argument, CURRENT-LINE-STYLE retourne le style de ligne courant. ‹line-style› est un petit entier positif qui désigne un style de ligne. Le nombre de styles différents disponibles, et l'aspect à l'écran de chaque style dépend des systèmes.

(LINE-STYLE-MAX) [SUBR à 0 argument]

retourne le plus grand argument admissible par la fonction CURRENT-LINE-STYLE.

(CURRENT-PATTERN ‹pattern›) [SUBR à 0 ou 1 argument]

‹pattern› devient le motif de l'environnement graphique courant. Sans argument, CURRENT-PATTERN retourne le motif courant. ‹pattern› est un petit entier positif qui désigne un motif. Le nombre de motifs différents disponibles, et l'aspect à l'écran de chaque motif dépend des systèmes.

(PATTERN-MAX) [SUBR à 0 argument]

retourne le plus grand argument admissible par la fonction CURRENT-PATTERN.

(CURRENT-MODE ‹mode›) [SUBR à 0 ou 1 argument]

‹mode› devient le mode de combinaison de l'environnement graphique courant. Sans argument retourne le mode de combinaison courant. ‹mode› est un petit entier qui désigne un mode de combinaison. 16 modes de combinaison sont disponibles (de 0 à 15). Chaque mode indique de quelle manière combiner un pixel de l'environnement graphique et un pixel affiché par l'une des fonctions graphiques primitives.

Les modes sont définis en fonction du pixel source (S) et du pixel destination (D) comme suit :

S	D	Pixel résultant															
Mode	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
-																	
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

#:MODE:SET [Variable]
#:MODE:OR [Variable
#:MODE:XOR [Variable]
#:MODE:NOT [Variable]

Ces variables contiennent les modes de combinaison les plus couramment utilisés. Elles ont les valeurs suivantes :

```
#:MODE:SET 3
#:MODE:OR  7
#:MODE:XOR 6
#:MODE:NOT 12
```

(CURRENT-CLIP <x> <y> <w> <h>) [SUBR à 0 ou 4 arguments]

Le rectangle de coin supérieur gauche (<x>, <y>), de largeur <w> et de hauteur <h> devient le rectangle de découpe courant. Sans argument retourne les coordonnées du rectangle de découpe courant dans les variables globales #:CLIP:X, #:CLIP:Y, #:CLIP:W, #:CLIP:H. Toutes les opérations d'affichage sont limitées par le rectangle de découpe : uniquement les affichages à l'intérieur du rectangle de découpe sont réalisés.

#:CLIP:X, #:CLIP:Y, #:CLIP:W, #:CLIP:H [Variables]

Ces variables contiennent les coordonnées du rectangle de découpe courant après un appel à la fonction CURRENT-CLIP sans argument.

20.2 Les Primitives Graphiques

Les fonctions strictement primitives sont les quatre primitives définies par le standard graphique GKS : DRAW-POLYLINE, DRAW-POLYMARKER, FILL-AREA et DRAW-SUBSTRING. Pour des raisons d'efficacité, des extensions à ces primitives sont fournies dans Le_Lisp. Ces extensions permettent principalement l'affichage et le remplissage de zones rectangulaires, opérations souvent extrêmement rapides sur les écrans bitmap. Ces fonctions pourraient, bien sûr, être définies à partir des primitives GKS.

(DRAW-POLYLINE <n> <vx> <vx>) [SUBR à 3 arguments]

<vx> et <vx> sont deux vecteurs de longueur supérieure à <n>. Cette fonction dessine les <n>-1 vecteurs définis par la suite de points (<vx>[0], <vx>[0]) ... (<vx>[n-1], <vx>[n-1]). Le type de ligne utilisé pour dessiner les vecteurs est donné par la valeur courante du paramètre LINE-STYLE. Le mode de combinaison courant décrit la façon dont la ligne se combine avec les objets déja dessinés.

Exemple : L'appel suivant dessine un triangle :

```
(DRAW-POLYLINE 4 #[0 10 20 0] #[0 10 0 0])
```

(DRAW-POLYMARKER <n> <vx> <vx>) [SUBR à 3 arguments]

Cette fonction est identique à la précédente mais dessine uniquement les extrémités des vecteurs. Le mode de combinaison courant décrit la manière dont les points se combinent avec les objets déjà dessinés.

Exemple : L'appel suivant dessine les sommets d'un triangle :

```
(DRAW-POLYMARKER 3 #[0 10 20 0] #[0 10 0 0])
```

(FILL-AREA <n> <vx> <vx>) [SUBR à 3 arguments]

<vx> et <vx> sont deux vecteurs de longueur supérieure à <n>. Cette fonction remplit le polygone défini par les <n> points dont les coordonnées (x, y) sont données par les vecteurs <vx> et <vx>. Le polygone est rempli par le motif défini par la valeur courante du paramètre FILL-PATTERN. Le mode de combinaison courant décrit la manière dont le polygone se combine avec les

objets déjà dessinés.

Exemple : L'appel suivant remplit une zone triangulaire :

```
(FILL-AREA 3 #[0 10 20] #[0 10 0])
```

(DRAW-SUBSTRING <x> <y> <s> <start> <length>) [SUBR à 5 arg.]

Voir la section 18 : l'environnement graphique minimum.

(DRAW-ELLIPSE <x> <y> <rx> <ry>) [SUBR à 4 arguments]

Dessine une ellipse dont les axes sont parallèles aux axes de coordonnées. Le point de coordonnées (<x>, <y>) est le centre de l'ellipse, <rx> et <ry> sont deux entiers qui représentent les deux rayons de l'ellipse.

(FILL-ELLIPSE <x> <y> <rx> <ry>) [SUBR à 4 arguments]

Cette fonction est identique à la précédente mais remplit l'ellipse avec le motif courant au lieu de la dessiner.

Ces deux fonctions ne sont pas des primitives GKS à strictement parler mais la plupart des systèmes GKS en fournissent les fonctionalités au moyen de fonctions d'échappement (ESCAPE).

(CLEAR-GRAPH-ENV) [SUBR à 0 argument]

Voir la section 18 : l'environnement graphique minimum.

20.3 Les Fonctions Etendues

20.3.1 Les fonctions de tracé de lignes

Ces fonctions tracent des lignes en utilisant le type de ligne (LINE-STYLE) courant. Nous donnons leur définition à partir des primitives compatibles avec GKS.

(DRAW-POINT <x> <y>) [SUBR à 2 arguments]

DRAW-POINT peut être défini en Lisp de la manière suivante :

```
(DE DRAW-POINT (x y)
    (DRAW-POLYMARKER 1 (VECTOR x) (VECTOR y)))
```

(DRAW-LINE <x0> <y0> <x1> <y1>) [SUBR à 4 arguments]

DRAW-LINE peut être défini en Lisp de la manière suivante :

```
(DE DRAW-LINE (x0 y0 x1 y1)
    (DRAW-POLYLINE 2 (VECTOR x0 x1) (vector y0 y1)))
```

(DRAW-RECTANGLE <x> <y> <w> <h>) [SUBR à 4 arguments]

DRAW-RECTANGLE peut être défini en Lisp de la manière suivante :

```
(DE DRAW-LINE (x0 y0 w h)
    (DRAW-POLYLINE 5 (VECTOR x0 (+ x0 w) (+ x0 w) x0 y0)
        (VECTOR y0 y0 (+ y0 h) (+ y0 h) y0)))
```

(DRAW-CIRCLE <x> <y> <r>) [SUBR à 4 arguments]

DRAW-CIRCLE peut être défini en Lisp de la manière suivante :

```
(DE DRAW-CIRCLE (x y r)
    (DRAW-ELLIPSE x y r r))
```

20.3.2 Les fonctions de remplissage

Ces fonctions remplissent des zones en utilisant le motif de remplissage courant (FILL-PATTERN). Nous donnons leur définition à partir des primitives GKS.

(FILL-RECTANGLE <x> <y> <w> <h>) [SUBR à 4 argments]

FILL-RECTANGLE peut être défini en Lisp de la manière suivante :

```
(DE FILL-RECTANGLE (x y w h)
    (FILL-AREA 4 (VECTOR x0 (+ x0 w) (+ x0 w) x0)
                 (VECTOR y0 y0 (+ y0 h) (+ y0 h))))
```

(FILL-CIRCLE <x> <y> <r>) [SUBR à 3 arguments]

FILL-CIRCLE peut être défini en Lisp de la manière suivante :

```
(DE FILL-CIRCLE (x y r)
    (FILL-ELLIPSE x y r r))
```

20.3.3 Affichage de texte

Ces fonctions sont définies au chapitre 18. Elles sont répertoriées ici uniquement à titre de rappel.

(DRAW-CN <x> <y> <cn>) [SUBR à 3 arguments]
(DRAW-STRING <x> <y> <s>) [MACRO à 3 arguments]

(WIDTH-SUBSTRING <string> <start> <length>)
(HEIGHT-SUBSTRING <string> <start> <length>)
(X-BASE-SUBSTRING <string> <start> <length>)
(Y-BASE-SUBSTRING <string> <start> <length>)
(X-INC-SUBSTRING <string> <start> <length>)
(Y-INC-SUBSTRING <string> <start> <length>)

(HEIGHT-SPACE)
(WIDTH-SPACE)

20.4 Extension du Système

Nous donnons ci-dessous deux exemples d'extension du système. Le premier permet de réaliser des opérations sur des mémoires de points (bitmaps) associées aux différents objets du système graphique. Le second permet de manipuler des environnements graphiques *en mémoire* et non nécessairement associés à un dispositif d'affichage.

Les fonctions définies ci-dessous sont uniquement un exemple d'extension du système graphique. Leur définition est un guide à l'usage des personnes désirant ajouter de nouvelles possibilités graphiques. Les implémenteurs sont encouragés à y apporter les modifications qu'ils désirent.

20.4.1 Les mémoires de points

BITMAP [Structure]

Une mémoire de points est un tableau de bits à deux dimensions. Ce peut être une vraie structure Lisp ou bien un pointeur dans une zone externe à Lisp. La structure interne exacte d'une mémoire de points dépend donc des systèmes. Elle peut être modélisée par la la définition de structure suivante :

```
(DEFSTRUCT BITMAP
           W
           H
           BITS)
```

Les champs W et H contiennent la largeur et la hauteur de la mémoire de points. Ce sont des petits entiers. Le contenu du champ BITS dépend des systèmes. Les champs d'une structure de ce type ne peuvent pas être modifiés.

Une mémoire de points s'imprime sous la forme suivante :

```
#B(<largeur> <hauteur> #[<string1> ... <stringN>])
```

Ou <largeur> et <hauteur> sont des petits entiers représentant la mémoire de points. et <stringI> des chaînes de caractères représentant les points de la I-ème ligne de la mémoire de points. Le #-macro caractère #B permet de lire une mémoire de points.

#B [#-Macro]

Permet de lire une mémoire de point.

#B peut être défini en Lisp de la manière suivante :

```
(DEFSHARP |B| ()
     (APPLY 'CREATE-BITMAP (READ)))
```

(#:BITMAP:W <bitmap>) [SUBR à 1 argument]
(#:BITMAP:H <bitmap>) [SUBR à 1 argument]

Les fonctions #:BITMAP:W et #:BITMAP:H permettent de consulter les champs W et H d'une mémoire de points donnée en argument.

(#:BITMAP:BITS <bitmap> <vstring>) [SUBR à 0 ou 1 argument]

convertit le tableau de bits de la mémoire de points <bitmap> en un vecteur de chaînes de caractères. Chaque chaîne représente une ligne de la mémoire de points. Le premier caractère de la première chaîne décrit les 8 premiers bits de la première ligne de la mémoire de points. Le bit de poids le plus fort dans cette chaîne décrit le premier bit.
Si le second argument est fourni ce doit être un vecteur de chaînes de caractères. Dans ce cas le contenu de ce vecteur sert à initialiser le contenu de la mémoire de points. Si l'argument décrit plus de points que ne peut contenir la mémoire les points supplémentaires sont ignorés. S'il contient moins de points, les points non fournis sont initialisés à une valeur dépendant des systèmes.

(CREATE-BITMAP <w> <h> <vstring>) [SUBR à 2 ou 3 arguments]

CREATE-BITMAP crée une nouvelle mémoire de points et la retourne en valeur. Le troisiéme argument, optionnel, permet d'initialiser le contenu de la mémoire de points avec le vecteur de chaînes de caractères <vstring>.

(KILL-BITMAP <bitmap>) [SUBR à 1 argument]

KILL-BITMAP permet de libérer la place mémoire utilisée par la mémoire de points donnée en argument. Cette fonction est nécessaire si une mémoire de points peut contenir un pointeur dans une zone externe au système Le_Lisp.

(BMREF <bitmap> <x> <y>) [SUBR à 3 arguments]
(BMSET <bitmap> <x> <y> <bit>) [SUBR à 4 arguments]

Ces fonctions permettent de consulter et de modifier le bit de coordonnées (<x>, <y>) de la mémoire de points <bitmap>.

(BITBLIT <b1> <b2> <x1> <y1> <x2> <y2> <w> <h>) [SUBR à 8 arguments]

recopie le rectangle de coin supérieur gauche (<x2>, <y2>), de largeur <w> et de hauteur <h> de la mémoire de points <b2> au point de coordonnées (<x1>, <y1>) dans la mémoire de points <b1>. Les points de la zone recopiée se combinent avec les points de <b1> selon le mode de combinaison courant. La valeur retournée par cette fonction n'est pas significative.
<b1> et <b2> peuvent représenter la même m'emoire de points. Ceci permet de réaliser facilement des animations.

```
ex:  ; Recopie du quart d'écran supérieur gauche de <bitmap> dans le quart
     ; d'écran inférieur droit de la même mémoire de points <bitmap>
     ; <bitmap> a pour largeur <w> et hauteur <h>
     ? (BITBLIT <bitmap> <bitmap>
     ?          (div <w> 2) (div <h> 2)
     ?          0 0
     ?          (div <w> 2) (div <h> 2))
     = 0
```

Les fonctions qui suivent permettent de manipuler les mémoires de points correspondant aux environnements graphiques, aux caractères des polices et au motifs de remplissages.

(WINDOW-BITMAP <window>) [SUBR à 1 argument]

retourne en valeur la mémoire de points de l'environnement graphique de la fenêtre <window>. La mémoire de points reçue est partagée avec celle de l'environnement graphique de la fenêtre <window>. Toute opération sur l'une est donc immédiatement répercutée sur l'autre.

(CHAR-BITMAP <cn> <bitmap>) [SUBR à 1 ou 2 arguments]

Avec un seul argument cette fonction retourne la mémoire de points représentant le caractère de code interne <cn> dans la police de caractères courante. Si le second argument est fourni, <bitmap> devient la mémoire de points du caractère <cn> dans la police courante.
La mémoire de points recue en valeur est partagée avec la police de caractère courante : une modification de cette mémoire de points (au moyen des fonctions BITBLIT ou BMSET) est immédiatement répercutée dans la police courante.

(PATTERN-BITMAP <pn> <bitmap>) [SUBR à 1 ou 2 arguments]

cette fonction a le même rôle que la précédente mais elle permet de manipuler les mémoires d'écran des motifs de remplissage. <pn> est ici un petit entier acceptable comme argument par la fonction CURRENT-PATTERN.

20.4.2 Les environnements graphiques en mémoire

Il peut être utile de préparer une image dans une mémoire de points hors de l'écran. On voudrait par exemple tracer des lignes, ou remplir des zones dans une mémoire de points, pout ensuite pouvoir afficher ces dessins (des icône), au moyen de la fonction BITBLIT, dans différents environnements.

Pour faire cela on peut définir un nouveau type d'environnement graphique qui a pour effet d'afficher les dessins non pas sur l'écran mais dans une mémoire de points hors de l'écran. Les fonctions de la section précédente permettent de réaliser toutes les opérations d'affichage simplement.

Il faudra donc ajouter des fonctions permettant la création d'environnements graphiques de ce nouveau type, et la sélection d'un environnement graphique quelconque (actuellement, seule la sélection d'un environnement graphique attaché à une fenêtre est possible). L'implantation orientée objet utilisée pour les primitives graphiques rend des telles extensions réalisables à peu de frais.

ANNEXE A

Le Fichier Initial

Jérôme Chailloux

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

Il est nécessaire de charger ce fichier avant toute utilisation du système Le_Lisp version 15.2 à cause de la redéfinition de la boucle principale (toplevel).

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'startup))
```

A.1 Les variables globales internes du système

```
(progn
   (defvar #:system:loaded-from-file 'startup)
   ; et maintenant refaisons le pour mettre 'startup
   ; dans #:system:loaded-from-file .....
   (defvar #:system:loaded-from-file 'startup)
   (defvar #:system:in-read-flag #:system:in-read-flag)
   (defvar #:system:print-for-read #:system:print-for-read)
   (defvar #:system:print-package-flag #:system:print-package-flag)
   (defvar #:system:real-terminal-flag #:system:real-terminal-flag)
   (defvar #:system:line-mode-flag #:system:line-mode-flag)
   (defvar #:system:foreign-language ())
)
```

A.2 Les variables dépendantes du système

Toutes ces variables résident dans le package "SYSTEM:" et contiennent les noms absolus des préfixes et suffixes des bibliothèques Le_Lisp.

```
(defvar unix                    ; liste des systèmes UN*X
        '(vaxunix vme pe32unix bell sps9
          sm90 micromega metheus apollo cadmus sun hp9000))
```

Pour les systèmes UNIX la variable #:SYSTEM:DIRECTORY permet de calculer tous les autres directory. La commande *newdir* de la distribution standard permet de la mettre à jour.

```
(when (memq (system) '#.unix)
      (defvar #:system:directory "/udd/lelisp/version15.2/"))

(progn                          ; pour rester silencieux pendant le chargement
 ; Catalogue (préfixe) contenant la bibliothèque système Le_Lisp
   (defvar #:system:llib-directory
           #.(selectq (system)
                      (#.unix   (catenate #:system:directory "llib/"))
                      (vaxvms   "lelisp$disk:[lelisp.LLIB]")
                      (multics  ">udd>Vlsi>Le_Lisp>llib>")
                      (pcdos    "
                      (t        "")))
 ; Catalogue (préfixe) contenant la bibliothèque utilisateur Le_Lisp
   (defvar #:system:llub-directory
           #.(selectq (system)
                      (#.unix   (catenate #:system:directory "llub/"))
                      (vaxvms   "lelisp$disk:[lelisp.LLUB]")
                      (multics  ">udd>Vlsi>Le_Lisp>llub>")
                      (pcdos    "
                      (t        "")))
 ; Catalogue (préfixe) contenant la bibliothèque des tests du système
   (defvar #:system:lltest-directory
           #.(selectq (system)
                      (#.unix   (catenate #:system:directory "lltest/"))
                      (vaxvms   "lelisp$disk:[lelisp.LLTEST]")
                      (multics  ">udd>Vlsi>Le_Lisp>lltest>")
                      (pcdos    "
                      (t        "")))
 ; Catalogue (préfixe) contenant les modules système Le_Lisp.
   (defvar #:system:llmod-directory
           #.(selectq (system)
                      (#.unix   (catenate #:system:directory "llmod/"))
                      (vaxvms   "lelisp$disk:[lelisp.LLMOD]")
                      (multics  ">udd>Vlsi>Le_Lisp>llmod>")
                      (pcdos    "
                      (t        "")))
 ; Catalogue (préfixe) contenant les fichiers système format objet Le_Lisp.
```

```
(defvar #:system:llobj-directory
        #.(selectq (system)
                   (#.unix    (catenate #:system:directory "llobj/"))
                   (vaxvms    "lelisp$disk:[lelisp.LLOBJ]")
                   (multics   ">udd>Vlsi>Le_Lisp>llobj>")
                   (pcdos     "
                   (t         "")))

; Catalogue (préfixe) contenant les définitions des terminaux virtuels

(defvar #:system:virtty-directory
        #.(selectq (system)
                   (#.unix    (catenate #:system:directory "virtty/"))
                   (vaxvms    "lelisp$disk:[lelisp.VIRTTY]")
                   (multics   ">udd>Vlsi>Le_Lisp>virtty>")
                   (pcdos     "
                   (t         "")))

; Catalogue (préfixe) contenant les définitions des bitmaps virtuels

(defvar #:system:virbitmap-directory
        #.(selectq (system)
                   (#.unix    (catenate #:system:directory "virbitmap/"))
                   (vaxvms    "lelisp$disk:[lelisp.VIRBITMAP]")
                   (multics   ">udd>Vlsi>Le_Lisp>virbitmap>")
                   (pcdos     "
                   (t         "")))

; Catalogue (préfixe) contenant les images-mémoire standard

(defvar #:system:core-directory
        #.(selectq (system)
                   (#.unix    (catenate #:system:directory "llcore/"))
                   (vaxvms    "lelisp$disk:[lelisp.LLCORE]")
                   (multics   ">udd>Vlsi>Le_Lisp>core")
                   (pcdos     "
                   (t         "")))

; Liste des Catalogue (préfixe) contenant des fichiers Le_Lisp.

(defvar #:system:path (list
      ""
      #:system:llib-directory
      #:system:llub-directory
      #:system:llmod-directory
      #:system:llobj-directory
      #:system:lltest-directory
      #:system:virtty-directory
      #:system:virbitmap-directory ))

; Extension (suffixe) des fichiers source Le_Lisp

(defvar #:system:lelisp-extension
        #.(selectq (system)
                   (#.unix    ".ll")
                   (vaxvms    ".ll")
                   (multics   ".ll")
                   (pcdos     ".ll")
```

```
                    (t        "")))
; Extension (suffixe) des fichiers de description de module Le_Lisp,
  (defvar #:system:mod-extension
          #.(selectq (system)
                     (#.unix    ".lm")
                     (vaxvms    ".lm")
                     (multics   ".lm")
                     (pcdos     ".lm")
                     (t         "")))
; Extension (suffixe) des fichiers format objet Le_Lisp
  (defvar #:system:obj-extension
          #.(selectq (system)
                     (#.unix    ".lo")
                     (vaxvms    ".lo")
                     (multics   ".lo")
                     (pcdos     ".lo")
                     (t         "")))
; Extension (suffixe) des fichiers images-mémoire
  (defvar #:system:core-extension
          #.(selectq (system)
                     (#.unix    ".core")
                     (vaxvms    ".cor")
                     (multics   ".core")
                     (pcdos     ".cor")
                     (t         "")))
; Fichier contenant la base de données "termcap"
  (defvar #:system:termcap-file
          #.(selectq (system)
                     (#.unix    "/etc/termcap")
                     (vaxvms    "lelisp$disk:[lelisp.VIRTTY]termcap.db")
                     (multics   ">udd>Vlsi>Le_Lisp>virtty>termcap.data")
                     (pcdos     "rmcap")
                     (t         "")))
; Catalogue (préfixe) contenant la base de données "terminfo"
  (defvar #:system:terminfo-directory
          #.(selectq (system)
                     (#.unix    "/usr/lib/terminfo/")
                     (vaxvms    "lelisp$disk[lelisp.terminfo]")
                     (multics   ">udd>Vlsi>Le_Lisp>virtty>terminfo>")
                     (pcdos     "rminfo
                     (t         "")))
; Nom de l'éditeur à appeler par ^F
  (defvar #:system:editor
          #.(selectq (system)
                     (pe32unix "emin")
                     (bell     "vi")
                     (sps9     "redit")
                     ((sm90 micromega metheus apollo pe32unix)
```

```
                "emin")
        (#.unix "emacs")
        (vaxvms "edit")
        (multics "emacs -task")
        (t      "")))
```

A.3 Les libellés des erreurs standard

A.3.1 Les erreurs fatales

```
#- #:system:foreign-language
  (progn
   (defvar ERRFSTK "***** Erreur fatale : pile pleine.")
   (defvar ERRFSGC "***** Erreur fatale : pile pleine durant un GC.")
   (defvar ERRFPGC "***** Erreur fatale : pile pleine durant un PRINT.")
   (defvar ERRFSUD "***** Erreur fatale : pile vide.")
   (defvar ERRFSTR "***** Erreur fatale : zone des chaines pleine.")
   (defvar ERRFVEC "***** Erreur fatale : zone des vecteurs pleine.")
   (defvar ERRFSYM "***** Erreur fatale : zone des symboles pleine.")
   (defvar ERRFCNS "***** Erreur fatale : zone des listes pleine.")
   (defvar ERRFFLT "***** Erreur fatale : zone des flottants pleine.")
   (defvar ERRFFIX "***** Erreur fatale : zone des entiers pleine.")
   (defvar ERRFHEP "***** Erreur fatale : zone du tas pleine.")
   (defvar ERRFCOD "***** Erreur fatale : zone du code pleine.")
  )
#+ #:system:foreign-language
  (progn
   (defvar ERRFSTK "***** Fatal error : stack overflow.")
   (defvar ERRFSGC "***** Fatal error : stack overflow during GC.")
   (defvar ERRFPGC "***** Fatal error : stack overflow during PRINT.")
   (defvar ERRFSUD "***** Fatal error : stack underflow.")
   (defvar ERRFSTR "***** Fatal error : no room for strings.")
   (defvar ERRFVEC "***** Fatal error : no room for vectors.")
   (defvar ERRFSYM "***** Fatal error : no room for symbols.")
   (defvar ERRFCNS "***** Fatal error : no room for lists.")
   (defvar ERRFFLT "***** Fatal error : no room for floats.")
   (defvar ERRFFIX "***** Fatal error : no room for fixs.")
   (defvar ERRFHEP "***** Fatal error : heap overflow.")
   (defvar ERRFCOD "***** Fatal error : no room for code.")
  )
```

A.3.2 Les erreurs non fatales

```
#- #:system:foreign-language
  (progn
   (defvar ERRMAC "erreur de la machine")
   (defvar ERRUDV "variable indefinie")
   (defvar ERRUDF "fonction indefinie")
   (defvar ERRUDM "methode indefinie")
```

```
    (defvar ERRUDT "echappement indefini")
    (defvar ERRBDF "mauvaise definition")
    (defvar ERRWNA "mauvais nombre d'arguments")
    (defvar ERRBPA "mauvais parametre")
    (defvar ERRILB "liaison illegale")
    (defvar ERRBAL "mauvaise liste d'arguments")
    (defvar ERRNAB "pas de portee lexicale")
    (defvar ERRXIA "bloc lexical perime")
    (defvar ERRSXT "erreur de syntaxe")
    (defvar ERRIOS "erreur d'entree/sortie")
    (defvar ERR0DV "division par 0")
    (defvar ERRNNA "l'argument n'est pas un nombre")
    (defvar ERRNIA "l'argument n'est pas un entier")
    (defvar ERRNFA "l'argument n'est pas un flottant")
    (defvar ERRNSA "l'argument n'est pas une chaine")
    (defvar ERRNAA "l'argument n'est pas un atome")
    (defvar ERRNLA "l'argument n'est pas une liste")
    (defvar ERRNVA "l'argument n'est pas une variable")
    (defvar ERRVEC "l'argument n'est pas un vecteur")
    (defvar ERRSYM "l'argument n'est pas un symbole")
    (defvar ERRNDA "l'argument n'est pas une adresse")
    (defvar ERRSTC "l'argument n'est pas une structure")
    (defvar ERROOB "argument hors limite")
    (defvar ERRSTL "chaine trop longue")
    (defvar ERRGEN "ne sait pas calculer")
    (defvar ERRVIRTTY "terminal inconnu")
    (defvar ERRFILE "fichier inconnu")
    (defvar ERRICF "fichier incompatible")
  )
#+ #:system:foreign-language
  (progn
    (defvar ERRMAC "machine error")
    (defvar ERRUDV "undefined variable")
    (defvar ERRUDF "undefined function")
    (defvar ERRUDM "undefined method")
    (defvar ERRUDT "undefined escape")
    (defvar ERRBDF "bad definition")
    (defvar ERRWNA "wrong nmber of arguments")
    (defvar ERRBPA "bad parameter")
    (defvar ERRILB "illegal binding")
    (defvar ERRBAL "bad argment list")
    (defvar ERRNAB "no lexical scope")
    (defvar ERRXIA "inactive lexical scope")
    (defvar ERRSXT "syntax error")
    (defvar ERRIOS "I/O error")
    (defvar ERR0DV "0 divide")
    (defvar ERRNNA "not a number")
    (defvar ERRNIA "not a fix")
    (defvar ERRNFA "non float argument")
    (defvar ERRNSA "non string argument")
    (defvar ERRNAA "not an atom")
    (defvar ERRNLA "not a list")
```

```
   (defvar ERRNVA "not a variable")
   (defvar ERRVEC "not a vector")
   (defvar ERRSYM "not a symbol")
   (defvar ERRNDA "not an address")
   (defvar ERRSTC "not a structure")
   (defvar ERROOB "argument out of bounds")
   (defvar ERRSTL "string too long")
   (defvar ERRGEN "can't compute")
   (defvar ERRVIRTTY "unknown terminal type")
   (defvar ERRFILE "unknow file")
   (defvar ERRICF "incompatible file")
  )
```

A.4 Les codes ASCII symboliques pour la macro

```
(mapc
    (lambda (x y) (putprop x y '#:sharp:value))
   '(null bell bs   tab lf   return cr   esc sp   del rubout)
   '(#^@   #^G   #^H #^I #^J #^M     #^M #^[ #/   127 127))
```

A.5 Quelques autres fonctions

Toutes ces fonctions doivent être re-écrites en LLM3.

```
(de explodech (s)
    (mapcar 'ascii (explode s)))

(de implodech (l)
    (implode (mapcar 'cascii l)))

(de lhoblist (s)
    ; retourne tous les symboles qui débutent par la chaîne s
    (maploblist (lambda (x) (index s x 0))))
```

A.6 Les macros-caractères standard

```
(dmc ^L ()
     ; ^L  :  pour charger un fichier d'extension #:system:lelisp-extension
     (list 'libloadfile (readstring) t))

(dmc ^A ()
   ; Pour charger un module.
   (list 'loadmodule (readstring)))

(dmc ^E ()
     (cond ((eq (peekcn) 13)
               ; juste ^E suivi de 'return'
               '(pepefile ()))
           ((memq (peekcn) '(#/( #^P))
```

```
            ; expression : débute par ( ou ^P))
            (list 'pepefile (kwote (read))))
          (t ; un nom de fichier (symbole)
            (list 'pepefile (kwote (concat (readstring)))))))
(dmc ^F ()
    (let ((lu (readline)))
        (ifn lu
            (list 'comline #:system:editor)
            (lets ((fct (implode lu))
                   (file (getprop fct '#:system:loaded-from-file)))
                  (ifn (memq (typefn fct) '(expr fexpr macro dmacro))
                       (list 'pretty fct)
                       (if file
                           (list 'progn
                                 (list 'comline
                                       (catenate #:system:editor " " file))
                                 (list 'load file t))
                           (setq file
                                 (catenate
                                     (gensym) #:system:lelisp-extension))
                           (list 'progn
                                 (list 'prettyf file fct)
                                 (list 'comline
                                       (catenate #:system:editor " " file))
                                 (list 'load file t))))))))
(dmc ^P ()
     (cons 'pretty
           (implode (pname (catenate "(" (readstring) ")")))))
(dmc |!| ()                          ;  pour appeler le shell
     (let ((l (readstring)))
          (comline (if (equal l "") "$SHELL" l))))
```

A.7 Load primitif

```
(defvar  #:system:redef-flag ())
 ; (de eof (n)
 ;     ; traitement standard de la fin de fichier.
 ;     (close n)
 ;     (inchan ())
 ;     (if #:system:in-read-flag
 ;         (error 'read 'errsxt 11)
 ;         (exit eof n)))
(de loadfile (file redef?)
   (let ((#:system:loaded-from-file file)
         (#:system:redef-flag redef?)
         (#:sys-package:colon #:sys-package:colon)
         (#:system:in-read-flag ())
         (inchan (inchan)) )
```

```
      (inchan (openi file))
      (protect
         (untilexit eof (eval (read)))
         (let ((in (inchan))) (when in (close in)))
         (inchan inchan) )
       file ))
```

A.8 La liste des fichiers de l'environnement standard

```
(de all-the-files ()
    '(      ;
            ; 1 - Système minimum
            ;
       (virtty virbitmap)
            ;
            ; 2 - Emacs ou PEPE
            ;
       (pepe)
            ;
            ; 3 - Environnement standard
            ;
       (array defstruct sort callext pretty trace debug)
            ;
            ; 4 - Le chargeur
            ;
       (#.(selectq (system)
                ((micromega sm90 metheus
                  apollo vme cadmus sun hp9000)    'lap68k)
                ((vaxunix vaxvms) 'lapvax)
                ((pe32unix pe32os) 'lape32)
                ((multics) 'lapmultics)
                (bell 'lapbell)
                (sps9 'lapsps9)
                (pcdos 'lap86)
                (t ())))
           ;
           ; 5 - Compilateur
           ;
       (llcp)))
```

A.9 Construction de l'environnement standard

```
(de save-std (nom msg . save-std)
    ; sauve une image standard de type "save-core" de nom nom
    ; msg sera la bannière de rappel de l'image
    ; save-std est la liste des utilitaires chargés.
    (setq msg (catenate msg
                (if (cadar save-std) " editeur," "")
                (if (caddar save-std) " environnement," "")
                (if (cadr (cddar save-std)) " chargeur")
                (if (caddr (cddar save-std)) " et compilateur.")))
    (print "Attendez, je sauve : " msg)
    (gc)
    (save-core (catenate #:system:core-directory
                         nom
                         #:system:core-extension))
    (initty)
    (inibitmap)
    (herald)
    #.(selectq (system)
          (#.unix
            '(let ((f (catenate (getenv "HOME") "/.lelisp")))
                  (when (probefile f)
                        (loadfile f t))))
           (vaxvms
            '(let ((f "sys$login:startup.ll"))
                  (when (probefile f)
                        (loadfile  f t))))
           (multics
             '(let ((f (catenate (getenv "HOME") ">start_up.ll")))
                   (when (probefile f)
                         (loadfile f t))))
           (t ()))
    msg)
(de load-std (nom . load-std)
    ; charge l'image standard
    ; 'nom' indique s'il faut faire une image mémoire (et ca devient le nom)
    ; 'load-std' est la liste des indicateurs
    ; (min editeur environ compilateur)
    (mapc (lambda (lf i)
            (when i
               (mapc (lambda (x)
                             (setq x (catenate #:system:llib-directory
                                               x
                                               #:system:lelisp-extension))
                             (print "Je charge " x)
                             (loadfile x t))
                     (or (consp i) lf))))
          (all-the-files)
          load-std)
     (setq #:system:load-feature load-std)
```

```
     (print " (llcp-std '<nom>)   pour compiler l'environnement standard")
     (when nom (save-std nom
                         "Systeme standard interprete,"
                         #:system:load-feature)))

(de llcp-std (nom)
    ; compilation de l'environnement standard
    ; et fabrication du core "nom"
    (when (typefn 'compiler)
          ; essaie de dégager un peu le compile-all-in-core
          (and (eq (typefn 'pprint) 'expr) (compiler 'pprint t))
          (and (eq (typefn 'loader) 'expr)
               (let ((#:ld:special-case-loader t))
                    (compiler 'loader t)))
          (and (eq (typefn 'pepefile) 'expr) (compiler 'pepefile t))
          (mapc 'remob '(all-the-files load-std load-cpl llcp-std))
          (compile-all-in-core)
          (when nom
              (save-std nom
                    "Systeme standard compile,"
                    #:system:load-feature))))

(de load-cpl (nom . load-std)
    ; charge l'image standard
    ; 'nom' indique s'il faut faire une image mémoire (et ca devient le nom)
    ; 'load-std' est la liste des indicateurs
    ; (min editeur environ loader compilateur)
    ; Il faut au minimum le code du chargeur et celui des modules.
    (libloadfile (car (cadddr (all-the-files))) t)
    (libload module t)
    ; On charge les fichiers indispensables.
    (mapc
        (lambda (m)
           (print "Je charge " (probepathm m))
           (unless (memq m #:module:compiled-list) (loadmodule m)) )
       '(loader toplevel files module defs) )
    ; On charge le compilateur en premier s'il doit etre chargé
    (setq load-std (cons (car (cddddr load-std)) load-std))
    ; Et maintenant le reste.
    (mapc (lambda (lf i)
            (when i
               (mapc (lambda (m)
                             (print "Je charge " (probepathm m))
                             (unless
                                (or (memq m #:module:compiled-list)
                                    (memq m #:module:interpreted-list) )
                                (loadmodule m) ))
                     (or (consp i) lf))))
          '((complice)
            (virtty virbitmap) (pepe) (defstruct sort array callext
                             trace pretty debug) ())
          load-std)
     ; On vire ce qui l'y a en trop.
```

```
(setq #:system:load-feature load-std)
(mapc 'remob '(all-the-files load-std load-cpl llcp-std))
(compile-all-in-core)
(when nom (save-std nom
                    "Systeme modulaire,"
                    #:system:load-feature)))
```

A.10 Fin de chargement de l'environnement minimum

```
(loadfile
   (catenate #:system:llib-directory 'toplevel #:system:lelisp-extension)
   () )
(loadfile
   (catenate #:system:llib-directory 'files #:system:lelisp-extension)
   () )
(loadfile
   (catenate #:system:llib-directory 'defs #:system:lelisp-extension)
   () )
(loadfile
   (catenate #:system:llib-directory 'genarith #:system:lelisp-extension)
   () )
(libautoload debug debug)
(libautoload module loadmodule unloadmodule readdefmodule)
```

A.11 Les fonctions Autoloads

```
 ; 0 - les fonctions standard
(libautoload virtty initty)
(libautoload virbitmap inibitmap)
(libautoload topwin topwindow)
 ; 1 - les utilitaires
(libautoload defstruct defstruct)
(libautoload array makearray aref aset)
(libautoload sort sort sortl sortn sortp)
(libautoload callext defextern cload)
(libautoload trace trace untrace step)
(libautoload schedule parallel parallelvalues tryinparallel)
(libautoload pretty pretty pprint prettyf)
(libautoload format format)
(libautoload setf setf)

 ; 2 - les éditeurs
(libautoload minimore more morend)
(libautoload edlin edlin edlinend)
(libautoload emacs emacs)
```

```
(libautoload pepe pepe pepefile)

 ; 3 - les jeux
(libautoload hanoi hanoi)
(libautoload whanoi whanoi)
(libautoload vdt vdt)
 ; 4 - compilateur et chargeur
  ; Ce #. est non portable.
(libautoload
        #.(selectq (system)
                ((micromega sm90 metheus
                  apollo vme sun cadmus hp9000) 'lap68k)
                ((vaxunix vaxvms) 'lapvax)
                ((pe32unix pe32os) 'lape32)
                ((multics) 'lapmultics)
                (bell 'lapbell)
                (sps9 'lapsps9)
                (pcdos 'lap86)
                (t ()))
       loader loaderesolve)
(libautoload llcp compile compilefiles compile-all-in-core)
```

A.12 Final de l'initialisation

```
(defvar #:system:loaded-from-file ())
(rmargin 78)
(print
   " (load-std sav min edit env ld llcp)   pour charger l'environnement std,"
   " (load-cpl sav min edit env ld llcp)   pour l'environnement modulaire.")
   'startup)
(input () )
```

ANNEXE B

Les Manipulations de Fichiers

Jérôme Chailloux
Bernard Serpette

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'errifc 'files))
(defvar #:sys-package:colon 'system)
```

B.1 Gestion des FEATURES

```
(defvar #:system:features-list ())
(de featurep (feature)
    (memq feature #:system:features-list))
(de add-feature (feature)
    (unless (featurep feature)
            (newl #:system:features-list feature)))
(de rem-feature (feature)
    (setq #:system:features-list (delq feature #:system:features-list)))
```

B.2 Load et Autoload

```
(defvar :previous-def-flag ())
```

B.2.1 Test sur les fichiers

```
(de probepathf (file)
    ; cherche si le fichier {PATH}file[.ll] existe.
    (search-in-path :path (suffixe file :lelisp-extension)))
(de probepathm (file)
    ; cherche si le fichier {PATH}file[.lm] existe.
    (search-in-path :path (suffixe file :mod-extension)))
(de probepatho (file)
```

```
    ; cherche si le fichier {PATH}file[.lo] existe.
    (search-in-path :path (suffixe file :obj-extension)))

(de suffixe (file suff)
    (if (eq (index suff file) (sub (slength file) (slength suff)))
        file
        (catenate file suff) ))

(de search-in-path (path file)
    ; cherche si le fichier {PATH}file existe.
    (when path
          (let ((real-file (catenate (if (consp path) (car path) path) file)))
               (if (probefile real-file)
                   real-file
                   (when (consp path) (search-in-path (cdr path) file))))))
```

B.2.2 des fichiers simples

```
(df load (file . redef?)
    (loadfile file (car redef?)) )

(de loadfile (file redef?)
    (ifn (probefile file)
         (error 'loadfile 'errfile file)
         (let ((:loaded-from-file file)
               (:redef-flag redef?)
               (#:sys-package:colon #:sys-package:colon)
               (:in-read-flag ())
               (inchan (inchan)) )
              (inchan (openi file))
              (protect (untilexit eof (eval (read)))
                       (let ((in (inchan)))
                            (when in (close in)))
                       (inchan inchan) ))
         file ))
```

B.2.3 des fichiers des bibliothèques

```
(df libload (file . redef?)
    (libloadfile file redef?) )

(de libloadfile (file redef?)
    (let ((real-file (probepathf file)))
         (ifn real-file
              (error 'libloadfile 'errfile file)
              (loadfile real-file redef?) )))
```

B.2.4 des fichiers compilés

```
(de loadobjectfile (file)
    (let ((file-obj? (probepatho file)))
         (ifn file-obj?
              (error 'loadobjectfile 'errfile file)
              (loadfile file-obj? t) )))
```

B.2.5 des fichiers autoload

```
(df autoload (file . lfnt)
    ; définition de fonctions autoload (autoload mod at1 ... atN)
    (mapc (lambda (fnt)
                  (setfn fnt 'fexpr
                         `(:b (:std-autoload ',fnt :b 'loadfile)))
                  (putprop fnt file 'autoload))
          lfnt))

(df libautoload (file . lfnt)
    ; définition de fonctions autoload (libautoload mod at1 ... atN)
    (mapc (lambda (fnt)
                  (setfn fnt 'fexpr
                         `(:b (:std-autoload ',fnt :b 'libloadfile)))
                  (putprop fnt file 'autoload))
          lfnt))

(de :std-autoload (:f :b :load-fct)
    (let ((:of (valfn :f))
          (:ot (typefn :f))
          (:autoload-file (getprop :f 'autoload)))
         (cond ((and
                   (catcherror (funcall :load-fct :autoload-file t))
                   (neq :of (valfn :f)))
                (remprop :f 'autoload)
                (eval (cons :f :b)))
               (t
                 (setfn :f :ot :of)
                 (error 'autoload 'errudf :f)))))
```

ANNEXE C

Le Nouveau Toplevel

Jérôme Chailloux

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'toplevel))
```

C.1 Le nouveau toplevel et le gestionnaire d'erreur

;----- La mécanique des interruptions programmables

```
(defvar #:sys-package:itsoft ())   ; le package des interruptions
```

;----- La mécanique de traitement des erreurs

```
(defvar #:system:debug ())              ; On debugge
(defvar #:trace:trace ())               ; les fonctions tracees

(defvar #:system:print-msgs 1)          ; l'OS imprime ses messages
(defvar #:system:error-flag ())         ; Impression des messages

(df catcherror (#:system:error-flag . #:system:l)
    ; le récupérateur d'erreur "à la Le_Lisp"
    (let ((#:system:error-flag (eval #:system:error-flag)))
         (let ((#:system:print-msgs
                    (if #:system:error-flag #:system:print-msgs 0))
               (#:system:debug
                    (if #:system:error-flag #:system:debug ())))
              (tag #:system:error-tag
                    (ncons (eprogn #:system:l))))))

(dmd errset (#:system:e #:system:i)
     ; le récupérateur d'erreur "à la Maclisp"
     (list 'catcherror #:system:i #:system:e))

(dmd err #:system:l
     ; l'argument est un PROGN à évaluer
     (mcons 'exit '#:system:error-tag #:system:l))
```

;----- la fonction break Elle sera redéfinie dans le debugger

```
(de break () (err))
```

;----- l'interruption syserror

```
(defvar #:system:f ())
(defvar #:system:m ())
(defvar #:system:b ())
(de syserror (#:system:f #:system:m #:system:b)
    ; 1 - vide le tampon d'entrée
    (teread)
    (tyflush)
    ; 2 - imprime le message sur TTY
    (when (and #:system:error-flag (null #:system:debug))
          (let ((outchan (outchan)))
               (outchan ())
               (protect
                   (printerror #:system:f #:system:m #:system:b)
                   (outchan outchan))))
    (break))
```

;--- le toplevel proprement dit

```
(defvar #:toplevel:status t)    ; toplevel interactif
(defvar #:toplevel:read ())     ; dernier objet lu
(defvar #:toplevel:cread ())    ; objet courant
(defvar #:toplevel:print ())    ; dernière forme imprimée
(defvar #:toplevel:eval ())     ; dernière valeur retournée
(dmd #:toplevel:topeval (e)
     ; évalue une expression au toplevel
     (list 'tag '#:system:error-tag
           (list 'lock '(lambda (tag val)
                     (cond ((null tag)
                            (ncons val))
                           ((eq tag '#:system:error-tag) ())
                           ((eq tag '#:system:toplevel-tag)
                            (exit #:system:toplevel-tag))
                           (t (error 'toplevel
                                     'errudt
                                     tag))))
                 e)))
(de toplevel ()
    (tag #:system:toplevel-tag
         (setq ; quelques réglages :
               #:system:error-flag t
               ; l'ancienne forme lue
               #:toplevel:read #:toplevel:cread
               ; la nouvelle forme lue (en liste par TOPEVAL)
               #:toplevel:cread (#:toplevel:topeval (read))
               #:toplevel:cread
               (when (consp #:toplevel:cread) (car #:toplevel:cread))
```

```
            ; la valeur de l'évaluation
            #:toplevel:eval (#:toplevel:topeval (eval #:toplevel:cread)))
        (when (and (consp #:toplevel:eval)
                   #:toplevel:status)
            (tyflush)           ;  pour ne pas mélanger les TYO éventuels
            (princn #/=)
            (princn #\sp)
            (#:toplevel:topeval (print (car #:toplevel:eval))))
        (setq #:toplevel:eval (car #:toplevel:eval))))
```

petit réglage du print

```
(printline 5000)
```

C.2 Les impressions du gestionnaire d'erreur

```
(de printerror (#:system:f #:system:m #:system:b)
    ; imprime un message d'erreur
    ;  -  pour la fonction ‹f›
    ;  -  le type d'erreur ‹m›
    ;  -  l'argument défectueux ‹b›
    (print "** " #:system:f
           " : " (selectq #:system:m
                      ; les erreurs standard
                      (errmac errmac)
                      (errudv errudv)
                      (errudf errudf)
                      (errudm errudm)
                      (errudt errudt)
                      (errbdf errbdf)
                      (errwna errwna)
                      (errbpa errbpa)
                      (errilb errilb)
                      (errbal errbal)
                      (errnab errnab)
                      (errxia errxia)
                      (errsxt errsxt)
                      (errios errios)
                      (err0dv err0dv)
                      (errnna errnna)
                      (errnia errnia)
                      (errnfa errnfa)
                      (errnsa errnsa)
                      (errnaa errnaa)
                      (errnla errnla)
                      (errnva errnva)
                      (errvec errvec)
                      (errsym errsym)
                      (errnda errnda)
                      (errstc errstc)
                      (erroob erroob)
```

```
                (errstl errstl)
                (errgen errgen)
                (errvirtty errvirtty)
                (errfile errfile)
                (erricf erricf)
                (t          #:system:m))
        " : " (cond
                ((and (eq #:system:m 'errsxt)
                      (numberp #:system:b)
                      (> #:system:b 0) (< #:system:b 13))
                 (selectq #:system:b
                      (1   "liste trop courte")
                      (2   "chaine trop longue")
                      (3   "symbole trop long")
                      (4   "mauvais debut d'expression")
                      (5   "symbole special trop long")
                      (6   "mauvais package")
                      (7   "mauvaise construction pointee")
                      (9   "mauvaise liste argument")
                      (10  "mauvaise valeur de splice-macro")
                      (11  "EOF durant un READ")
                      (12  "mauvaise utilisation du backquote")
                      (t   #:system:b)))
                ((and (memq (system) unix)
                      (eq #:system:m 'errmac)
                      (numberp #:system:b))
                 (selectq #:system:b
                      (4   "instruction illegale")
                      (8   "exception flottante")
                      (10  "erreur de bus")
                      (11  "violation de segment")
                      (t #:system:b)))
                (t #:system:b))))
```

ANNEXE D

Les Fonctions de Définition et Macros Système

Jérôme Chailloux
Matthieu Devin

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'defs))
```

Toutes les variables doivent être protégées :

```
(defvar #:sys-package:colon 'system)
(defvar :redef-flag t)
```

D.1 Tests des Macros Fonctions

```
(de macroexpand (:x)
    ; expanse un appel tant que l'on tombe sur une macro
    (cond ((atom :x) :x)
          ((symbolp (car :x))
             (cond ((eq (typefn (car :x)) 'macro)
                    (macroexpand (apply (cons 'lambda (valfn (car :x))) :x)))
                   ((eq (typefn (car :x)) 'dmacro)
                    (macroexpand
                       (apply (cons 'lambda (valfn (car :x))) (cdr :x))))
                   ((eq (typefn (car :x)) 'msubr)
                    (macroexpand
                       (call (valfn (car :x)) :x () ())))
                   ((eq (typefn (car :x)) 'dmsubr)
                    (macroexpand
                       (call (valfn (car :x)) (cdr :x) () ())))
                   ((eq (car :x) 'quote) :x)
                   (t (let ( (l ()) )
                         (while (consp :x) (newl l (macroexpand (nextl :x))))
                         (prog1 (nreverse l) (rplacd l :x)) ))))
          (t (let ( (:l ()) )
               (while (consp :x) (newl :l (macroexpand (nextl :x))))
               (prog1 (nreverse :l) (rplacd :l :x)) ))))
```

```
(de macroexpand1 (:x)
    ; expanse une macro un coup pour voir
    (cond ((eq (typefn (car :x)) 'macro)
           (apply (cons 'lambda (valfn (car :x))) :x))
          ((eq (typefn (car :x)) 'dmacro)
           (apply (cons 'lambda (valfn (car :x))) (cdr :x)))
          ((eq (typefn (car :x)) 'msubr)
           (call (valfn (car :x)) :x () ()))
          ((eq (typefn (car :x)) 'dmsubr)
           (call (valfn (car :x)) (cdr :x) () ()))
          (t :x)))
```

D.2 WITH : les variables fonction

Liaison dynamique des EXPR en lecture/écriture.

```
(dmd with (:l . :body)
    (let ((:var (let ((n -1))
                  (mapcar (lambda (x)
                            (symbol 'with (concat "arg" (incr n))))
                          :l))))
         `(let (,@(mapcar (lambda (:var :l)
                            (list :var
                                  (firstn (1- (length :l))
                                          :l)))
                          :var
                          :l))
              (protect
                  (progn ,@:l ,@:body)
                  ,@(mapcar (lambda (:var :l)
                              (append1 (firstn (1- (length :l))
                                               :l)
                                       :var))
                            :var
                            :l)))))
```

D.3 Les fonctions auxiliaires de test

```
(de :def-check-all (namelarg fnt)
    ; teste s'il y a bien deux arguments au moins
    (unless (consp (cdr namelarg))
            (error fnt 'errnla (cdr namelarg)))
    (let ((name (car namelarg)) (larg (cadr namelarg)))
         ; teste toute une définition :
         ; de nom : name
         ; de liste d'argument : larg
         ; le nom de la fonction de définition est : fnt
```

```
        ; 1 - test le nom
        (unless (variablep name)
                (error fnt 'errbdf name))
        ; 2 - test la liste des paramètres
        (:def-check-larg larg name)
        ; 3 - test la redéfinition
        (when (and (not :redef-flag)
                   (typefn name))
              (print "** " fnt " : fonction redefinie : " name))
        ; 4 - detrace si la fonction était tracée
        (when (memq name #:trace:trace)
              (eval `(untrace ,name)))
        ; 5 - sauve l'ancienne définition
        (when (and :previous-def-flag
                   (typefn name))
              (putprop name
                  (getdef name)
                  ':previous-def))
        ; 6 - positionne l'indicateur loaded-from-file
        (when :loaded-from-file
              (putprop name
                  :loaded-from-file
                  ':loaded-from-file))))
(de :def-check-larg (l fnt)
    ; provoque ERRBPA si la liste "l"
    ; contient autre chose que des paramètres.
    (cond ((null l) t)
          ((variablep l) t)
          ((and (consp l)
                (car l)
                (:def-check-larg (car l) fnt)
                (:def-check-larg (cdr l) fnt))
           t)
          (t (error fnt 'errbpa l))))
```

D.4 Redéfinitions

```
(de :resetfn (symbol typefn lambda)
    (let ((l ()))
         (newl l `(fentry ,symbol ,typefn))
         (when (eq typefn 'fsubr)
               (newl l `(push ',lambda)))
         (when (memq typefn '(subr1 subr2 subr3 fsubr))
               (newl l '(push a1)))
         (when (memq typefn '(subr2 subr3))
               (newl l '(push a2)))
         (when (eq typefn 'subr3)
               (newl l '(push a3)))
         (unless (eq typefn 'fsubr)
                 (newl l `(mov ',lambda a1)))
```

```
        (unless (eq typefn 'nsubr)
                (newl l
                      `(mov ',(selectq typefn
                                       (subr0 0) (subr1 1) (subr2 2) (subr3 3)
                                       (fsubr 2))
                            a4)))
        (if (eq typefn 'fsubr)
            (newl l '(jmp apply))
            (newl l '(jmp #:llcp:ffuncall)))
        (newl l '(end))
        (loader (nreverse l) ()))))

(de resetfn (symbol typefn valfn)
    (let ((otypefn (typefn symbol)))
        (if (and (memq otypefn '(subr0 subr1 subr2 subr3 nsubr fsubr))
                 (featurep 'loader)) ; il faut le loader
            (cond ((eq otypefn typefn))
                  ((and (eq 'expr typefn)
                        (selectq otypefn
                            (subr0
                                  (null (car valfn)))
                            (subr1
                                  (and (consp (car valfn))
                                       (null (cdr (car valfn)))))
                            (subr2
                                  (and (consp (car valfn))
                                       (consp (cdr (car valfn)))
                                       (null (cddr (car valfn)))))
                            (subr3
                                  (and (consp (car valfn))
                                       (consp (cdr (car valfn)))
                                       (consp (cddr (car valfn)))
                                       (null (cdddr (car valfn)))))
                            (nsubr
                                  (or (variablep (car valfn))
                                      (variablep (cdr (last (car valfn))))
                                      (ge (length (car valfn)) 4))))))
                   (:resetfn symbol otypefn (cons 'lambda valfn))
                   (putprop symbol (cons typefn valfn) 'resetfn))
                  ((and (eq 'fexpr typefn)
                        (eq 'fsubr otypefn))
                   (:resetfn symbol otypefn (cons 'flambda valfn))
                   (putprop symbol (cons typefn valfn) 'resetfn))
                  (t
                    (when (and #:system:redef-flag
                               (null (getprop symbol 'autoload)))
                          (print
                                "** " 'resetfn
                                " : fonction incompatible : " symbol))
                    (setfn symbol typefn valfn)))
            (setfn symbol typefn valfn)
            (when (and (featurep 'compiler)
                       (macro-openp symbol))
```

```
                (remove-macro-open symbol))))
  symbol)
```

D.5 Redéfinition des fonctions de définition

```
(setfn 'de 'fexpr
    '(:l
       (:def-check-all :l 'de)
       (resetfn (car :l) 'expr (cdr :l))))

(setfn 'df 'fexpr
    '(:l
       (:def-check-all :l 'df)
       (resetfn (car :l) 'fexpr (cdr :l))))

(setfn 'dm 'fexpr
    '(:l
       (:def-check-all :l 'dm)
       (setfn (car :l) 'macro (cdr :l))))

(setfn 'dmd 'fexpr
    '(:l
       (:def-check-all :l 'dmd)
       (setfn (car :l) 'dmacro (cdr :l))))

(setfn 'defmacro 'fexpr
    '(:l
       (:def-check-all :l 'defmacro)
       (setfn (car :l) 'dmacro (cdr :l))))

(setfn 'defun 'fexpr
    '(:l
       (:def-check-all :l 'defun)
       (resetfn (car :l) 'expr (cdr :l))))

(setfn 'ds 'fexpr
    '(:l
       (:def-check-all :l 'ds)
       (setfn (car :l) (cadr :l) (caddr :l))))
```

D.6 Les LET en tout genre

D.6.1 LETN : des LET avec nom

```
(dmd letn (:name :larg . :body)
      `(flet ((,:name ,(mapcar 'car `,:larg) ,@:body))
             (,:name ,@(mapcar 'cadr `,:larg)))))))))
```

D.6.2 LETS/SLET/LET* : le LET sequeniel

```
(dmd slet (l . :body)
     (cond ((null l) `(let () ,@:body))
           ((cdr l) `(let (,(car l)) (lets ,(cdr l) ,@:body)))
           (t `(let (,(car l)) ,@:body))))
(synonym 'lets 'slet)
(synonym 'let* 'slet)
```

D.7 Simulation des structures de contrôle à la PASCAL

syntaxe : (for (var init step end e1 ... eN) <suite_de_S-expressions>)
La variable var est déclarée localement au corps du for par un let et varie de init à end (inclus). Le résultat est la valeur de (progn e1 ... eN)

```
(de :generate-for (stepvar endvar)
    ; fabrique un corps de FOR
    `(let ((,:var ,init)
           ,@(if stepvar `((,stepvar ,step)) (setq stepvar step) ())
           ,@(if endvar  `((,endvar  ,end))  (setq endvar  end)  ()))
          (cond ((> ,stepvar 0)
                 (while (<= ,:var ,endvar)
                        ,@:body
                        (incr ,:var ,stepvar)))
                ((< ,stepvar 0)
                 (while (>= ,:var ,endvar)
                        ,@:body
                        (incr ,:var ,stepvar)))
                (t (error 'for "increment nul" 0)))
          ,@res))
(dmd for ((:var init step end . res) . :body)
     (unless (variablep :var)
             (error 'for 'errnva :var))
     (unless (and init step end)
          (error 'for 'errsxt (list :var init step end)))
     (if (numberp step)
         (let ((test (cond ((> step 0) '<=)
                           ((< step 0) '>=)
                           (t (error 'for "increment nul" 0)))))
              (if (or (numberp end) (symbolp end))
```

```
                `(let ((,:var ,init))
                      (while (,test ,:var ,end)
                             ,@:body
                             (incr ,:var ,step))
                      ,@res)
                (let ((endvar '#:system:for:arg2))
                     `(let ((,:var ,init)
                            (,endvar ,end))
                           (while (,test ,:var ,endvar)
                                  ,@:body
                                  (incr ,:var ,step))
                           ,@res))))
          (if (or (numberp end) (symbolp end))
              (if (symbolp step)
                  (:generate-for () ())
                  (:generate-for '#:system:for:arg1 ()))
              (if (symbolp step)
                  (:generate-for () '#:system:for:arg2)
                  (:generate-for '#:system:for:arg1 '#:system:for:arg2)))))
```

D.8 Fermetures

(closure ‹lvar› ‹fnt›) à la "lisp-machine"
‹lvar› liste de variables à clôturer
‹fnt› une fonction LAMBDA ou LAMBDA-NAMED

```
(de closure (lvarclot :f)
    (let ((listval (mapcar '(lambda (val) (list 'quote (eval val))) lvarclot))
          (:lvar (cadr :f))
          (:body (cddr :f)))
         `(lambda ,:lvar
              ((lambda ,lvarclot
                   (protect (progn ,@:body)
                        ,@(mapcar '(lambda (slot :var)
                                           `(rplaca (cdr ,`(quote ,slot))
                                                    ,:var))
                                  listval  lvarclot)))
               ,@listval))))
```

;----- Utilisation des fermetures

```
; (setfn 'nextint 'expr
;        (let ((n 1))
;             (cdr (closure '(n) '(lambda () (setq n (1+ n))))))))))
; (setfn 'fib 'expr
;        (let ((x 1) (y 1))
;             (cdr (closure '(x y)
;                           '(lambda () (setq y (prog1 (+ x y) (setq x y))))))))))
```

D.9 Simulation des structures de contrôle à la FORTRAN!

;----- PROG/PROG*/RETURN/DO/DO*

```
(de :generate-tagbody (:body)
    (when :body
          (if (every 'consp :body)
              :body
              (list (cons 'tagbody :body)))))

(de :generate-block (:body :l)
    ; ?!?!? vérifier que dans les parties valeurs, test et valeur de retour
    ; ?!?!? des DO... il n'est pas possible d'avoir des RETURN/RETURN-FROM
    ; ?!?!? pbs également si RETURN apparait dans une macro!
    (if (tag ok (:generate-block-aux :body))
        `(block () ,:l)
        :l))

(de :generate-block-aux (:body)
    (cond ((atom :body) ())
          ((memq (car :body) '(return return-from)) (exit ok t))
          (t (any ':generate-block-aux :body))))

(dmd prog :l
     ; autorise la forme (prog)
     (:generate-block (cdr :l)
          (if (car :l)
              `(let ,(car :l) ,@(:generate-tagbody (cdr :l)))
              `(progn ,@(:generate-tagbody (cdr :l))))))

(dmd prog* :l
     ; autorise la forme (prog*)
     (:generate-block (cdr :l)
          (if (car :l)
              `(let* ,(car :l) ,@(:generate-tagbody (cdr :l)))
              `(progn ,@(:generate-tagbody (cdr :l))))))

(dmd do (:lvar (:test . :result) . :body)
     (:generate-block :body
             `(let ,(mapcar (lambda (x)
                                    (list (car x) (cadr x)))
                            :lvar)
                   (until ,:test
                          ,@(:generate-tagbody :body)
                          ,@(let ((:x (mapcan
                                        (lambda (x)
                                           (when (consp (cddr x))
                                                 (list (car x) (caddr x))))
                                        :lvar)))
                                 (when :x `((psetq ,@:x)))))
                   ,@:result)))

(dmd do* (:lvar (:test . :result) . :body)
     (:generate-block :body
```

```
        `(let* ,(mapcar (lambda (x)
                          (list (car x) (cadr x)))
                        :lvar)
           (until ,:test
                  ,@(:generate-tagbody :body)
                  ,@(let ((:x (mapcan
                                (lambda (x)
                                  (when (consp (cddr x))
                                        (list (car x) (caddr x))))
                                :lvar)))
                      (when :x `((setq ,@:x)))))
           ,@:result)))
```

D.10 BACKTRACK : on se croirait en prolog

```
(dmd backtrack (:name :lvar . :body)
     (unless (symbolp :name)
             (error 'backtrack 'errnaa :name))
     `(tag :backtrack
           ,@(if (null :lvar)
                 (mapcar (lambda (e)
                           `(tag ,:name ,e (exit :backtrack)))
                         :body)
                 `((let ((:backtrack (list ,@:lvar)))
                     ,@(mapcan (lambda (e)
                                 `((tag ,:name
                                        ,e (exit :backtrack))
                                   (desetq ,:lvar
                                           :backtrack)))
                               :body))))))
```

D.11 le catch-all-but disparu au profit de lock ...

```
(dmd catch-all-but (tag . :body)
   `(lock (lambda (tag val)
            (cond ((null tag) val)
                  ((memq tag ',tag) (evexit tag val))
                  (t (error 'catch-all-but errudt tag))))
          ,@:body))
```

ANNEXE E

L'Arithmétique Générique mixte

Jérôme Chailloux

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'genarith))
```

Le chargement de ce fichier transforme les appels de l'arithmétique générique en arithmétique mixte, supprimant ainsi les erreurs ERRGEN en cas de débordement entier.

Seules les méthodes : + - 0- * / 1/ ont un sens ici.

```
(de #:genarith:+ (n1 n2)
    (if (and (fixp n1) (fixp n2))
        (fadd (float n1) (float n2))
        (#:genarith:error '+ n1 n2)))

(de #:genarith:- (n1 n2)
    (if (and (fixp n1) (fixp n2))
        (fsub (float n1) (float n2))
        (#:genarith:error '- n1 n2)))

(de #:genarith:0- (n)
    (if (fixp n)
        (fsub 0.0 (float n))
        (#:genarith:error '0- n)))

(de #:genarith:* (n1 n2)
    (if (and (fixp n1) (fixp n2))
        (fmul (float n1) (float n2))
        (#:genarith:error '* n1 n2)))

(de #:genarith:/ (n1 n2)
    (if (and (fixp n1) (fixp n2))
        (fdiv (float n1) (float n2))
        (#:genarith:error '/ n1 n2)))

(de #:genarith:1/ (n)
    (if (fixp n)
        (fdiv 1.0 (float n))
        (#:genarith:error '1/ n)))

(defvar #:ex:mod 0)
```

```
(de #:genarith:quomod (n1 n2)
    (let ((q (floor (/ n1 (abs n2)))))
         (setq #:ex:mod (- n1 (* (abs n2) q)))
         (if (< n2 0) (- q) q)))

(de #:genarith:error (f . l)
    (while l
           (if (and (fixp (car l)) (floatp (car l)))
               (nextl l)
               (error f 'ERRNNA (car l)))))

(defvar #:sys-package:genarith 'genarith)
```

Réglage de la fonction QUOTIENT/QUOMOD canonique (courtesy of Jean Vuillemin)

```
(de floor (r)
    ; Renvoie le plus grand entier relatif z tel que z<=r:
    (ifn (numberp r)
         (error 'floor 'ERRNNA r)
         (let ((z (truncate r)))
              (if (> z r)
                  (- z 1)
                  z))))
```

ANNEXE F

Les Tableaux

Jérôme Chailloux

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'array))
```

Toutes ces fonctions sont "autoload" dans le système minimum. ce qui est bien pratique

```
(de makearray (arg1 . #:system:larg)
    ; fabrique un tableau (d'au moins une dimension)
    (if #:system:larg
        (let ((result (makevector arg1 ())))
             (for (i 0 1 (1- arg1))
                  (vset result i (apply 'makearray #:system:larg)))
             result)
        arg1))
```

```
(dmd aref (inst . #:system:larg)
     ; accès à un élément d'un tableau
     (cond ((null #:system:larg) inst)
           ((atom #:system:larg) (error 'aref 'errwna #:system:larg))
           (t `(vref (aref ,inst ,.(nreverse (cdr (reverse #:system:larg))))
                     ,(car (last #:system:larg))))))

(dmd aset (inst . #:system:larg)
     ; écriture d'un élément d'un tableau
     (cond ((atom #:system:larg) (error 'aset 'errwna #:system:larg))
           ((consp (cdr #:system:larg))
              `(vset (aref ,inst ,.(nreverse (cddr (reverse #:system:larg))))
                     ,(cadr (reverse #:system:larg))
                     ,(car (last #:system:larg))))
           (t inst)))
```

ANNEXE G

Les Structures

Jean Marie Hullot
Jérôme Chailloux

Un defstruct très simple, tiré de celui d'Alcyone avec :
- l'utilisation des vecteurs typés
- fonction d'accès sous forme d'EXPR qui ne consent pas
- fonction de création : #:type:make
- compatibilité d'utilisation avec Ceyx (Objval et TCONS libres)
- utilisation des P-listes des types.

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'defstruct))
```

Tous les symboles précédés de : seront créés dans le package DEFSTRUCT

```
(defvar #:sys-package:colon 'defstruct)
(add-feature 'defstruct)
```

G.1 Le type STRUCTURE

La Pliste d'une structure contient, sous l'indicateur defstruct un CONS qui contient :
- le segment de valeur d'appel à la fonction VECTOR
  pour créer le segment de vecteur de ce type simple.
- la liste des noms des champs à la définition (pour la
  lecture et l'écriture de la syntaxe #S(nom val .....)

```
(de structurep (x)
    (and (vectorp x)
         (getprop (typevector x) 'defstruct)))
```

G.2 La fonction de définition de structure

```
(defmacro defstruct (name . lfield)
  ; pose de la valeur initiale () par défaut
  (unless (variablep name)
          (error 'defstruct 'errnva name))
  (let ((name1 name)
        (index -1)
        (make (mapcar (lambda (x)
                              (cond ((symbolp x) ())
```

```
                                ((or (atom x)
                                     (not (symbolp (car x))))
                                 (error 'defstruct 'errnaa x))
                                (t (cadr x))))
                    lfield))
     (lfieldt (mapcar (lambda (x)
                              (if (atom x) x (car x)))
                      lfield)))
    (until (or (eq (packagecell name1) '||)
               (null (getprop (packagecell name1) 'defstruct)))
           (setq name1 (packagecell name1))
           (setq make (append (car (getprop name1 'defstruct)) make))
           (setq lfieldt (append (cdr (getprop name1 'defstruct))
                                 lfieldt)))
    `(progn
            (putprop ',name ',(cons make lfieldt) 'defstruct)
            (de ,(symbol name 'make) ()
                 (let ((vector (vector ,@make)))
                      (typevector vector ',name)
                      vector))
           ,@(mapcan
                    (lambda (f)
                            (setq f (symbol name f))
                            (incr index)
                            `((de ,f &nobind
                                  (#:system:structaccess ',f ,index
                                   (arg 0)
                                   (when (eq (arg) 2)
                                         (arg 1))
                                   (arg)))
                              (when (featurep 'compiler)
                                    (defmacro-open
                                        ,f (struct . valeur)
                                        (ifn valeur
                                             `(vref ,struct ,,index)
                                             `(vset ,struct
                                                    ,,index
                                                    ,(car valeur))))
                              )))
                    lfieldt)
           ',name)))
```

G.3 La fonction de création générique

```
(de new (type)
    (if (getprop type 'defstruct)
        (apply (symbol type 'make) ())
        (error 'new 'errstc type)))
```

ANNEXE H

Les Tris physiques

Jérôme Chailloux
Olivier Guillaumin

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'sort))
```

Toutes ces fonctions sont "autoload" dans le système minimum.

```
(de sort (fn l)
    ; trie (au moyen de la fonction fn à 2 arguments) la liste l
    (if (null (cdr l))
        l
        (let ((l1 l) (l2))
             (setq l  (nthcdr (1- (div (length l) 2)) l)
                   l2 (cdr l))
             (rplacd l ())
             (ffusion (sort fn l1)
                      (sort fn l2)))))

(de ffusion (l1 l2)
    ; fusionne physiquement les 2 listes triées l1 et l2
    (unless (funcall fn (car l1) (car l2))
            (psetq l1 l2 l2 l1))
    (prog1 l1
           (while (and (cdr l1) l2)
                  (when (funcall fn (car l2) (cadr l1))
                        (rplacd l1 (prog1 l2 (setq l2 (cdr l1)))))
                  (nextl l1))
           (when l2 (rplacd l1 l2))))
```

Les tris prédéfinis

```
(de sortl (l)
    ; tri classique alphabétique.
    (sort 'alphalessp l))

(de sortn (l)
    ; tri numérique
    (sort '< l))
```

```
(de sortp (l)
    ; tri sur les noms des symboles avec les packages
    (sort 'pkgcmp l))

(de pkgcmp (a b)
    ; compare 2 symboles en considérant que
    ; les packages font partie du nom.
    (if (eq (packagecell a) (packagecell b))
        (alphalessp a b)
        (pkgcmp (packagecell a) (packagecell b))))
```

ANNEXE I

Les Appels Externes

Jérôme Chailloux
Matthieu Devin

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'callext))
```

Toutes ces fonctions sont "autoload" dans le système minimum.

```
(dmd defextern (nom ltype . type)
     (buildextern nom (getglobal nom) ltype (or (car type) 'fix)))

(de buildextern (nom adr ltype type)
     ; défini une procédure externe
     (lets ((n -1)
            (lvar (mapcar (lambda (l)
                            (symbol '#:system:callext
                                    (concat "arg" (incr n))))
                          ltype))
            (body `(callextern
                     (precompile ,(if (numberp adr) adr `',adr)
                         () () (eval (kwote (getglobal ',nom))) )
                     ,(convtypextern type)
                     ,@(mapcan (lambda (type var)
                                 (if (eq type 'external)
                                     `((vag ,var) ,(convtypextern type))
                                     `(,var ,(convtypextern type))))
                               ltype lvar))))
           (when (eq type 'external)
                 (setq body `(loc ,body)))
           (if (and (numberp adr) (zerop adr))
               (error 'defextern 'errudf nom)
               `(de ,nom ,lvar ,body))))

(de convtypextern (type)
    (selectq type
             (external 0)
             (fix 1)
             (float 2)
             (string 3)
             (vector 4)
```

```
            (rfix 5)                    ; FIX par reference (FORTRAN)
            (rfloat 6)                  ; FLOAT par reference (FORTRAN)
            ((t) 0)                     ; T arrête les clauses!!
            (t (error 'convtypextern 'erroob type))))
#+ (memq (system) '(vaxunix sps9 sm90 sun))
(progn
   (defextern _cload (string external external) external)
   (de cload (s)
      (#:system:ccode (_cload (string s) (#:system:ccode) (#:system:ecode)))))
#- (memq (system) '(vaxunix sps9 sm90 sun))
(de cload (string)
    (print "*** cload non implemente sur le systeme " (system)))
```

Les tests de lelisp.c :

```
#|     en enlever en cas de test
   (defextern _cchdir (string) fix)
   (defextern _chome () string)
   (defextern _cmoinsun () fix)
   (defextern _ctest (string float fix vector) float)
   (setq vect #[a b])
   (_ctest "FooBar" 123.45 123 vect)
|#
```

ANNEXE J

Le paragrapheur

Jérôme Chailloux

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'pretty))
```

Tous les symboles précédés de : seront créés dans le package PRETTY.

```
(defvar #:sys-package:colon 'pretty)
(add-feature 'pretty)
```

J.1 Les fonctions internes

```
(defvar :quotelevel 0)
(defvar :quotelength 0)
(de  :p (l)
     ; paragraphe l'expression ‹l›
     (cond
       ((getfn (type-of l) 'pretty ())
          ; essaie de lancer un paragrapheur de type étendu
          (funcall (getfn (type-of l) 'pretty ()) l))
       ((atom l) (prin l))
       ((and (eq (car l) 'QUOTE) (null (cddr l)))
          ; le cas de (QUOTE s)  =›  's
          (princn #/')
          (with ((printlevel :quotelevel)
                 (printlength :quotelength))
                (prin (cadr l))))
       ((and (consp (car l)) (eq (caar l) 'LAMBDA))
          ; le cas ((LAMBDA ...) ...)  =›  (LET ...)
          (:p (mcons 'let
                     (mapcar 'list (cadar l) (cdr l))
                     (cddar l))))
       ((and (if (symbolp (car l))  (‹› (ptype (car l)) 3) t)
             (:inlinep l)))
       (t (princn #/( )
          (let ((f (car l)) (l (cdr l)))
```

```
                (:p f)
                (selectq (if (symbolp f)
                             (ptype f)
                             t)
                     (1 (:progn))
                     (2 (:p1) (:progn))
                     (3 (:p1) (:p1) (:progn))
                     (4 (:cond))
                     (5 (:p1) (:cond))
                     (6 (with ((lmargin (+ (lmargin) 5)))
                              (while l (:p1) (:p1) (when l (terpri)))))
                     (7 (:p1) (:tagbody))
                     (t (:progn)))
                (and
                   l
                   (princn #\sp)
                   (princn #/. )
                   (princn #\sp)
                   (prin l)))
            (princn #/) ))))))

(de  :p1 ()
     ; édite l'élément suivant (sauf le 1er)
     (with ((lmargin (:lmargin f)))
           (princn #\sp)
           (:p (nextl l)))))))

(de  :progn ()
     ; paragraphe le PROGN courant
     ; ne traite que les listes bien formées
     (with ((lmargin (:lmargin f)))
           (while (consp l)
                  (if (numberp (car l))
                      (:p1)
                      (if (< (outpos) (lmargin))
                          (outpos (lmargin))
                          (terpri))
                      (:p (nextl l)))))))

(de  :tagbody ()
     ; paragraphe le TAGBODY courant
     ; ne traite que les listes bien formées!
     (with ((lmargin (:lmargin f)))
           (while (consp l)
                  (if (atom (car l))
                      (with ((lmargin (- (lmargin)
                                         (min (lmargin) 7))))
                            (terpri)
                            (:p1))
                      (if (< (outpos) (lmargin))
                          (outpos (lmargin))
                          (terpri))
                      (:p (nextl l)))))))

(de  :cond ()
```

```
      ; paragraphe le COND courant
      (with ((lmargin (+ (lmargin) 3)))
            (while (consp l)
                   (terpri)
                   (if (:inlinep (car l))
                       (nextl l)
                       (princn #/( )
                       (let ((l (nextl l)) (f t))
                            (:p (nextl l))
                            (when l (:progn)))
                       (princn #/) )))))))

(de   :lmargin (f)
      ; calcule la nouvelle marge gauche
      (+ (lmargin)
         (cond ((listp f) 1)
               ((vectorp f) 1)
               ((< (lmargin) (div :sizeline 2))
                   (if (< (plength f) 8)
                       (+ (plength f) 2)
                       4))
               ((< (lmargin) (scale :sizeline 3 4))
                   2)
               (t 0)))))))

(de   :inlinep (l)
      ; si <l> rentre dans la ligne imprime <l>
      ; sinon échappement de nom :inlinep et retourne ()
      (let ((:outpos (outpos)))
           (if (tag :inlinep
                    (let ((#:sys-package:itsoft 'pretty))
                         (prin l)
                         t))
               t
               (fillstring (outbuf) :outpos #\sp)
               (outpos :outpos)
               ())))

(de   #:pretty:eol ()
      ; le gestionnaire de fin de ligne de PRETTY
      (exit :inlinep ()))
```

J.2 Les fonctions utilisateur

```
(de   pprint (s)
      ; fonction utilisateur : paragraphe l'expression <s>
      (with ((lmargin (lmargin)))
            (let ((#:system:print-for-read t)      ; Impression pour la lecture
                  (#:system:print-package-flag     ; Packages a la demande
                           (or #:system:print-package-flag t))
                  (:sizeline (- (rmargin) (lmargin)))
                  (f ()))
```

```
                (:p s))
            (terpri))
      s)

(de  pprin (s)
    ; ne positionne rien
    (let ((:sizeline (- (rmargin) (lmargin))))
        (:p s)))

(df  pretty :l
    ; paragraphe la liste de fonctions <l>
    (mapc (lambda (l)
                (cond ((memq l #:trace:trace)
                       (print "; " l " est tracee")
                       (terpri)
                       (pprint (:get-plist-def l 'trace)))
                      ((getprop l 'resetfn)
                       (pprint (:get-plist-def l 'resetfn)))
                      (t (pprint (getdef l))))
                (terpri))
          :l))

(de :get-plist-def (symbol prop)
    (makedef symbol
             (car (getprop symbol prop))
             (cdr (getprop symbol prop))))

(df  prettyf :l
     ; comme PRETTY mais le résultat est sorti sur fichier
     ; le nom du fichier est donné en 1er argument
     (with ((outchan (openo (car :l))))
           (apply 'pretty (cdr :l))
           (close (outchan))
           (car :l)))

(de  prettyend ()
     (mapc 'remob (oblist '#.#:sys-package:colon))
     (rem-feature 'pretty)
     (libautoload pretty pprint pprin pretty prettyf prettyend)
     'prettyend)
```

J.3 Pose des ptypes standard

```
(let ((x '
(
 +       1
 -       1
 *       1
 append  1
 and     1
 calln   1
 catenate 1
 concat  1
```

```
list    1
max     1
mcons   1
min     1
or      1
plus    1
prin    1
print   1
prog1   1
progn   1
protect 1
times   1

any     2
block   2
call    2
catcherror      2
do      2
do*     2
every   2
evexit  2
evtag   2
exit    2
funcall 2
if      2
ifn     2
lambda  2
let     2
lets    2
let*    2
map     2
mapc    2
mapcar  2
maplist 2
mapcon  2
mapcan  2
repeat  2
slet    2
tag     2
unless  2
until   2
untilexit   2
when    2
while   2
with    2

de      3
defun   3
df      3
dm      3
dmc     3
dmd     3
dms     3
defmacro            3
```

```
 defsharp            3
 letn                3
 backtrack           3
 cond     4
 selectq  5
 setq     6
 setqq    6
 psetq    6
 tagbody  7
)))
(while x
   (ptype (nextl x) (nextl x)))))
```

ANNEXE K

Le Pisteur et le Mode Pas-à-pas

Jérôme Chailloux
Matthieu Devin

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'errief 'trace))
```

Tous les symboles précédés de : seront créés dans le package TRACE.

```
(defvar #:sys-package:colon 'trace)
(add-feature 'debug)
```

K.1 Les variables globales

```
(defvar :trace ())              ; contient la liste des fonctions tracées
```

K.2 Les fonctions auxiliaires

```
(de flat (l)
    (let ((r)) (flat-aux l) (reverse r)))

(de flat-aux (l)
    (cond ((null l) ())
          ((atom l) (newl r l))
          (t (flat-aux (car l)) (flat-aux (cdr l)))))
```

K.3 Les fonctions de tracage

```
(defvar tracewindow ())

(df trace l
    ; trace la liste de fonctions "l"
    (mapc ':trace-one l)
    (setq tracewindow
          (when (and :trace (featurep 'window) (null tracewindow))
                (create-window '#:window:tty
                   (div (bitxmax) 2) (mul (height-space) 3)
                   (div (bitxmax) 2) (bitymax) "Le_Lisp : Trace" 0 1)))
    l)

(df untrace l
    ; enlève la trace de toutes les fonctions
    ; de la liste 'l' ou de toutes les fonctions
    ; tracées si 'l' = ()
    (mapc ':untrace-one (or l (setq l :trace)))
    (when (and (null :trace) (featurep 'window) tracewindow)
          (kill-window tracewindow)
          (setq tracewindow ()))
    l)

(de :untrace-one (f)
    (let ((val (getprop f 'trace)))
         (if (atom val)
             (print f " n'etait pas tracee.")
             (when (getprop f 'resetfn)
                   (remprop f 'resetfn))
             (setfn f (car val) (cdr val))
             (remprop f 'trace)
             (setq :trace (delq f :trace)))))
```

Un spécification de trace (argument de trace-one) a le format suivant

```
  trace ::= fct
          | (fct [trace-spec]*)
          | ((fct*) [trace-spec]*)  ;  pas encore trace-spec ::= (wherein fct) | (wherein
(fct*)) ;  pas encore
          | (entry expr*)
          | (exit  expr*)
          | (when  expr)   ; trace conditionnelle
          | (break expr)   ; break conditionnel
          | (step expr)    ; pas à pas conditionnel
```

La spécif par defaut (correspondant à (trace foo)) est

```
  (foo (entry (prin 'foo "--->") (print-parameters 'foo))
       (exit  (prin 'foo "<---") (print #:trace:value))
       (when t)
       (break ())
       (step ())
  )
```

```
(defun default-specif (item)
    (selectq item
```

```
        (entry
              `((print ',:fct " ---> "
                       ,@(if (index "subr" :ftype)
                             ; pas de ` a cause des mapcan!
                             (mapcan (lambda (u) (list u " "))
                                     (flat :larg))
                             (mapcan (lambda (u) (list (kwote u) "=" u " "))
                                     (flat :larg))))))
        (exit
            `((print ',:fct " <--- " :value)))
        (when
             '(t))              ; pas de when (when t)
        (break
              ())               ; pas de break
        (step
             ())                ; pas de step
    ))
(defun parse-specif (specif)
    (mapcar (lambda (item) (or (assq item specif)
                               (cons item (default-specif item))))
            '(entry exit when break step)))
(defvar :not-in-trace-flag t)
(defvar :step-in-trace-flag ())
```

Les fonctions utilisées internement par trace

```
(synonymq :when when)
(synonymq :let let)
(synonymq :itsoft itsoft)
(defun build-tracing-fval (specif)
    `(,:larg
      (unstep
            (:when :not-in-trace-flag
             (:let ((:not-in-trace-flag ()))
              (setq :step-in-trace-flag ())
              (when ,(car (cassq 'when specif))
                    ,(if (featurep 'window)
                         `(with (,(list 'current-window 'tracewindow))
                                ,@(cassq 'entry specif))
                         `(progn ,@(cassq 'entry specif)))
                    (when ,(car (cassq 'break specif))
                          (let ((:not-in-trace-flag t))
                               (:itsoft 'syserror
                                '(,:fct break tracebreak)))))))
            (:let ((:value
                    (if (and ,(car (cassq 'when specif))
                             (or :step-in-trace-flag
                                 ,(car (cassq 'step specif))))
                        (step ,:call)
                        (cstep ,:call))))
             (:when :not-in-trace-flag
              (:let ((:not-in-trace-flag ()))
```

```
                (when ,(car (cassq 'when specif))
                      ,(if (featurep 'window)
                           `(with (,(list 'current-window
                                          'tracewindow))
                                  ,@(cassq 'exit specif))
                           `(progn
                                  ,@(cassq 'exit specif))))))
            :value))))

(defvar :fct)                   ; la fonction tracee
(defvar :ftype)                 ; son ftype
(defvar :larg)                  ; ses arguments, (pseudos pour subr)
(defvar :call)                  ; la forme pour lancer la fct

(de :trace-one (:fct)
    (let ((specif))
         (when (consp :fct)
               (setq specif (cdr :fct)
                     :fct (car :fct)))
         (when (memq :fct :trace) (funcall 'untrace :fct))
         ; liaison des spéciales
         (let (
               (:ftype (or (car (getprop :fct 'resetfn)) (typefn :fct)))
               :larg :call
               (fval (or (cdr (getprop :fct 'resetfn)) (valfn :fct)))
              )
              (remprop :fct 'resetfn)
              (selectq :ftype
                  ((expr fexpr macro dmacro)
                   (setq :larg  (car fval)
                         :call `(progn ,@(cdr fval))))
                  (subr0
                        (setq :larg  ()
                              :call `(call ',fval () () ())))
                  ((subr1)
                   (setq :larg '(:arg1)
                         :call `(call ',fval :arg1 () ())))
                  (subr2
                        (setq :larg '(:arg1 :arg2)
                              :call `(call ',fval :arg1 :arg2 ())))
                  (subr3
                        (setq :larg '(:arg1 :arg2 :arg3)
                              :call `(call ',fval :arg1 :arg2 :arg3)))
                  ((fsubr msubr dmsubr)
                   (setq :larg ':arg1
                         :call `(call ',fval :arg1 () ())))
                  (nsubr
                        (setq :larg ':arg1
                              :call `(calln ',fval :arg1))))
              (cond ((null :ftype)
                     (print "trace : je ne connais pas : " :fct))
                    ((not (memq :ftype '(expr fexpr macro dmacro
                                          msubr dmsubr subr0 subr1 subr2
```

```
                                        subr3 fsubr nsubr)))
                        (print "trace : je ne sais pas tracer "
                               :fct " (" :ftype ")"))
                       (t
                         (putprop :fct (cons :ftype fval) 'trace)
                         (newl :trace :fct)
                         (resetfn :fct
                             (or (car (memq :ftype '(expr fexpr macro dmacro)))
                                 (cassq :ftype
                                        '((subr1 . expr) (subr2 . expr)
                                          (subr3 . expr) (subr0 . expr)
                                          (fsubr . fexpr) (nsubr . expr)
                                          (msubr . macro) (dmsubr . dmacro))))
                             (build-tracing-fval (parse-specif specif)))))))
```

```
(de tracend ()
    (untrace)
    (mapc 'remob (oblist 'trace))
    (libautoload trace trace untrace)
    'tracend)
```

K.4 Le mode pas-à-pas

Le STEPPER : permet d'exécuter une expression en PAS A PAS
ex : (step (fib 10)). Le signal d'invite du pas à pas est "step>"

```
(setq #:sys-package:colon 'step)
```

TEMPORAIRE: le temps de pouvoir les declarer &DYNAMIC.

```
(defvar :depth 0)
(defvar :value ())
(defvar :speak t)
(defvar :history ())
(makunbound ':speak)

(df step (:exp)
  (let ((#:sys-package:itsoft
         (cons '#.#:sys-package:colon #:sys-package:itsoft))
        (:depth 0)                          ; la profondeur
        (:value)                            ; la valeur retournée
        (:speak t)                          ; la trace parle.
        (:history))                         ; l'histoire
       (tag step (traceval :exp))))

(de :stepeval (:forme :env)
  ; lancé par IT SOFT
  (let ((:depth (1+ :depth))
        (:history (cons :forme :history)))
```

```
       (cond ((and (consp :forme)
                   (eq (car :forme) 'unstep))
              (eval :forme :env))
             ((null :speak) (stepeval :forme :env))
             (t (:steploop #/.)))))

(df unstep :exp
    (eprogn :exp))

(df cstep (:exp)
    (if (and (boundp ':speak) :speak)
        (traceval :exp)
        (eval :exp)))

(de :stepmargin (:n)
    (repeat (if (< :n 20) :n 21) (princn #\sp)))

(de :steploop (:cmd)
    (selectq :cmd
        (#/. (:stepmargin :depth)
         (with ((printlevel 3) (printline 1))
               (prinflush :depth " -> " :forme " step>"))
         (teread)
         (:steploop (car (readline))))
        (#/=
         (itsoft 'syserror (list 'step 'break :forme))
         (:steploop #/.))
        (#/< (setq :value (let ((:speak ()))
                            (eval :forme :env)))
         (:stepmargin :depth)
         (print :depth " <- " :value))
        (#/q (exit #:system:toplevel-tag))
        (#/h (let ((:n 0)
                   (:history (reverse :history)))
               (while :history
                      (:stepmargin :n)
                      (print (incr :n) " " (nextl :history)))))
         (:steploop #/.))
        (#/? (print ";Les commandes de pas a pas sont : ")
         (print ";   CR  pour passer a l'expression suivante")
         (print ";   .   pour voir l'expression courante")
         (print ";   <   pour evaluer sans pas a pas et y revenir")
         (print ";   q   retour au toplevel")
         (print ";   h   pour avoir l'historique du pas a pas")
         (print ";   ?   pour avoir ce texte ....")
         (:steploop #/.))
        (t (setq :value (traceval :forme :env))
           (:stepmargin :depth)
           (print :depth " <- " :value))))

(de stepend ()
    (mapc 'remob (oblist 'step))
    (libautoload trace step unstep cstep)
    'stepend)
```

K.5 Récupération de l'espace

```
(de debugend ()
    ; récupère la place des fonctions de mise au point
    (tracend)
    (stepend)
    (rem-feature 'debug)
    'debugend)
```

ANNEXE L

Utilitaire permettant d'explorer la pile

Matthieu Devin

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
    (error 'load 'erricf 'debug))
(unless (featurep 'virtty)
        (initty))
```

Dans le package #:system:debug

```
(setq #:sys-package:colon '#:system:debug)
```

L.1 Interface avec le monde

Globales réglant le debug

lignes maxi de pretty print

```
(defvar #:system:debug-line 5)
```

niveaux maxi de pile

```
(defvar #:system:stack-depth 5)
```

Impression du contenu de la pile

```
(defun printstack nl   ;  ((&opt n 32767) (&opt l (cstack))) ...
    (let* ((l (if (and (consp nl) (consp (cdr nl))) (cadr nl) (cstack)))
           (m (if (consp nl)     ; nombre de blocs a imprimer
                  (min (car nl) (length l))
                  (length l)))
           (n (length l))        ; numéros des blocs
           frame)
          (mapc (lambda (f) (when (:is-hidden-block f) (decr n)))
                l)
          (setq m (min m n))
          (incr n)
          (while (gt m 0)
                 (nextl l frame)
                 (unless (:is-hidden-block frame)
                         (decr n)
```

```
                        (decr m)
                        (prin "   [stack ")
                        (when (< n 100) (princn #\sp))
                        (when (< n 10) (princn #\sp))
                        (prin n "]   (")
                        (selectq (car frame)
                            ; sur le type du bloc
                            (1  ; type lambda  (1 llink fval lparam v1 .. vn)
                                (prin (or (findfn (caddr frame))
                                          'let)))
                            (2  ; type label (2 fct1 ofval1 oftyp1 ...)
                                (prin 'flet " " (cadr frame)))
                            (3  ; type échappement (3 tag-name)
                                (prin 'tag " "(cadr frame)))
                            (4  ; type itsoft (4 llink nom etat forme funct)
                                (prin 'itsoft " " (caddr frame)))
                            (5  ; type lock (5 fval)
                                (prin 'lock " " (or (symbolp (cadr frame))
                                                    (findfn (cadr frame))
                                                    "(lambda ...)")))
                            (6  ; type protect (5 progn)
                                (prin 'protect))
                            (7  ; type sys-protect (6)
                                (prin 'sys-protect))
                            (8  ; type schedule (8 XXX)
                                (prin 'schedule))
                            (9  ; type tagbody (9 et1 corp1 ... etN corpN)
                                (prin 'tagbody))
                            (10 ; type bloc (10 slot)
                                (prin 'block))
                            (11)
                            (t  ; type erronne
                                (prin "*** bloc inconnu : " (car frame))))
                        (print " ...)")))))

(de :is-hidden-block (frame)
  (or (eq (car frame) 7)    ; sysprotect
      (eq (car frame) 11)   ; progn
      (eq (car frame) 4))   ; itsoft
)
```

Passage en/Sortie du mode debug

```
(df debug  #:system:l
    (cond ((atom #:system:l)
           ; retourne l'indicateur courant
           #:system:debug)
          ((memq (car #:system:l) '(t ()))
           ; positionnement global
           (setq #:system:debug (car #:system:l)))
          (t ; positionnement temporaire
           (let ((#:system:debug (car #:system:l)))
                (eval (car #:system:l))))))
```

Création d'une boucle d'inspection

```
(defun break ()
    (if #:system:debug
        (let ((#:system:debug t)
              (#:sys-package:itsoft
               (if (and (consp #:sys-package:itsoft)
                        (eq '#.#:sys-package:colon
                            (car #:sys-package:itsoft)))
                   #:sys-package:itsoft
                   (cons '#.#:sys-package:colon #:sys-package:itsoft))))
            (clockalarm 0)
            (tag continue
                 (with ((inchan ())
                        (outchan ()))
                       (:break-loop (cstack)))
                 (err)))
        (err)))
```

Lancement d'une commande du débuggeur

```
(defun debug-command (:char)
    (eprogn (cdr (cassq :char :commands))))
```

L.2 Implantation

Boucle d'inspection

```
(defvar :current-form)
(defvar :current-function)
(defvar :current-error-form)
(defvar :error-message)
(defvar :break-number 0)

(de :break-loop (:stack)
    (let (:error-message
          :current-function
          :current-form
          :current-error-form
          current-window
          (:break-number (add1 :break-number)))
         (protect
                (progn
                       (when (and (featurep 'window) (current-window))
                             (setq current-window (current-window))
                             (current-window
                                 (:create-break-window :break-number)))
                       (with ((prompt
                                      (if (gt :break-number 1)
                                          (catenate :break-number ">? ")
                                          ">? ")))
                             (:init-stack (:stack-in-error :stack))
                             (debug-command #/e)
                             (debug-command #/.)
```

```
                        (protect
                            (untilexit break
                                (itsoft 'toplevel ()))
                            (untilexit #:system:debug (:up-stack)))))
                (when (and (featurep 'window)
                           (current-window))
                      (kill-window (current-window))
                      (current-window current-window)))))
```

Interaction Le toplevel

```
(de :toplevel ()
    (tag #:system:debug
         (catcherror t
             (print "= " (eval (read))))))
```

Décodage des commandes

```
(defun :bol ()
  (super-itsoft '#.#:sys-package:colon 'bol ())
  (let* ((:inbuf (inbuf))
         (:c (sref (inbuf) 0)))
        (when (and (assq :c :commands) (eq 3 (inmax)))
              (sset :inbuf 0 #\cr) (sset :inbuf 1 #\lf)
              (debug-command :c))))
```

Initialisations Initialisation de la fenêtre

```
(de :create-break-window (n)
    (create-window '#:window:tty
           (div (bitxmax) 3)
           (mul (mul n 2) (height-space))
           (scale (bitxmax) 2 3)
           (scale (bitymax) 2 3)
           (catenate "Le_Lisp : Break Loop #" (string n))
           1
           1)))
```

Initialisation de la pile

```
(de :find-syserror (stack)
    (when stack
          (if (and (eq 4 (caar stack))
                   (eq 'syserror (caddar stack)))
              stack
              (:find-syserror (cdr stack)))))

(de :find-break (stack)
    (when stack
          (if (and (eq 1 (caar stack))
                   (eq (valfn 'break) (caddar stack)))
              stack
              (:find-break (cdr stack)))))

(de :cut-to-toplevel (stack)
  (if (:is-at-toplevel stack)
      (displace stack '(()))
      (let (frame)
```

```
          (until (or (null (cdr stack))
                     (:is-at-toplevel (cdr stack)))
                 (nextl stack))
          (when (consp stack) (rplacd stack ())))))

(de :is-at-toplevel (stack)
  (or (and (eq 4 (caar stack)) (eq 'toplevel (caddar stack)))
      (and
          (eq 7 (caar stack))   ; sysprot
          (nextl stack)
          (or (eq 7 (caar stack)) (eq 5 (caar stack)))
          (nextl stack)
          (and (eq 3 (caar stack)) (eq '#:system:error-tag (cadar stack)))
          (nextl stack)
          (if (eq 11 (caar stack)) (nextl stack) t)
          (and (eq 3 (caar stack)) (eq '#:system:toplevel-tag (cadar stack)))
          (nextl stack)
          (if (eq 1 (caar stack)) (nextl stack) t)
          (and (eq 4 (caar stack)) (eq 'toplevel (caddar stack))))))

(de :stack-in-error (stack)
    (:cut-to-toplevel stack)
    (let ((stack (or (:find-syserror stack)
                     (:find-break stack))))
         (cond ((and (eq 1 (caar stack))
                     (eq (valfn 'break) (caddar stack)))
                ;  break au sommet
                (setq :error-message '(break break ())
                      :current-error-form '(break)))
               ((eq 4 (caar stack))
                ; syserror au sommet
                (setq :error-message (nth 10 (car stack)))
                (setq :current-error-form ; break step: forme dans errmess
                      (if (eq 'step (car :error-message))
                          (caddr :error-message)
                          (nth 4 (car stack)))))
               (t (setq :error-message '(debug errerr ()))))
         (nextl stack)
         (:remove-fn
          '(cstep unstep step #:step:steploop #:step:stepeval)
          stack)))

(de :remove-fn (lfn stack)
  (setq stack (cons () stack)
        lfn (mapcar 'valfn lfn))
  (let ((bstack stack)
        (fstack stack))
       (while (setq fstack (:find-function (cdr fstack)))
              (ifn (:is-in-lvalfn (caddr (car fstack)) lfn)
                   (setq stack fstack)
                   (rplacd stack (cdr fstack))))
       (setq stack bstack)
       (cdr bstack)))

(de :is-in-lvalfn (valfn lvalfn)
```

```
    (any (lambda (vfn)
                 (or (eq vfn valfn)
                     (equal (loc valfn) vfn)))

         lvalfn))
(defun :init-stack (stack)
  (when (:is-a-struct-access (car stack)) ;   hack #:system:structaccess
        (nextl stack))
  (setq :current-form (ncons stack))
  (setq :current-function (ncons (:find-function stack))))
```

Les commandes

```
(defvar :commands
    '((#/v "show variables" (:print-current-variables))
      (#/h "print top of stack" (:printstack #:system:stack-depth))
      (#/H "print complete stack" (:printstack))
      (#/e "show error message" (:print-error))
      (#/. "show current stack frame" (:print-current-function))
      (#/+ "down stack" (:down-stack) (:print-current-function))
      (#/- "up stack" (:up-stack) (:print-current-function))
      (#/t "back to toplevel" (exit #:system:toplevel-tag))
      (#/q "exit inspection loop" (exit break))
      (#/r "resume and correct error" (:continue))
      (#/c "continue" (exit continue))
      (#/z "step traced functions"
       (setq #:trace:step-in-trace-flag t) (:continue))
      (#/? "list commands" (:help))))
```

Commandes d'impression

```
(de :printstack n
    (if n (setq n (car n)) (setq n (length (car :current-form))))
    (printstack n (car :current-form)))

(de :print-current-variables ()
    (:print-variables (car :current-form)))

(defun :print-variables (:s)
    (cond ((:has-function-definition (car :s))
           (:print-arguments (nth 3 (car :s))))
          ((eq 1 (caar :s))
           (:print-arguments (nth 3 (car :s)))
           (:bind/unbind (car :s))
           (:print-variables (cdr :s))
           (:bind/unbind (car :s)))
          (:s (:print-variables (cdr :s)))))

(defun :print-arguments (:larg)
    (cond ((or (eq :larg '&nobind) (null :larg)))
          ((symbolp :larg) (print " " :larg "=" (symeval :larg)))
          ((consp :larg)
           (:print-arguments (car :larg))
           (:print-arguments (cdr :larg)))))

(de :print-error ()
    (apply 'printerror :error-message)
```

```
    (selectq (cadr :error-message)
        ((errwna  ;   "mauvais nombre d'arguments"
          errbpa  ;   "mauvais parametre"
          errilb) ;   "liaison illegale"
         (let ((d (:getdef (car :error-message))))
              (:print-filtered
                  (list (car d) (cadr d) (:hilited:make (caddr d))))))
    ))
(defun :print-current-function ()
    (let ((current-definition
              (:getdef (findfn (caddr (caar :current-function))))))
         (when current-definition
               (:print-filtered
                   (:hilite-expr
                       (:current-expr current-definition)
                       current-definition)))))
```

Commandes de déplacement dans la pile

```
(defun :down-stack ()
  (let* ((:next-form (cdr (car :current-function)))
         (:next-function (:find-function :next-form)))
        (ifn :next-function
             (exit #:system:debug)
             (newl :current-form :next-form)
             (newl :current-function :next-function)
             (let ((:previous-form (cadr :current-form)))
                  (while (neq :previous-form (car :current-form))
                         (:bind/unbind (nextl :previous-form)))))))
(de :up-stack ()
    (ifn (cdr :current-function)
         (exit #:system:debug)
         (let ((:rebind-frames
                   (:up-frame-list (cadr :current-form) (car :current-form))))
              (nextl :current-function)
              (nextl :current-form)
              (while :rebind-frames
                     (:bind/unbind (nextl :rebind-frames))))))
(de :up-frame-list (f1 f2)
    (let ((res ()))
         (until (eq (car f1) (car f2))
                (newl res (nextl f1)))
         res))
```

Commandes diverses

```
(de :continue ()
    (selectq (cadr :error-message)
        (break (exit continue))
        (errudv
               (prinflush (prompt) "(setq " (caddr :error-message) " '")
               (with ((prompt ""))
                     (let ((form
                                `(setq ,(caddr :error-message) ',(read))))
```

```
                        (exit continue (eval form)))))
        (errudf
              (prinflush "Function ")
              (exit continue (read)))
        (t
          (print "This error can not be resumed"))))
(defun :help ()
    (mapc (lambda ((c doc . rest)) (print "; " (ascii c) ":     " doc))
          :commands))
```

Dépilage d'un block de pile

```
(defun :bind/unbind (:frame)
    (when (eq 1 (car :frame))    ; bloc lambda
          ; type lambda  (1 llink fval lparam vn ... v1)
          (let ((:lval (nreverse (cddddr :frame))))
               (:exchange-arguments (cadddr :frame) :lval)
               (rplacd (cdddr :frame) (nreverse :lval)))))

(defun :exchange-arguments (:larg :lval)
    (cond ((or (eq :larg '&nobind) (null :larg)))
          ((symbolp :larg)
           (rplaca :lval
                   (prog1 (:cval :larg)
                          (:scval :larg (car :lval)))))
          ((consp :larg)
           (:exchange-arguments (car :larg)
               (if (consp (car :larg)) (car :lval) :lval))
           (:exchange-arguments (cdr :larg) (cdr :lval)))))

(defun :scval (:s :v)
    (if (boundp ':v) (set :s :v) (makunbound :s)))

(defun :cval (:s)
    (if (boundp :s) (symeval :s) '_undef_))

(de :current-expr (definition)
    (let ((:frame-list
              (:up-frame-list (car :current-form) (car :current-function))))
         (while :frame-list
                (setq definition (:find-expr (nextl :frame-list) definition)))
         (when (null (cdr :current-form))
               (setq definition (:find-error definition)))
         definition))
```

Trouver l'expression ayant créé le bloc frame dans sexpr

```
(de :find-expr (frame sexpr)
    (or
       (:find-tree
        (or
           (selectq (car frame)
               ; sur le type du bloc
               (1  ; type lambda  (1 llink fval lparam v1 ... vn)
                   (lambda (expr)
                           (and (consp (car expr))
                                (eq (cdar expr) (caddr frame))))))
```

```
                ; type label (2 fct1 ofval1 oftyp1 ... oftypn)
                (3  ; type échappement (3 tag-name)
                    (lambda (expr)
                            (and (memq (car expr) '(tag untilexit))
                                 (consp (cdr expr))
                                 (eq (cadr expr) (cadr frame)))))
                ; type itsoft (4 llink nom etat forme funct)
                (5  ; type lock (5 fval)
                    (lambda (expr)
                            (and (eq (car expr) 'lock)
                                 (consp (cdr expr))
                                 (eq (cadr expr) (cadr frame)))))
                (6  ; type protect (6 progn)
                    (lambda (expr)
                            (and (eq (car expr) 'protect)
                                 (consp (cdr expr))
                                 (eq (cddr expr) (cadr frame)))))
                ; type sys-protect (7)
                (8   ; type schedule (8 XXX)
                     (lambda (expr) (eq (car expr) 'schedule)))

                (9   ; type tagbody (9 et1 corp1 ... etN corpN)
                )
                ; type bloc   (10 slot)
                (11  ; type progn (11 progn)
                     (lambda (expr) (exit found (caadr frame)))))
            (lambda (expr) (exit found ())))
         sexpr)
        sexpr))
```

Trouver l'expression qui a provoqué l'erreur dans expr

```
(de :find-error (expr)
    (or
       (when (memq (cadr :error-message) '(errudf errbal errwna errilb break))
             (:find-tree
                 (lambda (expr)
                         (eq expr :current-error-form))
                 expr))
       (when (eq (cadr :error-message) 'errudv)
             (:find-tree
                 (lambda (expr)
                         (when (eq (car expr) (caddr :error-message))
                               (exit found (car expr))))
                 expr))
       (:find-tree
           (lambda (expr)
                   (eq (car expr) (car :error-message)))
           expr)
       expr))
```

L.2.1 Utilitaires

Recherche et substitution dans un arbre

```
(de :find-tree (:fn :tree)
    (tag found (:find-tree1 :fn :tree)))

(de :find-tree1 (:fn :tree)
  (when (consp :tree)
        (when (funcall :fn :tree)
           (exit found :tree))
        (while (consp :tree)
               (:find-tree1 :fn (nextl :tree)))))
```

Substitution avec test EQ

```
(defun substq (n o s)
    (cond ((atom s)
           (if (eq s o) n s))
          ((consp s)
           (cons
                (if (eq (car s) o)
                    n
                    (substq n o (car s)))
                (if (eq (cdr s) o)
                    n
                    (substq n o (cdr s)))))))
```

Manipuler la pile sous forme de CSTACK

```
(de :find-function (stack)
    (until (or (:has-function-definition (car stack))
               (null stack))
           (nextl stack))
    stack)

(defun :has-function-definition (frame)
    (and (eq (car frame) 1)
         (findfn (caddr frame))))

(de :is-a-struct-access (frame)
    (and (eq (car frame) 1)
         (let ((valfn (caddr frame)))
              (and (consp valfn)
                   (eq (car valfn) '&nobind)
                   (consp (cdr valfn))
                   (consp (cadr valfn))
                   (eq (caadr valfn) '#:system:structaccess)))))
```

Récupérer la définition d'une fonction

```
(de :getdef (f)
    (cond ((memq f #:trace:trace)
           (:get-plist-def f 'trace))
          ((getprop f 'resetfn)
           (:get-plist-def f 'resetfn))
          (t (getdef f))))
```

```
(de :get-plist-def (symbol prop)
    (makedef symbol
        (car (getprop symbol prop))
        (cdr (getprop symbol prop))))
```

L.2.2 Gestion des objets mis en valeur

```
(de :hilite-expr (expr1 expr2)
    (ifn expr1
         expr2
         (substq (:hilited:make expr1) expr1 expr2)))

(defun :hilited:make (o)
    (tcons ':hilited o))

(defun :hilited:prin (f)
    (with ((tyattrib t))
          (pprin (cdr f))))

(synonymq :hilited:pretty :hilited:prin)
```

Limitation de l'impression hilitée

```
(defvar :outlist)
(defvar :hibegin)
(defvar :hiend)
(defvar :nlines)

(defvar #:tty:system:hilited:tyattrib ())
(de #:tty:system:hilited:tyattrib (x)
    (if x
        (setq :hibegin (cons :nlines (outpos)))
        (setq :hiend (cons :nlines (outpos)))))

(de :hilited:eol ()
    (incr :nlines)
    (newl :outlist (substring (outbuf) 0 (outpos)))
    (fillstring (outbuf) 0 #\sp)
    (outpos (lmargin)))

(de :print-filtered (x)
    (let ((:nlines 0)
          (:hibegin ())
          (:hiend ())
          (:outlist ()))
         (let ((#:sys-package:itsoft ':hilited)
               (#:sys-package:tty '#:tty:system:hilited))
              (pprint x))
         (ifn (and :hibegin :hiend)
              (setq :hibegin (cons 0 0)
                    :hiend (cons 0 0)))
         (:flush-hilited)))

(de :flush-hilited ()
  (setq :outlist (nreverse :outlist))
  (unless (le :nlines #:system:debug-line)
          (let ((nscroll (min (sub1 (car :hibegin))
                              (sub (add1 (car :hiend))
```

```
                                #:system:debug-line))))
            (when (gt nscroll 0)
                  (rplaca :hibegin (sub (car :hibegin) nscroll))
                  (rplaca :hiend (min (sub (car :hiend) nscroll)
                                      (sub1 #:system:debug-line)))
                  (rplacd :outlist
                          (nthcdr (add1 nscroll) :outlist))
                  (rplaca :outlist (catenate (car :outlist) " ...")))
            (when (gt (length :outlist) #:system:debug-line)
                  (rplacd (nthcdr (sub1 #:system:debug-line) :outlist)
                          ())
                  (rplaca (last :outlist)
                          (catenate (car (last :outlist)) "   ...")))))
(let ((#:system:print-for-read ()))
     (with ((outchan t))
           (with ((rmargin 78))
                 (repeat (car :hibegin) (print (nextl :outlist)))
                 (prin (substring (car :outlist) 0 (cdr :hibegin)))
                 (tyattrib t)
                 (cond ((eq (car :hiend) (car :hibegin))
                        (prin (substring (car :outlist)
                                         (cdr :hibegin)
                                         (sub (cdr :hiend)
                                              (cdr :hibegin)))))
                       (t
                         (print (substring (nextl :outlist) (cdr :hibegin)))
                         (repeat (sub1 (sub (car :hiend) (car :hibegin)))
                                 (tyattrib t) (print (nextl :outlist)))
                         (tyattrib t)
                         (prin (substring (car :outlist) 0 (cdr :hiend)))))
                 (tyattrib ())
                 (print (substring (nextl :outlist) (cdr :hiend)))
                 (while :outlist (print (nextl :outlist)))))))
```

ANNEXE M

L'éditeur vidéo minimum PEPE

Jérôme Chailloux

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'pepe))
```

Tous les symboles précédés de : seront créés dans le package PEPE.

```
(defvar #:sys-package:colon 'pepe)
(add-feature 'pepe)
```

PEPE est l'éditeur minimum de Le_Lisp version 15.2. Il permet d'éditer des fichiers ou n'importe quelle expression Lisp. Il fonctionne sur tout terminal vidéo qui doit au moins permettre l'effacement de tout l'écran et le positionnement absolu du curseur.

Le tampon de PEPE est une liste de chaînes de caractères, chacunes d'elles représentant une ligne. Cette représentation qui a le mérite de la simplicité entrainera la fabrication d'une nouvelle chaîne pour chaque caractère tapé.

Une optimisation facile à réaliser consiste à remplacer tous les appels des fonctions arithmétiques génériques (+, -, >, incr, decr ...) par des appels aux fonctions spécialisées (add, sub, ge ...) et d'utiliser les fonctions de base sur les chaînes de caractères SLEN, SSET et SREF.

PEPE, tout comme EMACS, garde les propriétés suivantes :
- écrit entièrement en Lisp
- extensible très facilement
- indépendant vis-à-vis du matériel
- reaffichage asynchrone.

M.1 Les Variables Globales

```
(defvar :buffer ())          ; le tampon des lignes
(defvar :xcursor 0)          ; X courant
(defvar :ycursor 0)          ; Y courant
(defvar :column 0)           ; colonne courante
(defvar :ydisplay 0)         ; numéro de la 1ère ligne visible
(defvar :file "tmp")         ; nom du fichier courant
(defvar :commands ())        ; liste des commandes
(defvar :escommands ())      ; liste des ESC-commandes
(defvar :modbuf ())          ; indicateur de tampon modifié
(defvar :kill ())            ; la chaîne du dernier kill
(defvar :searchstrg "")      ; la chaîne de la dernière recherche
```

M.2 La Boucle Principale

```
(df pepe (:f)
    ; la forme FSUBRée de la fonction suivante
    (pepefile :f))

(de pepefile (:f)
    ; la fonction qui évalue son nom de fichier
    (let ((:xmax          (tyxmax))
          (:ymax          (tyymax))
          (:xmax+1        (1+ (tyxmax)))
          (#:tty:-xmax+1 (- (1+ (tyxmax))))
          (:xmax-1        (1- (tyxmax)))
          (:ymax+1        (1+ (tyymax)))
          (:ymax-1        (1- (tyymax)))
          (:xmax+1*ymax   (* (1+ (tyxmax)) (tyymax))))
       (let ((oscreen (makestring (* :xmax+1 :ymax+1) #\sp))
             (nscreen (makestring (* :xmax+1 :ymax+1) #\sp)))
            (:clrscreen)
            (ifn :f
                 (unless :buffer (setq :buffer (list "")))
                 (setq :file "tmp"
                       :xcursor 0 :ycursor 0
                       :column 0 :ydisplay 0
                       :modbuf ())
                 (cond
                     ((equal :f t)
                       ; je veux un fichier scratch
                       (setq :buffer (list "")))
                     ((atom :f)
                       ; ce doit être un fichier qui existe
                       (tag eoc (setq :buffer (:readfile :f)))
                       (setq :file (string :f)))
                     (t ; c'est donc un PROGN à évaluer
                       (setq :buffer (list ""))
                       (let ((#:sys-package:itsoft ':eval))
                            (eval :f))
```

```
                    (setq :buffer (nreverse :buffer)))))
            (typrologue)
            ;
            ; le top-level proprement dit de PEPE
            ;
            (untilexit pepe
               (tag eoc
                    (:redisplay)
                    (tycursor :xcursor :ycursor)
                    (:pepecmd (tyi))))
            ; je sors de PEPE : qu'Il soit avec vous!
            (tycursor 0 :ymax)
            (tycleol)
            (tycursor 0 :ymax-1)
            (tycleol)
            (tyepilogue)
            (tyflush)
            'pepe))))))

(de :eval:eol ()
    ; récupération de la ligne imprimée
    ; s'il faut éditer le résulat d'une évaluation
    (newl :buffer (substring (outbuf) 0 (outpos)))
    (fillstring (outbuf) 0 #\sp (outpos))
    (outpos (lmargin)))
```

M.3 Les interprètes des commandes

```
(de :pepecmd (c)
    ; interprète la commande ‹c›
    ; la A-liste des commandes se trouve dans :commands
    (let ((l (cassq c :commands)))
         (if l (eprogn l)
             ; ce n'est pas une commande
             (if (< c 32)
                 (:deadend)
                 (:insertchar c))))
    (when (setq c (tys))
          (:pepecmd c)))

(de :escommand ()
    ; L'interprète des commandes ‹esc› X
    ; la A-Liste des commandes se trouve dans :escommands
    (let ((c (tyi)))
         ; passage en majuscule
         (when (and (>= c #/a) (<= c #/z))
               (decr c 32))
         (let ((l (cassq c :escommands)))
              (if l (eprogn l) (:deadend)))))
```

M.4 Les fonctions d'affichage

A de rares exceptions, ces fonctions n'utilisent pas les fonctions du terminal virtuel mais la fonction de reaffichage asynchrone REDISPLAYSCREEN.

```
(de  :redisplay ()
     ; reaffiche toute la fenêtre visible
     (fillstring nscreen 0 #\SP)
     (let ((y #:tty:-xmax+1)
           (s (length :buffer))
           (l (nthcdr :ydisplay :buffer)))
          (repeat :ymax
                  (bltstring nscreen (incr y :xmax+1)
                             (car l) 0 :xmax+1)
                  (when (> (slength (nextl l)) :xmax+1)
                        (chrset (+ y :xmax) nscreen #/)))
          (:fillminibuf 0
                 "Pepe :                          lignes  <M>")
          (:fillminibuf 7 :file)
          (:fillminibufnb (+ :xmax+1*ymax 28) s)
          (when (<= s 1)
                (chrset (+ :xmax+1*ymax 35) nscreen #\sp ))
          (unless :modbuf (:fillminibuf 38 "    "))
          (:trueredisplay)))

(dmd :trueredisplay ()
     `(redisplayscreen nscreen oscreen :xmax+1 :ymax+1))

(dmd :fillminibuf (x strg)
     ; écrit dans la dernière ligne à partir de <x> la chaîne <strg>
     `(bltstring nscreen
         (+ :xmax+1*ymax ,x) ,strg 0))

(de  :fillminibufnb (pos nb)
     ; écrit dans nscreen à partir de <x> la valeur numérique <nb>
     (if (< nb 10)
         (chrset pos nscreen (+ #/0 nb))
         (:fillminibufnb (1- pos) (div nb 10))
         (chrset pos nscreen (+ #/0 (rem nb 10)))))

(de :clrscreen ()
    ; efface tout l'écran
    (tycursor 0 0)
    (tycls)
    (fillstring oscreen 0 #\sp))
```

M.5 Les fonctions de dialogue

```
(de :more ()
    ; demande s'il faut continuer les impressions.
    (:fillminibuf 42 "Encore ? ")
    (:trueredisplay)
    (tycursor 50 :ymax)
    (unless (chrpos (tyi) " YyOoTt") (exit eoc))
    (:fillminibuf 42 "          "))

(de :readname (strg)
    ; Lecture d'une chaîne sur le terminal
    ; ‹strg› est la chaîne d'invite
    (:fillminibuf 42 strg)
    (let ((l) (c) (p (+ 42 (slength strg))))
         (:trueredisplay)
         (tycursor p :ymax)
         (while (neq (setq c (tyi)) #^M)
                (cond ((eq c #^G) (exit eoc ()))
                      ((eq c #\bs)
                       (:fillminibuf (decr p) " ") (decr p) (nextl l))
                      (t (:fillminibuf p (list c)) (newl l c)))
                (:trueredisplay)
                (tycursor (incr p) :ymax))
         (if l (string (nreverse l)) (exit eoc ""))))
```

M.6 Les Fonctions Auxiliaires

```
(dmd :deadend ()
     ; fin de la route : on ne plus plus bouger!
     `(progn (tybeep) (exit eoc)))

(dmd :currentline ()
     ; retourne la ligne courante
     `(nth (+ :ydisplay :ycursor) :buffer))

(dmd :currentlines ()
     ; retourne la liste commencant par la ligne courante
     `(nthcdr (+ :ydisplay :ycursor) :buffer))

(dmd :cursor (x y)
     ; change la position du curseur de PEPE
     `(setq :xcursor ,x :ycursor ,y))
```

M.7 Une Brouette de Prédicats Utiles

```
(dmd :bolp ()
     ; teste si on se trouve en début de ligne
     `(= :xcursor 0)))

(dmd :eolp ()
     ; teste si on se trouve en fin de ligne
     `(>= :xcursor (slength (:currentline))))

(dmd :bobp ()
     ; teste si on se trouve en début de buffer
     `(and (= :ydisplay 0) (= :ycursor 0)))

(dmd :eobp ()
     ; teste si on se trouve en fin de buffer
     `(>= (+ :ydisplay :ycursor 1) (length :buffer)))

(dmd :bosp ()
     ; teste si on se trouve au début de l'écran
     `(= :ycursor 0))

(dmd :eosp ()
     ; teste si on se trouve à la fin de l'écran
     `(>= :ycursor :ymax-1))
```

M.8 Les Commandes de Base de PEPE

```
(de :left ()
    ; un coup à gauche
    (ifn (:bolp)
         (setq :column (decr :xcursor))
         (:up) (:endline)))

(de :right ()
    ; un coup à droite
    (ifn (:eolp)
         (if (>= :xcursor :xmax)
             (:deadend)
             (setq :column (incr :xcursor)))
         (:begline)
         (:down)))

(de :endline ()
    ; va à la fin de la ligne
    (setq :xcursor (min (slength (:currentline)) :xmax)
          :column :xcursor))

(de :begline ()
    ; va au début de la ligne
    (setq :xcursor 0 :column 0)))

(de :up ()
    ; va à la ligne précédente
    (if (:bobp)
        (:deadend)
```

```
        (ifn (:bosp)
             (setq :ycursor (1- :ycursor)
                   :xcursor (min :column
                                 (slength (:currentline))))
             (decr :ydisplay 6)
             (incr :ycursor 5)
             (when (< :ydisplay 0)
                   (setq :ydisplay 0 :ycursor 0)))))

(de :down ()
    ; va à la ligne suivante
    (if (:eobp)
        (:deadend)
        (ifn (:eosp)
             (setq :ycursor (1+ :ycursor)
                   :xcursor (min :column
                                 (slength (:currentline))))
             (incr :ydisplay 6)
             (decr :ycursor 5))))

(de :nextscreen ()
    ; passe à l'écran suivant
    (when (> (+ :ydisplay :ymax-1) (length :buffer))
          (:deadend))
    (incr :ydisplay :ymax-1)
    (:begline))

(de :prevscreen ()
    ; passe à l'écran précédent
    (when (< (decr :ydisplay  :ymax-1) 0)
          (setq :ydisplay 0))
    (:begline))

(de :insertchar (c)
    ; rajoute le caractère <c> à la position courante
    ; règle le pb du bltstring à droite sur la même chaîne.
    (slet ((l (:currentlines))
           (s (catenate " "  (car l))))
          (if (>= :xcursor :xmax)
              (:deadend)
              (bltstring s 0 s 1 :xcursor)
              (chrset :xcursor s c)
              (rplaca l s))
          (setq :column (incr :xcursor))
          (setq :modbuf t)))

(de :deletechar ()
    ; enlève le caractère à la position du curseur
    (let ((l (:currentlines))
          (s))
         (if (:eolp)
             (rplac l (catenate (car l) (cadr l)) (cddr l))
             (setq s (makestring (1- (slength (car l))) #\sp))
             (bltstring s 0 (car l) 0 :xcursor)
             (bltstring s :xcursor (car l) (1+ :xcursor))
             (rplaca l s)))
```

```
    (setq :modbuf t))
(de :breakline ()
    ; casse la ligne à la position du curseur
    (let ((l (:currentlines)))
         (ifn (:eolp)
              (rplac l (substring (car l) 0 :xcursor)
                       (cons (substring (car l) :xcursor) (cdr l)))
              (rplacd l (cons "" (cdr l))))
         (setq :modbuf t)))
(de :killine ()
    ; détruit la ligne courante
    (if (:bobp)
        (if (cdr :buffer)
            (setq :kill (nextl :buffer))
            (setq :buffer (list "")))
        (let ((l (:currentlines)))
             (setq :kill (car l))
             (rplac l (cadr l) (cddr l))))
    (setq :modbuf t)
    (if (:eobp)
        (or (:bobp) (:up))
        ; et l'autre cas si la ligne est plus petite
        (when (< (slength (:currentline)) :xcursor)
              (:endline))))
```

M.9 Les Fonctions de Recherche

```
(de  :search ()
     ; cherche une chaîne de caractères
     (let ((s (tag eoc (:readname "chaine: ")))
           (f (:currentlines))
           (ind :xcursor))
          (cond ((null s) (exit eoc))
                ((eqstring s "") (setq s :searchstrg))
                (t (setq :searchstrg s)))
          (untilexit search
             (let ((r (index s (nextl f) ind)))
                  (cond
                   (r (setq :xcursor (+ r (slength s)))
                      (let ((l1 (length (:currentlines)))
                            (l2 (1+ (length f))))
                           (if (> (+ :ycursor (- l1 l2)) :ymax-1)
                               (setq :ydisplay (- (length :buffer) l2 2)
                                     :ycursor 2)
                               (setq :ycursor (+ :ycursor (- l1 l2)))))
                      (exit search))
                   ((null f) (exit search (:deadend)))
                   (t (setq ind 0)))))))
(de :matchparent (x)
```

```
    ; Vérificateur de parenthèses à la Lisp
    ; x peut être une parenthèse ou un crochet.
    (tag fin (until (= (:curlexnext) x))))

(de :curlexnext ()
    (let ((char (:curchar)))
         (:right)
         (selectq char
                 (#/( (until (= (:curlexnext) #/))))
                 (#/[ (until (= (:curlexnext) #/])))
                 (#/" (until (= (:curchar) #/") (:right))
                      (:right))
                 (#/# (selectq (:curchar)
                         ((#/( #/[ #/") (:curlexnext))
                         (#// (:right))
                         (t ())))
                 (#/; (:left) (:begline) (:down))
                 (t ()))
         char))

(de :curchar ()
    (let ((line (:currentline)))
         (if (or (<= (slength line) :xcursor)
                 (eqstring line ""))
             0
             (chrnth :xcursor line))))
```

M.10 Les Fonctions sur les Fichiers

```
(de :readfile (f)
    ; Lecture d'un fichier : retourne une liste (non vide)
    ; de lignes d'au plus :xmax caractères.
     (let (ll (in (probefile f)))
          (if in
              ; c'est un fichier connu
              (with ((inchan (openi f)))
                    (inmax 0)
                    (untilexit eof (newl ll (readstring)))
                    (if (consp ll) (nreverse ll) (list "")))
              ; c'est un nouveau fichier
              (tybeep)          ;  pour indiquer une création!
              (list ""))))

(de :writefile (f)
    ; Ecriture d'un fichier
     (let ((out (catcherror () (openo f))))
       (unless (consp out) (:deadend))
       (with ((outchan (car out)))
             (let ((:xcursor :xcursor)
                   (:ycursor :ycursor)
                   (#:system:print-for-read ()))
                  (:cursor 33 :ymax)
```

```
                (with ((rmargin :xmax+1))
                      (mapc 'print :buffer)))
            (close (outchan)))     ; Redde Caesari quae sunt Caesaris !
       (setq :modbuf ())))

(de  :insertfile ()
     ; Insert un fichier à la position du curseur.
     (let ((l (:currentlines)))
          (rplacd l (nconc (:readfile (:readname "fichier? "))
                           (cdr l)))
          (setq :modbuf t)))

(de :backup ()
    ; change le nom de :file en :file.BAK
    ; à utiliser avant un :write
    (let ((i (index "." :file)) (:backup))
         (setq :backup
               (catenate (if i (substring :file 0 i) :file) ".BAK"))
         (renamefile :file :backup)))

(de  :help ()
     ; affiche un aide mémoire des commandes.
     (let ((:ycursor :ycursor)
           (y #:tty:-xmax+1)
           (in))
          (unless (catcherror ()
                     (setq in (openi (catenate #:system:llib-directory
                                               'pepehelp
                                               #:system:lelisp-extension))))
                  (:deadend))
          (:clrscreen)
          (with ((inchan in))
                (fillstring nscreen 0 #\SP)
                (untilexit eof
                  (bltstring nscreen (incr y :xmax+1)
                             (readstring) 0 :xmax+1)))
          (:fillminibuf 0 "Aide memoire de PEPE"))
     ; attend la frappe d'un caractère pour continuer
     (tag eoc (:more))
     (:clrscreen))
```

M.11 Les Autres Fonctions

```
(de :evalbuffer ()
    ; Evaluation de tout le tampon sans sortir de PEPE
    ; indépendant du format des lignes.
    ; Toutes les impressions vont sur le terminal très proprement
    (let ((#:sys-package:itsoft ':evalbuffer)
          (:buffer :buffer) ; sauvetage du tampon
          (xy #:tty:-xmax+1))
         (with ((lmargin 0)
                (rmargin :xmax-1))
```

```
                (tycursor 0 0)
                (tyflush)
                (untilexit eoc (print "=> " (eval (read))))
                (:more))))

(de   :evalbuffer:bol ()
      ; nouvelle ligne en entrée sous EVALBUFFER
      (ifn :buffer
           (exit eoc)
           (slet ((l (nextl :buffer))
                  (n (slength l)))
                 (bltstring (inbuf) 0 l)
                 (chrset n (inbuf) #\cr)
                 (chrset (incr n) (inbuf) #\lf)
                 (inmax (incr n)))))

(de   :evalbuffer:syserror (f m b)
      (printerror f m b)
      (exit eoc))

(de   :evalbuffer:eol ()
      ; ITSOFT fin de ligne sous EVALBUFFER
      (fillstring nscreen (incr xy :xmax+1) #\sp :xmax+1)
      (bltstring nscreen xy (outbuf) 0 (outpos))
      (fillstring (outbuf) 0 #\sp (outpos))
      (outpos (lmargin))
      (when (>= xy (* :xmax+1 :ymax-1))
            (:more)
            (setq xy #:tty:-xmax+1)))
```

M.12 Initialisation des Clefs

```
(dmd defkey (k . f)  `(newl :commands (cons ,k ',f)))

(progn
        (defkey  #^A          (:begline))
        (defkey  #:tty:left   (:left))
        (defkey  #^C          (exit pepe))
        (defkey  #^D          (:deletechar))
        (defkey  #^E          (:endline))
        (defkey  #:tty:right  (:right))
        (defkey  #^G          (:deadend))
        (defkey  #^H          (:left)
                              (:cursor :xcursor :ycursor)
                              (:deletechar))
        (defkey  #^K          (:killine))
        (defkey  #^L          (:clrscreen))
        (defkey  #^M          (:breakline) (:right))
        (defkey  #:tty:down   (:down))
        (defkey  #^O          (:breakline))
        (defkey  #:tty:up     (:up))
        (defkey  #^S          (:search))
        (defkey  #^V          (:clrscreen) (:nextscreen))
```

```
        (defkey  #^Y          (:breakline)
                              (:cursor :xcursor :ycursor)
                              (mapc ':insertchar
                                    (pname :kill)))
        (defkey  #$7F         (:left)
                              (:cursor :xcursor :ycursor)
                              (:deletechar))
        (defkey  27           (:escommand))
)
(df defesckey (k . f)     (newl :escommands (cons k f)))
(progn
        (defesckey  #/E   (:evalbuffer))
        (defesckey  #/F   (setq :file
                                (:readname "Nom de fichier? ")))
        (defesckey  #/I   (:insertfile))
        (defesckey  #/R   (setq :file (:readname "Fichier? ")
                                :buffer (:readfile :file)
                                :xcursor 0 :ycursor 0
                                :column 0 :ydisplay 0
                                :modbuf ()))
        (defesckey  #/S   (:backup) (:writefile :file))
        (defesckey  #/V   (:clrscreen) (:prevscreen))
        (defesckey  #/W   (:writefile (:readname "Fichier? ")))
        (defesckey  #/X   (apply (or (getfn '#.#:sys-package:colon
                                            (concat (:readname
                                                      "Fonction? ")))
                                     ':deadend)
                                 ()))
        (defesckey  #/Z   (:writefile :file)
                          (tycursor 0 :ymax)
                          (loadfile :file t)
                          (exit pepe))
        (defesckey  #/)   (:matchparent #/())
        (defesckey  #/]   (:matchparent #/[))
        (defesckey  #/<   (:clrscreen)
                          (setq :ydisplay 0 :xcursor 0 :ycursor 0 :column 0))
        (defesckey  #/>   (:clrscreen)
                          (setq :ycursor (length :buffer)
                                :ydisplay (max 0 (- :ycursor
                                                    (div :ymax 2)))
                                :ycursor (- :ycursor :ydisplay 1))
                          (:endline))
        (defesckey  #/?   (:help))
))
```

M.13 Pour Récupérer la Place de PEPE

```
(de  pepend ()
     ; détruit tous les symboles de :...
     (mapc 'remob (oblist '#.#:sys-package:colon))
     ; enlève le trait PEPE
     (rem-feature 'pepe)
     ; redéfinit les fonctions autoload
     (libautoload pepe pepe pepefile))
```

ANNEXE N

Les séquenceurs de base

Jérôme Chailloux
Pierre Louis Neumann
Francis Dupont

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

```
(unless (= (version) 15.2)
        (error 'load 'erricf 'schedule))

(defvar #:sys-package:colon 'schedule)
```

N.1 Initialisation des ITs horloge

```
(defvar #:system:clock-tick 0.05)

(de clock ()
    (when (debug) (princn #/.))
    (suspend))

(dmd clockstart ()
     `(clockalarm #:system:clock-tick))

(dmd clockstop ()
     `(clockalarm 0.))
```

N.2 Les séquenceurs de base

```
(df parallel :ll
    ; évalue les expressions de :l en parallèle.
    ; retourne () quand toutes les expressions ont été évaluées.
    (without-interrupts
        (let ((:l (append :ll ())))    ; évite les modifs physiques du corps.
            (while :l
                (schedule (lambda (:v)
                                (newr :l (list 'resume (kwote :v))))
                          (let ((:e (nextl :l)))
                               ; ce LET doit être dans la portée du
                               ; bloc d'activation schedule!
```

```
                        (clockstart)
                        (with-interrupts (eval :e))
                        (clockstop)))))))

(df parallelvalues :ll
    ; évalue les expressions de :l en parallèle.
    ; retourne la liste des expressions évaluées dans l'ordre de :l
    (without-interrupts
        (let ((:ltask ())                       ; les tâches en suspend
              (:l :ll)                          ; les choses a faires
              (:r (makelist (length :ll) ()))   ; liste des valeurs de retour
              (:i -1))                          ; le compteur de slots
            (while (or :l :ltask)
                (schedule (lambda (:v) (newr :ltask :v))
                    (clockstart)
                    (ifn :l
                         (resume (nextl :ltask))
                         (let ((:e (nextl :l)))
                              ; ce LET doit être dans la portée du
                              ; bloc d'activation schedule!
                              (incr :i)
                              (rplaca (nthcdr :i :r)
                                      (with-interrupts (eval :e)))))
                    (clockstop)))
            :r)))

(df tryinparallel :ll
    ; évalue les expressions de :l en parallèle.
    ; retourne la première valeur calculée et arrête toutes
    ; les autres tâches en suspend.
    (without-interrupts
        (let ((:ltask ())
              (:l :ll))
            (tag :return-value
                 (while (or :l :ltask)
                        (schedule (lambda (:v) (newr :ltask :v))
                            (clockstart)
                            (let ((:e (nextl :l)))
                                 ; ce LET doit être dans la portée du
                                 ; bloc d'activation schedule!
                                 (if :e
                                     (exit :return-value
                                           (prog1
                                                (with-interrupts
                                                    (eval :e))
                                                (clockstop))))
                            (resume (nextl :ltask)))))))))

(dmd progn-no-suspend :body
     ; évalue les expressions de :l en séquence et sans être suspendu!
     `(schedule resume ,@:body))

(dmd letparallel (:lvar . :body)
     ; LET dans lequel les valeurs des variables sont évaluées
```

```
; en parallèele : la liaison s'effectaunt également en parallèle.
`(letvq ,(mapcar 'car :lvar)
        (parallelvalues ,@(mapcar 'cadr :lvar))
        ,@:body))
```

Table des matières

11 Les Outils de Mise au Point

12 Le Chargeur/Assembleur LLM3

Index des Concepts

Index de la Machine LLM3

Index des Fonctions et des Variables Le_Lisp

NAME

complice – compilateur modulaire Le_Lisp

SYNOPSIS

complice [**–parano** flagp] [**–w** flagp] [**–v**] [**–o** diro] [**–p** dirp] [**–i**] [module1 ... moduleN]

DESCRIPTION

Complice compile les modules Le_Lisp *module1...moduleN. Complice* cherche les descriptions de ces modules (fichier *moduleI.lm*) dans le path Le_Lisp de l'utilisateur (variable globale #:SYSTEM:PATH), et crée des fichiers binaire Le_Lisp (fichiers *module.lo*) dans le directory courant. Les messages d'erreur et les avertissement à la compilation sont imprimés au terminal. Si aucun message d'erreur n'est imprimé les modules ont étés compilés correctement.

L'option **–parano** permet de positionner la variable globale Lisp #:COMPLICE:PARANO? à la valeur de *flagp. Flagp* doit être l'un des deux symboles T ou NIL. Par défaut cette variable vaut NIL, *complice* suppose que les fonctions appelées par FUNCALL et APPLY n'utilisent aucune variable de l'environnement dynamique de leur appelant.

L'option **–w** permet de positionner la variable globale Lisp #:COMPLICE:WARNING? à la valeur de *flagw. Flagw* doit être l'un des deux symboles T ou NIL. Par défaut cette variable vaut T, *complice* imprime des messages d'avertissement.

L'option **–v** permet de passer *complice* en mode verbeux. Les formes Lisp évaluées dans l'environnement du compilateur sont notamment imprimées à l'écran.

L'option **-o** permet de stocker les fichiers résultats de la compilation des modules dans le directory Par défaut ces fichiers sont crées dans le directory courant.

L'option **–p** permet rajouter le directory **dirp** en tête du path utilisateur Le_Lisp. Cette option peut appraître plusieurs fois dans la ligne de commande, ce qui permet d'ajouter plusieurs directories au path.

L'option **–i** rentre sous le toplevel interactif de complice. Ceci permet de lancer un système Le_Lisp dont le compilateur (fonctions Lisp COMPILE et COMPILE-ALL-IN-CORE) est *complice.* Les arguments qui suivent cette option sont passés tels quels à l'interprète Le_Lisp (voir la commande lelisp).

L'appel simple : complice correspond a' l'appel : complice -i

SEE ALSO

Le_Lisp V15.2, le manuel de référence, Jérôme Chailloux et al., INRIA, Rocquencourt, 78153 Le Chesnay Cedex.

FILES

cmplc	l'interprète Le_Lisp muni du compilateur complice
/tmp/cp$$	fichiers temporaires crées durant la compilation

NAME

config – construit une image mémoire Le_Lisp

SYNOPSIS

/usr/local/lelisp/vax/config name bin conf [<tailles-lelispbin>]

DESCRIPTION

Construit l'image mémoire et le script shell de lancement d'un système Le_ Lisp de nom *name.*

Cette commande doit être lancée depuis le directory système de la distrbution Le_Lisp. Ce directory a le même nom que la machine utilisée (vax, sm90, sun, etc.).

Bin est le nom de l'interpréte Le_Lisp à utiliser. En général ce nom est *lelispbin.* On utilise parfois des binaires contenant l'interprète Le_Lisp et des programmes C ou FORTRAN. Ceci permet de construire une image mémoire contenant des programmes définissables comme *fonctions externes* de l'interprète Le_Lisp (voir le manuel de référence, section 14).

Conf est un fichier Le_Lisp qui décrit la construction de l'image mémoire.

Config lance l'interprète *lelispbin* qui lit d'abord le fichier *startup.ll* de la bibliothèque Le_Lisp, puis évalue les formes Lisp du fichier *conf.* L'évaluation de ces formes doit créer une image mémoire de nom *name.core* dans le sous-directory *llcore* du directory d'installation de Le_Lisp (/usr/local/lelisp usuellement). Durant l'évaluation du fichier *conf* la variable globale Lisp #:SYSTEM:NAME a pour valeur le symbole Lisp *name.* Ceci permet d'utiliser le même fichier de configuration pour construire des systèmes de nom différents (avec différentes tailles de zones par exemple).

Config crée ensuite un script shell de nom *name* qui permet de lancer l'interprète *lelispbin* en restaurant l'image mémoire construite.

On peut optionnellement donner des arguments qui décrivent la taille des différentes zones mémoire de l'interprète. Ces arguments sont passés tels quels à *lelispbin.*

EXEMPLE

Construire un système de nom *lelisp* avec le binaire *lelispbin,* le fichier de configuration *lelispconf.ll* et une zone code de 440 koctets.

```
config lelisp lelispbin lelispconf.ll -code 400
```

Le fichier lelispconf.ll contient les formes Lisp suivantes :

```
(load-std () t t t t t)
(llcp-std #:system:name)
```

SEE ALSO

Le_Lisp V15.2, le manuel de référence, Jérôme Chailloux et al., INRIA, Rocquencourt, 78153 Le Chesnay Cedex.

NAME

lelisp, lelisp+, lelisp++ – lance le système Le_Lisp

SYNOPSIS

lelisp [**–r** file] [cons]

DESCRIPTION

Lance le système Le_Lisp. Après impression d'une bannière identifiant le système, on se trouve sous une boucle de dialogue, caractérisée par l'impression du signe d'invite "? ". La fonction (END) termine la session Lisp et retourne au shell. Les commandes *lelisp+* et *lelisp++* ont le même comportement mais lancent un Lisp avec de plus grandes zones de données. *Lelisp* utilise environ 850 koctets de mémoire centrale, *lelisp+* 1 Mégaoctet, et *lelisp++* 1,6 Mégaoctet.

L'option **–r** permet de restaurer l'image mémoire de nom **file**, à l'entrée de l'interprète Le_Lisp. Cette image doit avoir été créée par le même système Le_Lisp, sinon le message d'erreur *image mémoire non compatible* est imprimé et Le_Lisp termine immédiatement.
Par défaut l'image mémoire de nom /usr/local/lelisp/llcore/lelisp.core est restaurée. Cette image mémoire contient les outils de mise au point, et le compilateur Le_Lisp.

L'argument optionnel **cons** est un nombre qui précise, en 8Kcons, la taille de la zone allouée aux cellules de listes. La valeur par défaut de cet argument est 4, ce qui correspond à une zone de 256 Koctets.

SEE ALSO

Le_Lisp V15.2, le manuel de référence, Jérôme Chailloux et al., INRIA, Rocquencourt, 78153 Le Chesnay Cedex.

FILES

lelisp script shell permettant de lancer le système
/usr/local/lelisp/llcore/lelisp.core
image mémoire restaurée par défaut
/usr/local/lelisp/<sys>/lelispgo lanceur Le_Lisp
/usr/local/lelisp/<sys>/lelispbin interptète Le_Lisp

NAME

lelispgo, lelispbin – lanceur et interprète du système Le_Lisp

SYNOPSIS

lelispgo lelispbin [<options-lelispbin>]

lelispbin fd [**–stack** stack] [**–code** code] [**–heap** heap] [**–number** number] [**–float** float] [**–vector** vector] [**–string** string] [**–symbol** symbol] [**–cons** cons] [**–r** core] [file]

DESCRIPTION

Lelispgo lance l'interprète *lelispbin* dont il recoit le nom en paramètre. Un *pipe* est établi entre les deux processus. Le file descriptor d'écriture de ce pipe est passé comme premier argument du processus *lelispbin*. Les arguments supplémentaires éventuels de *lelispgo* sont passés tels quels à *lelispbin*.

Lelispbin est l'interprète du système Le_Lisp. Il recoit en premier argument le file descriptor du *pipe* qui lui permet de commmuniquer avec le lanceur *lelispgo*. Les options **–stack...–cons** permettent de préciser les tailles de chacune des zones de données de l'interprète. Les unités des paramètres numériques *stack...cons* sont définies dans le manuel de référence Le_Lisp v15.2.

L'option **–r** permet de restaurer l'image mémoire de nom *core* au lancement du système. Cet argument exclus l'option **file**.

L'option **file** permet de demander la lecture du fichier de nom *file* au lancement du système. Ce fichier est lu par la fonction lisp INPUT. Cette option exclus l'option **–r**.

Si aucune des options **–r** ou **file** n'est fournie, *lelispbin* lit le fichier *startup.ll* de la bibliothèque Le_Lisp.

DIAGNOSTICS

Si *lelispgo* ne peut pas lancer le processus *lelispbin* (fichier inexistant, droits d'accès, etc.) il termine immédiatement en imprimant un message d'erreur.

Si le fichier initial ou l'image mémoire à restaurer n'existent pas, ou sont illisibles par l'utilisateur, *lelispbin* termine immédiatement en imprimant un message d'erreur.

SEE ALSO

Le_Lisp V15.2, le manuel de référence, Jérôme Chailloux et al., INRIA, Rocquencourt, 78153 Le Chesnay Cedex.

NAME

newdir – mets à jour les paths absolus du système Le_Lisp.

SYNOPSIS

/usr/local/lelisp/vax/newdir

DESCRIPTION

Mets à jour les paths absolus définis dans les fichiers suivants de la distribution Le_Lisp : llib/startup.ll, ceyx/make.ll, system/makefile, system/cmplc.

Cette commande doit être lancée depuis le directory système de la distrbution Le_Lisp. Ce directory a le même nom que la machine utilisée (vax, sm90, sun, etc.).

Il faut lancer cette commande une fois et une seule après chaque installation de Le_Lisp dans un nouveau directory.

Cette commande nécessite les droits d'accès en écriture sur les fichiers à mettre à jour.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

Dépôt légal: 300786 / 500

ISBN 2 - 7261 - 0448 - 7