LE LANGAGE LISP

I - INTRODUCTION

LISP signifie LISt Processing (traitement de liste). Ce langage a été inventé par J. McCarthy au M.I.T. en 1960. Son créateur le destinait aux problèmes non numériques alors que la préoccupation générale des langages nés à cette époque était le calcul numérique. De ce fait, LISP est resté longtemps dans l'oubli. Son essor est dû en partie à l'Intelligence Artificielle. Aujourd'hui, c'est le langage le plus utilisé dans ce domaine.

Depuis sa création, LISP a longtemps échappé à toute standardisation. De fait, il existe à l'heure actuelle plusieurs dialectes de LISP. Nous étudierons "CommonLisp", langage qui fait l'objet d'une normalisation ANSI depuis 1984. Guy L. Steele (Carnegie Mellon University, Lexington, Massachusetts) a publié la norme établie par le comité X3J13 en Octobre 1989. CommonLisp présente l'avantage de disposer d'un compilateur rendant les programmes aussi efficaces en exécution que ceux développés dans des langages plus traditionnels.

Ce polycopié a pour objectif d'aider à l'initiation au langage LISP en évitant une lecture complète de la documentation CommonLisp. Dans un deuxième temps, cette lecture est cependant vivement conseillée afin d'acquérir une vue plus précise et plus globale du langage. Une documentation en ligne est disponible sur le site Web de l'ENSI Caen.

II - LES OBJETS DE LISP

II.1 - L'alphabet

- Les lettres a,b, ... ,z, A, B, ... Z (par défaut les minuscules sont transformées en majuscules sauf dans les chaînes de caractères).
- Les chiffres 0,1,...,9.
- D'autres caractères : ! " # $ & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ { ∧ } ` ~

Remarque : l'emploi de quelques caractères spéciaux comme ' (le caractère quote), %, # ,... , qui ont une signification particulière en LISP, est délicat. Il faut en général les entourer de guillemets pour les faire figurer dans une chaîne de caractères ou encadrer la chaîne de caractères qui les contient par deux barres verticales s'il s'agit d'un symbole (ex : |exemple.nom|).

Les commentaires de fin de ligne sont précédés du caractère ;
Les commentaires sur plusieurs lignes commencent par #| et se terminent par |#

II.2 - Les objets atomiques

Les objets atomiques représentent les informations élémentaires.

Exemples :
ALICE CH4 12CENTS 167 tic-tac exemple_d_atome
|to:to| (pour to:to) "exemple de chaînes de caractères"

On distingue trois sortes d'objets atomiques : les nombres, les symboles et les chaînes de caractères.

CommonLisp manipule des nombres entiers, des nombres rationnels et des nombres réels. Leur notation correspond à la notation usuelle Pascal en ce qui concerne les entiers et les réels. Les rationnels sont notés x/y (ex: 1/2).

Les symboles contiennent au moins un caractère non numérique, ceci en tenant compte de la représentation flottante des nombres (2E3 est un nombre et 2A3 est un symbole). Ils jouent le rôle d'identificateurs et servent à nommer les variables et les fonctions.

Les chaînes de caractères sont désignées par une suite de caractères encadrée de guillemets. Le caractère " est un séparateur, il n'est donc pas possible de le faire apparaître dans une chaîne de caractères.

II.3 - Les listes

Une liste est représentée par une parenthèse ouvrante, une parenthèse fermante et entre les deux, des objets atomiques ou d'autres listes, séparés par des blancs.

Exemples :

(X1 X2 X3) ((X1) (X2 X3) X4) (lisp (est un) langage)

La liste vide est désignée par NIL ou () . Cette liste est aussi considérée comme un atome. C'est la seule liste ayant cette propriété.

Pour toute liste L=(A1 ... AN), les Ai s'appellent les éléments de L. Tout traitement d'une liste laissant invariant ses éléments, s'appelle un traitement au premier niveau.

Exemple :

L'inversion au premier niveau de la liste
((A B) (A B C) (B C)) donne : ((B C) (A B C) (A B))

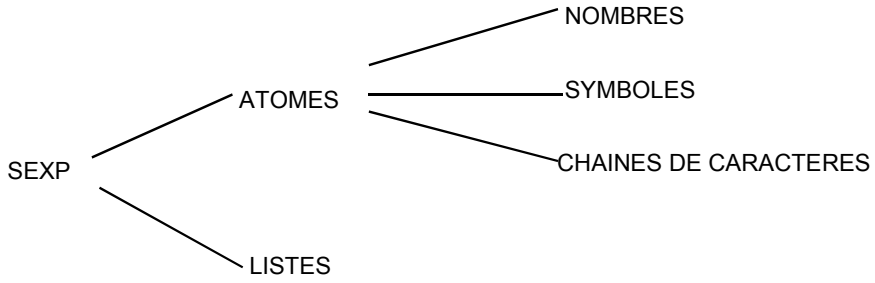
L'inversion de la même liste à tous les niveaux donne :
((C B) (C B A) (B A))

Les listes servent à regrouper (structurer) des informations pour mieux les traiter. La structure de liste sera la principale structure de données LISP examinée dans ce cours. CommonLisp manipule également des vecteurs,des structures et des tables de hachage.

II.4 - Les expressions symboliques

Une expression symbolique (ou S-expression ou Sexp) est un atome ou une liste.

En résumé, les objets LISP sont subdivisés de la manière suivante:



II.5 - Les fonctions

La notion de fonction est une notion primordiale en LISP. Les fonctions sont les seuls moyens de former des programmes. Un grand nombre de fonctions manipulant des atomes et des listes sont prédéfinies et sont à la disposition de l'utilisateur dès le lancement du système. L'utilisateur a par ailleurs des outils pour regrouper ces fonctions et contrôler l'enchaînement de leur exécution. Le corps d'une fonction est l'ensemble des instructions qui indique à l'interpréteur la marche à suivre pour transformer les arguments d'entrée.

II.5.1 - L'appel des fonctions

L'appel d'une fonction avec ses arguments est représenté par une liste dont le premier élément est le symbole fonctionnel et les autres éléments les arguments de la fonction. Cela correspond à une notation préfixée.

L'emploi des nombres pour nommer des fonctions est interdit.

En Pascal ou en C, l'appel d'une fonction F est F(X1,X2,...,XN);
en LISP, ce sera (F X1 X2 ... XN).

Et en généralisant, 2 + 3 sera représenté par (+ 2 3) et
(X - 1)(Y + 2) par (* (- X 1) (+ Y 2)).

Une forme est une liste représentant une fonction et la suite de ses arguments. Evaluer une forme consiste à calculer la valeur de la fonction qu'elle représente pour les arguments spécifiés dans la liste (ex: (GCD 14 3)).

II.5.2 - Exemples de fonctions

Nous allons étudier quelques fonctions numériques (ayant pour arguments des nombres). Celles-ci sont des fonctions de l'arithmétique générique. Leurs arguments peuvent être soit des entiers, soit des rationnels, soit des réels. Le type du résultat dépend du type des arguments. Il est réel si l'un des arguments est réel. Il est entier si cela est possible.

a) L'addition : fonction à n arguments, représentée par le symbole fonctionnel "+".

Exemples :

```
(+ 2 3 6 9)      ---> 20
(+ 8 -2 6)       ---> 12
(+ 8)            ---> 8
(+)              ---> 0
(+ 2 2.0)        ---> 4.0
(+ 2 2/3)        ---> 8/3
```

b) La soustraction : fonction à n arguments, représentée par le symbole fonctionnel "-".

Exemples :

```
(- 4 2)          ---> 2
(- 10 2 3 4)     ---> 1      (= 10-2-3-4)
(- 8)            ---> -8
```

c) La multiplication : fonction à n arguments, représentée par le symbole fonctionnel "*".

Exemples :

```
(* 2 3 5)        ---> 30
(* 5)            ---> 5
(*)              ---> 1
```

d) La division : fonction à n arguments, représentée par le symbole fonctionnel "/".

Exemples :

```
(/ 12 6)         ---> 2
(/ 123.45 6.7)   ---> 18.4254
(/ 21 5)         ---> 21/5

(/ 2)            ---> 1/2
(/ 21 5.0)       ---> 4.2
```

II.5.3 - Le principe d'évaluation des arguments d'une fonction

L'expression (2 + 5) * 3 s'écrit en LISP :

(* (+ 2 5) 3) [1]

L'interpréteur LISP, décelant le symbole "*" en tête de liste, tentera de multiplier entre eux les autres éléments de la liste. Cette tentative échouerait si la forme (+ 2 5) présente dans la forme [1] n'était pas évaluée auparavant. L'interpréteur LISP évaluera donc systématiquement tous les arguments de la fonction "*", avant de les multiplier entre eux. De ce fait, il évaluera également le nombre 3 présent en deuxième argument dans la forme [1] pour donner la valeur 3.

Nous venons de décrire un mécanisme fondamental en LISP, celui de l' évaluation des atomes et des formes.

II.5.4 - Evaluation des atomes et des formes

a) A chaque atome on peut associer une valeur qui sera un atome ou une liste. Evaluer un atome consiste à trouver cette valeur.

Certains atomes ont une valeur prédéfinie au lancement du système. Il s'agit notamment :

- des nombres. Ils ont leur propre valeur.
  Ex: 3 ---> 3
- T ou t : utilisé pour la valeur logique "vrai". Sa valeur est t.
- NIL ou nil : représente la valeur logique "faux". Sa valeur est NIL ou ().
- certains symboles tels que pi qui a pour valeur 3.141592653589793238 5L0.
- des mots-clés. Tout symbole commençant par : est un mot-clé et leur valeur est le symbole lui-même.
  Ex: :tout ---> :TOUT

b) Les arguments de la plupart des fonctions sont évalués avant l'entrée dans le corps de la fonction. On dit que ces fonctions sont normales.

c) Les arguments de certaines fonctions ne sont pas évalués avant l'entrée dans le corps de la fonction (mais peuvent être évalués sélectivement dans le corps de la fonction). Il s'agit des fonctions spéciales :

Exemple : à l'appel de la fonction IF qui teste si la valeur d'une expression est Vrai ou Faux, seul le test est d'abord évalué. Ensuite soit l'expression correspondant au "alors", soit l'expression correspondant au "sinon" est évaluée.

d) LISP reconnaît d'autres types de fonctions, appelées MACRO fonctions qui ont un mode d'évaluation particulier.

Remarque :

Le nombre d'arguments d'une fonction est parfois quelconque. Nous l'indiquerons à l'aide du mot &rest dans la liste des paramètres, ainsi que le prévoit la syntaxe de la définition des fonctions par la fonction "defun".

III - LES PRINCIPALES FONCTIONS PREDEFINIES

Nous allons voir dans ce chapitre les principales fonctions prédéfinies.

III.1 - Evaluation et affectation

a) (QUOTE <s>) [spéciale]

Fonction spéciale à un argument, retourne la S-expression <s> non évaluée.
Il existe un macro-caractère prédéfini qui facilite l'utilisation (très fréquente) de cette fonction; il s'agit du caractère apostrophe ' . Quand ce caractère est placé devant une expression quelconque <s>, à la lecture l'interpréteur LISP fabrique la liste (QUOTE <s>).

Exemples :

```
?(QUOTE 'A)          ?'A          ?''A          ?(QUOTE (+ 1 2))
='A                  =A           ='A           =(+ 1 2)
```

b) (EVAL <s>)

EVAL retourne la valeur de l'évaluation de la valeur de <s>.

Exemples :

```
?(EVAL '(+ 1 4))          ?(EVAL '(SETQ A 'B))
=5                        =B
?(EVAL (SETQ A 'B))
=ERREUR (si B n'a  pas de valeur)
```

c) (SET <symb> <s>)

<symb> a pour valeur un symbole. SET associe à ce symbole la valeur de <s> et retourne cette valeur. Il s'agit d'une opération d'affectation.

Exemples :

```
?(SET 'A '(X Y Z))          ?A
=(X Y Z)                    =(X Y Z)
```

d) (SETQ <sym1> <s1> ... <symN> <sN>) [spéciale]

Fonction à 2N arguments qui associe au symbole <symI> la valeur de <sI> et retourne à la valeur <sN>. Les affectations se font séquentiellement.

Exemples :

```
?(SETQ ZERO 0 UN 1 DEUX 2)      ?ZERO          ?UN
=2                              =0                    =1

?(SETQ A 'B  A 'C)              ?A                    ?B
=C                              =C                    =ERREUR
```

Ethymologie: Q figure dans SETQ pour QUOTE (les arguments d'ordre pair ne sont pas quotés).

e) (SETF <place1> <s1> ... <placeN> <sN>) [spéciale]

C'est une généralisation de SETQ où <place> indique le moyen d'accès à la donnée à modifier.

Exemples : (cf §III.2 pour la signification de la fonction CAR)

```
?(SETF L '(a b c))
=(A B C)

?(SETF (CAR L) 4)          ? (CAR L)          ?L
= 4                        = 4                = (4 B C)
```

D'autres exemples de l'utilisation du SETF seront vus dans la suite du polycopié.

f) (FUNCTION <fn>) [spéciale]

Cette fonction retourne l'interprétation fonctionnelle de <fn>.
Si <fn> est un symbole, la définition fonctionnelle associée au symbole est retournée.
Si <fn> est une lambda expression, la "fermeture lexicale" sera retournée, ce qui permet d'observer correctement, lors des évaluations successives, les règles de liaison lexicale. Une fermeture lexicale consiste en l'association d'un code et d'un environnement (valeur des variables).
L'abréviation de l'appel de cette fonction est #' .

Exemples: (voir aussi les fonctions apply et funcall)

```
?(defun adder (x) (function (lambda (y) (+ x y))))
=adder
?(setq add3 (adder 3))
=#<CCL::INTERPRETED-LEXICAL-CLOSURE #x...>
?(funcall add3 5)
 = 8

?(defun deux-fn (x)
     (list (function (lambda () x))   (function (lambda (y) (setq x y)))))
=deux-fn
?(setq fns (deux-fn 6))
=(#<CCL::INTERPRETED-LEXICAL-CLOSURE #x...> ... )
?(funcall (car fns))
= 6
?(funcall (cadr fns) 43)
= 43
?(funcall (car fns))
=43

?(defun decroissant? (x y f) (if (> (funcall f x) (funcall f y)) t nil))
= decroissant?
?(decroissant? 1 2 #'(lambda (x) (* x x)) )
=nil
?(setq reciproque #'(lambda (x) (/ 1 x)) )
=#<CLOSURE :LAMBDA (X) (/ 1 X)>
?(decroissant? 1 2 reciproque)
=t
```

III.2 - Fonctions de décomposition de listes

a) (CAR <l>) ou (FIRST <l>)

<l> est évalué et si <l> a pour valeur une liste, CAR retourne le premier élément de cette liste.

Exemples :

```
?(CAR '(A B C))              ?(CAR (A B C))              ?(CAR '((A B) C)
=A                           =ERREUR                     =(A B)

?(CAR (SETQ X '(A B)))            ?(CAR X)
=A                                =A

?(CAR '(SETQ X '(A B))            ?(CAR 'A)              ?(CAR NIL)
=SETQ                        =ERREUR              =NIL
```

Ethymologie: CAR est l'abréviation de Contenent of Adress Register. C'est le mnémonique d'une instruction d'une ancienne machine (IBM 704) sur laquelle a été implanté le premier LISP.

b) (CDR <l>) ou (REST <l>)

Si <l> a pour valeur une liste, CDR retourne la liste obtenue en enlevant son premier élément.

Exemples :

```
?(CDR '(A B C))                ?(CDR '((A) (B)))          ?(CDR '(A))
=(B C)                         =((B))                     =NIL

?(CDR (SETQ X '(A B)))         ?(CDR X)                   ?(CDR '(SETQ X '(A B)))
=(B)                           =(B)                       =(X '(A B))

?(CDR 'A)                      ?(CDR NIL)
=ERREUR                        =NIL
```

Ethymologie: CDR est l'abréviation de Contenent of Decrement Register et a la même origine que CAR.

c) (C ... R <l>)

La fonction composée Cx_1R o Cx_2R o...o Cx_nR où x_i est le caractère A ou le caractère D, peut être notée par $Cx_1x_2...x_nR$.

Exemples :

```
?(CADR '(A B C))   ;(c-à-d  (CAR (CDR ...))  )        ?(CADDAR '((A B C) E F))
=B                                                              =C
```

Le nombre de CAR et CDR à composer ne doit pas dépasser 3 ou 4 suivant les versions.

d) (LAST <s>)

Si <s> a pour valeur une liste L, LAST retourne la liste ayant pour élément le dernier élément de L. Sinon un message d'erreur est retourné.

Exemples :

```
?(LAST '(A B C))               ?(LAST '(A B (C D)))       ?(LAST 12)
=(C)                           =((C D))                   =ERREUR
?(LAST NIL)
=NIL
```

e) (NTH <i> <s>)

Retourne le nième élément de <s>, les éléments étant numérotés à partir de zéro.

Exemple :

```
?(NTH 2 '(A B C D))
=C
```

f) (POP <place> <symb>)

<place> indique le moyen d'accès à une valeur devant être une liste. Cette fonction retourne le CAR de cette liste et affecte comme nouvelle valeur à <place> le CDR de son ancienne valeur. <symb> est un argument facultatif qui reçoit la valeur retournée par POP. Cette fonction agit par modification physique de la liste.

Exemple :

```
?(SETF A '(X Y Z W))
=(X Y Z W)
?(POP A)
=X
?A
=(Y Z W)
?(POP (CDR A))
= Z
?A
=(Y W)
```

III.3 - Fonctions de construction de listes

a) (CONS <s1> <s2>)

CONS retourne la liste ayant pour CAR la valeur de <s1> et pour CDR la valeur de <s2>.

Exemples :

```
?(CONS 'A '(B C))                    ?(CONS '(B C)  '(B C))
=(A B C)                             =((B C) B C )

?(CONS '(A) NIL)
=((A))
```

Remarques :

- Les deux listes NIL et (NIL) ont NIL pour CAR et CDR. (CONS NIL NIL) retourne (NIL).
- Si la valeur de <s2> est un atome, CONS retourne une paire pointée:

```
?(CONS 'A 'B)
= (A . B)
?(CDR '(A . B))
= B                  ;(et non (B))
```

b) (LIST <s1> <s2> ... <sN>)

Retourne la liste des valeurs des <si>.

Exemples :

```
?(LIST 'A 'B 'C)                ?(LIST (+ 2 3) (CDR '(A B C)))
=(A B C)                                =(5 (B C))
?(LIST NIL)
=(NIL)
```

c) (APPEND <l1> <l2> ... <lN>)

 a pour valeur une liste Li. APPEND retourne la concaténation d'une copie du premier niveau des listes L1 .. LN-1 à la liste LN. Les arguments atomiques, sauf le dernier, n'ayant pas pour valeur la liste vide, provoquent une erreur.

Exemples :

```
?(APPEND '(A B) '(B C) '(C A))          ?(APPEND NIL '(A))
```

=(A B B C C A) =(A)

?(APPEND 'A 'B)
=ERREUR

?(APPEND '(A B) 'C '(D E))
=ERREUR

?(APPEND '(A B) 'C)
= (A B . C)

?(APPEND '(A B) () '(C))
=(A B C)

d) (PUSH <s> <place>)

<place> indique le moyen d'accès à une valeur devant être une liste. PUSH place en tête de cette liste la valeur de <s> et retourne la nouvelle liste formée. Les fonctions POP et PUSH permettent de construire et manipuler physiquement des piles-listes.

Exemple :

```
?(SETF A '(X Y Z))
=(X Y Z)
?(PUSH 'W A)
=(W X Y Z)
?A
=(W X Y Z)
?(PUSH 'U (CDDR A))
=(U Y Z)
?A
=(W X U Y Z)
```

III.4 - Autres fonctions sur les listes

a) (LENGTH <s>)

Si la valeur de <s> est une liste, LENGTH retourne le nombre d'éléments (au premier niveau) de cette liste; sinon LENGTH retourne un message d'erreur.

Exemple :

```
(LENGTH '((A B) (A B C) (B C)))     ---> 3
```

b) (REVERSE <s>)

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

Exemple :

```
?(REVERSE '(A (B C) D))
=(D (B C) A)
```

c) (SUBST <s1> <s2> <s>)

SUBST fabrique une nouvelle copie de la valeur de l'expression <s> en substituant à chaque occurrence de la valeur de <s2>, la valeur de <s1>.

Exemple :

```
?(SUBST '(A B) 'X '(X (X Y)))
=((A B) ((A B) Y))
```

d) (REMOVE <item> <l>)

REMOVE retourne une copie de <l> dans laquelle l'<item> a été supprimé.

III.5 - Prédicats et fonctions de test

(NULL <s>)
Si la valeur de <s> est NIL, NULL retourne T, sinon retourne NIL.

(ATOM <s>)
Si la valeur de <s> est un atome, ATOM retourne T, sinon retourne NIL.

(LISTP <s>)
Si la valeur de <s> est une liste, même vide, LISTP retourne T, sinon retourne NIL.

(CONSP <s>)
Si la valeur de <s> est une liste non vide, CONSP retourne T, sinon retourne NIL.

(SYMBOLP <s>)
Si la valeur de <s> est un symbole, SYMBOLP retourne T, sinon retourne NIL.

(FUNCTIONP <s>)
Si la valeur de <s> est une fonction, FUNCTIONP retourne T, sinon retourne NIL.

(NUMBERP <s>)
Si la valeur de <s> est un nombre, NUMBERP retourne T, sinon retourne NIL.

(INTEGERP <s>)
Si la valeur de <s> est un nombre entier, INTEGERP retourne T, sinon retourne NIL.

(RATIONALP <s>)
Si la valeur de <s> est un nombre rationnel, RATIONALP retourne T, sinon retourne NIL.

(CONSTANTP <s>)
Si la valeur de <s> est constante, CONSTANTP retourne T, sinon retourne NIL.

(FLOATP <s>)
Si la valeur de <s> est un nombre réel, FLOATP retourne T, sinon retourne NIL.

(EQUAL <s1> <s2>)
Teste si la valeur de <s1> et la valeur de <s2> sont égales. Si oui, EQUAL retourne T, Sinon EQUAL retourne NIL.

(EQ <s1> <s2>)
Si les valeurs des <si> sont des symboles, EQ retourne T s'ils sont égaux.
Si ce ne sont pas des symboles, EQ va tester l'identité des représentations internes des arguments, c'est-à-dire tester s'ils ont la même adresse physique.

(EQL <s1> <s2>)
Teste si des nombres ou des symboles sont égaux, même s'ils n'ont pas la même adresse physique.

(NOT <s>)
Si la valeur de <s> est T, cette fonction retourne NIL, sinon retourne T.

(MEMBER <s> <l> &key :test)

&key indique la possibilité d'ajouter des arguments dans la forme d'appel avec le mot-clé :TEST

Teste si la valeur x de <s> est un élément (au premier niveau) de la liste L, valeur de <l>. Si OUI, MEMBER retourne la suite des éléments de L commençant par <s>, et sinon, retourne NIL. Si la valeur de <l> est un atome, MEMBER retourne un message d'erreur. Cette fonction utilise le prédicat EQL par défaut. Si on souhaite utiliser un autre prédicat pour effectuer le test, il faut le noter dans la liste d'arguments après le mot-clé :TEST.

Exemples :

```
?(MEMBER 'B '(A B C A B))                    ?(MEMBER 'A 'B)
=(B C A B)                                   =ERREUR

?(MEMBER '(A B) '((A) (A B) (B)))            ?(MEMBER 'A '((A)))
=NIL                                         =NIL

?(MEMBER '(A B) '((A) (A B) (B)) :test 'EQUAL)    ?(MEMBER 'F '(A . F))
=((A B) (B))                                      =ERREUR
```

(= <n1> <n2> ... <nN>)
Si les valeurs de <n1>, <n2>, ... <nN> sont des nombres égaux, cette fonction retourne T, sinon elle retourne NIL.

(< <n1> <n2> ... <nN>)
Si les valeurs des <nI> sont croissantes de façon monotone, cette fonction retourne T, sinon elle retourne NIL.

(> <n1> <n2> ... <nN>)
Si les valeurs des <nI> sont décroissantes de façon monotone, cette fonction retourne T, sinon elle retourne NIL.

(<= <n1> <n2> ... <nN>)
Si les valeurs des <nI> sont non décroissantes de façon monotone, cette fonction retourne T, sinon elle retourne NIL.

(>= <n1> <n2> ... <nN>)
Si les valeurs des <nI> sont non croissantes de façon monotone, cette fonction retourne T, sinon elle retourne NIL.

(/= <n1> <n2> ... <nN>)
Si les valeurs des <nI> sont toutes différentes, cette fonction retourne T, sinon retourne NIL.

(ZEROP <n>)
Si la valeur de <n> est un nombre et vaut 0, cette fonction retourne T, sinon NIL.

(PLUSP <n>)
Si la valeur de <n> est un nombre et est positive, cette fonction retourne T, sinon NIL.

(MINUSP <n>)
Si la valeur de <n> est un nombre et est négative, cette fonction retourne T, sinon NIL.

(ODDP <n>)
Si la valeur de <n> est un nombre impair, cette fonction retourne T, sinon NIL.

(EVENP <n>)
Si la valeur de <n> est un nombre pair, cette fonction retourne T, sinon NIL.

III.6 - Fonctions de contrôle

Ce sont des fonctions qui contrôlent l'exécution des instructions en imposant une exécution linéaire ou au contraire en rompant la linéarité ou encore en effectuant une exécution en boucle.

Ces fonctions sont presque toujours des fonctions spéciales. Il existe une grande variété de structures de contrôle en LISP.

III.6.1 - Les fonctions de contrôle conditionnelles

a) (IF <s1> <s2> <s3>) [spéciale]

Si la valeur de <s1> est différente de NIL, IF retourne la valeur de <s2>; sinon, IF évalue <s3>.
IF permet de construire une structure de contrôle de type "si alors sinon" .
Exemples :

?(IF T 1 2)	?(IF NIL 1 2)	?(IF NIL 1)
=1	=2	=NIL

Schéma de résolution de l'équation de second degré $aX^2 + bX + c = 0$:

```
(IF (= a 0)
      (IF (= b 0)
            (IF (= c 0)
                (PRINT "une infinité de solutions")
                (PRINT "aucune solution"))
            (PRINT "une seule solution"))
      (LET ((DELTA (- (* b b) (* 4 a c))))
           (IF (> DELTA 0)
               (PRINT "2 solutions réelles")
               (IF (= DELTA 0)
                   (PRINT "solution double")
                   (PRINT "2 solutions imaginaires")))))
```

b) (WHEN <test> <s1> ... <sN>) [spéciale]

Si <test> est Vrai, WHEN évalue chacune des expressions <sI>.

c) (UNLESS <test> <s1> ... <sN>) [spéciale]

Si <test> est Faux, UNLESS évalue chacune des expressions <sI>.

d) (COND <l1> <l2> ... <lN>) [spéciale]

COND est la fonction conditionnelle la plus complète de LISP.

Chaque <li> est une liste appelée clause, ayant la structure suivante:

(<ss> <s1> <s2> ... <sN>) où <ss> est une expression quelconque et <s1> ... <sN> le corps de la clause. COND sélectionne la première <li> dont la valeur de <ss> est différente de NIL. Si le nombre des <si> de la clause est nul, COND retourne la valeur de <ss>, sinon les <s1> ... <sN> de la clause sont évalués et la valeur de <sN> est retournée. Si aucune clause n'est sélectionnée, COND retourne NIL.

Exemple :

```
?(defun test (x y)
    (COND
       ((> x y)   'sup)
       ((= x y)   'egal)
       (t   'inf))
  )
= test
?(test 12 5)
= sup
?(test 5 8)
=inf
```

e) (CASE <forme-clé> <l1> <l2> ... <lN>) [spéciale]

Chaque <li> est de la forme:
((<clé1> <clé2> ... <cléN>) | <clé> <s1> <s2> ... <sN>)

Exemple:

```
(CASE objet
     (cercle  (* pi r r))
     ((sphere  balle) (* 4 pi r r)))
```

Le mot-clé OTHERWISE permet d'indiquer que faire dans tous les autres cas.

f) (TYPECASE <expr> <l1> <l2> ... <lN>) [spéciale]

Chaque <Li> est de la forme:
(<type> <s1> <s2> ... <sN>)

g) (OR <s1> <s2> ... <sN>) [spéciale]

Evalue dans l'ordre <s1> <s2> ... ,s'arrête au premier <si> dont la valeur est distincte de NIL et retourne cette valeur.

Exemple :

```
?(OR (CDR '(A)) NIL (+ 2 3) (+ 3 4))
= 5
```

h) (AND <s1> <s2> ... <sN>) [spéciale]

Evalue dans l'ordre <s1> <s2> ... et s'arrête au premier <si> dont la valeur est NIL. Si aucun des <si> n'a pour valeur NIL, AND retourne la valeur de <sN>, sinon AND retourne NIL.

Exemple :

```
?(AND (SETF X '(A B)) (CDDR X) (SETF Y 0))
=NIL
?X                    ?Y
=(A B)                =ERREUR
```

III.6.2 - Les fonctions de contrôle séquentielles

a) (PROGN <s1> <s2> ... <sN>)

Evalue en séquence <s1>, ... <sN> et retourne la valeur de <sN>.

Exemple :

Pour créer une structure de contrôle de la forme
Si X Alors A1 A2 ... AN Sinon B1 B2 ... BM :

(IF X (PROGN A1 ... AN) (PROGN B1 B2 ... BM))

b) (PROG1 <s1> <s2> ... <sN>)

Evalue en séquence <s1>, ... <sN> et retourne la valeur de <s1>.

III.6.3 - Les blocs

Il est possible en CommonLisp de sortir directement d'un bloc d'instructions. L'utilisation des fonctions correspondantes doit cependant rester limitée à des cas particuliers de problèmes tels que la gestion des erreurs.

a) (BLOCK <nom> {<forme>}*)

Définit un bloc de nom <nom>. Les formes définies dans ce bloc sont évaluées en séquence sauf en cas de rencontre d'un appel aux fonctions RETURN ou RETURN-FROM. La valeur retournée est celle de la dernière forme évaluée.

b) (RETURN [<résultat>])

Sort du bloc courant (par exemple une itération) et retourne la valeur de <résultat>.

```
(loop while t do
   (case (read)
     (1 (print "ok"))
     (otherwise (return ()))))
```

c) (RETURN-FROM <nom> [<résultat>])

Sort du bloc de nom <nom> et retourne la valeur de <résultat>.

III.6.4 - Les fonctions de répétition

En LISP, toutes les structures répétitives définies dans les langages tels que PASCAL ou C sont implantées. On retrouve l'équivalent des fonctions WHILE, UNTIL, REPEAT et des fonctions itératives FOR et DO.

a) DO est une fonction générale d'itération à une ou plusieurs variables de contrôle.

```
(DO  ( { <var> | (<var> [<init> [<pas>] ) } * )
       (<test-fin> {<resultat>}* )
```

{<déclaration>}* { <s-exp> }*)

Exemple:

```
?(DO    ((x foo (cdr x))
         (y bar (cdr y))
         (i 1 (+ i 1))
        ((< i 5) i)
        (format nil "x= ~s  y= ~s  i= ~s "  x y i))
```

b) DOLIST est une fonction d'itération sur les listes.

```
?(DOLIST (x '(a b c)) (prin1 x))
ABC
=NIL
```

c) DOTIMES est une fonction élémentaire d'itération sur les entiers.

La variable est initialisée à 0 et le test >= est utilisé pour sortir de la boucle.

```
?(DOTIMES (i 10 i) (prin1 i))
0123456789
=10
```

d) LOOP et ses clauses de contrôle (que l'on peut combiner) permettent d'écrire tout type de boucle. La description qui en est faite ici n'est pas complète (se référer à la documentation).

(LOOP [named <nom>] {<clause-variable>}* {<clause-principale>}*)

- Clauses d'itération:
  FOR | AS <var> FROM <expr> TO | DOWNTO <expr> BY <expr>
  <var> IN | ON | BEING <liste>
  REPEAT <s-exp>

- Clauses de test de fin de boucle
  WHILE | UNTIL | ALWAYS | NEVER | THEREIS <s-expr>

- Clauses d'accumulation de valeurs
  COLLECT | APPEND | NCONC | SUM | COUNT | MIINIMIZE | MAXIMIZE <s-expr>

- Clause d'initialisation
  WITH

- Clauses conditionnelles
  IF | WHEN | UNLESS | ELSE | END

- Clauses inconditionnelles
  DO | RETURN <s-expr>

Exemples :

(LOOP FOR i FROM 1 TO 10 DO (PRINT i)) --> 1 2 ... 10 NIL
(avec un nombre par ligne)

(LOOP AS i BELOW 5 DO (PRINT i)) --> 0 1 2 3 4 NIL
(avec un nombre par ligne)

(LOOP FOR item IN '(1 2 5) DO (PRINT item)) --> 1 2 5 NIL
(avec un nombre par ligne)

```
(LOOP WHILE (hungry-p) DO (eat))

(LOOP FOR x IN '((a) (b) (c))  APPEND x)   -->  (A B C)

(LOOP FOR x FROM 1 TO 5  AND y = NIL THEN x  COLLECT (LIST x y))
        -->  ((1 NIL) (2 1) (3 2) (4 3) (5 4))          (x et y sont incrémentés en parallèle)

(LOOP FOR j FROM 0 TO 10 WHILE test DO  ...

(SETF  a  4)
(LOOP (setf a (+ a  1))  (when  (> a  7) (return a)) )
        -->   8
```

III.7 - Fonctions numériques

CommonLisp possède plusieurs types de nombres: les nombres entiers à précision illimitée, les rationnels exacts et les nombres flottants. Par ailleurs, CommonLisp dispose également de fonctions pour traiter les nombres complexes, qui sont notés #C(<n1> <n2>) (ex: #C(0 1)).

Le résultat des fonctions de cette catégorie sont du même type que les arguments, si cela est possible.

(+ &rest <nombres>)

Retourne la somme des valeurs des arguments (&rest indique ici que le nombre d'arguments de la fonction est quelconque). La fonction retourne 0 s'il n'y a pas d'argument.

(- <nombre> &rest <nombres>)

Cette fonction retourne la valeur de <nombre> à laquelle on soustrait les valeurs des autres arguments.

(* &rest <nombres>)

Retourne le produit des valeurs des arguments (valeur 1 par défaut).

(/ <nombre> &rest <nombres>)

Retourne le quotient de la division de la valeur de <nombre> par celles des autres arguments. Si les valeurs des nombres sont entières, le quotient devra être entier, sinon le résultat sera un nombre rationnel. Si seul <nombre> est présent, le résultat est égal à 1/<nombre> .

(REM <n1> <n2>) ou (MOD <n1> <n2>)

Retourne le reste de la division de la valeur de <n1> par celle de <n2>.
Exemples :
  (REM 16 7) ---> 2
  (REM 21 -8) ---> 5

(**1+** <n>)

Retourne N+1, où N est la valeur de <n>.

(INCF <place> [<écart>])

Incrémente la valeur de <place> de la valeur de <écart> (1 par défaut).

(**1-** <n>)

Retourne N-1, où N est la valeur de <n>.

(DECF <place> [<écart>])

Décrémente la valeur de <place> de la valeur de <écart> (1 par défaut)..

(ABS <n>)

Retourne la valeur absolue de la valeur de <n>.

Fonctions de conversion:

(COERCE <nombre> <type-résultat>)

Exemple : (COERCE 4 'float) --> 4.0

(FLOAT <n>)

(RATIONAL <n>)
(NUMERATOR <rationnel>)
(DENOMINATOR <rationnel>)

(FLOOR <nombre> &optional <diviseur>) : entier inférieur et distance à l'entier supérieur
(CEILING <nombre> &optional <diviseur>)
(TRUNCATE <nombre> &optional <diviseur>)
(ROUND <nombre> &optional <diviseur>)

Ces fonctions retournent 2 valeurs (séparées par un ; et un retour à la ligne).

Exemples:

```
?(FLOOR 5.2)    ?(CEILING 5.2)          ?(TRUNCATE 5.2)      ?(ROUND 5.8)
=5 ;        =6 ;     =5 ;     =6 ;
  0.8000002          -0.8000002      0.2000002        -0.19999999
```

Autres fonctions

(MIN <nombre> &rest <nombres>) : calcul du plus petit nombre

(MAX <nombre> &rest <nombres>) : calcul du plus grand nombre

(GCD &rest <entiers>) : calcul du plus grand diviseur commun

(LCM &rest <entiers>) : calcul du plus petit commun multiple

(SIN <nombre en radians>) : fonction sinus

Autres fonctions numériques: expt(ex: 2^5), exp, log, sqrt, phase, sin, cos, asin, acos, atan, sinh

III.8 - Définition des fonctions

Le programmeur a souvent besoin de regrouper plusieurs instructions dépendantes de N paramètres <x1>, <x2>, ... <xN> et de les faire exécuter dans divers endroits du programme avec des valeurs différentes pour ces paramètres. Il est alors commode d'attribuer un nom à ce regroupement d'instructions et d'utiliser ce nom pour activer ces instructions. On dira dans ce cas qu'on a défini une

fonction avec les paramètres formels <x1> <x2> ... <xN>. Le nom attribué sera le nom de la fonction, les instructions regroupées constitueront son corps et ses arguments au moment de son appel ses paramètres actuels ou effectifs.

L'une des manières d'arriver à cette fin en LISP est d'utiliser la fonction DEFUN (l'abréviation de DEfinir FUNction).

III.8.1 - Définition des fonctions normales

a) Principe de base

(DEFUN <symb> <lvar> [{<déclaration>}* | <documentation> {<forme>}*) [spéciale]

où :
- <symb> est un symbole, qui sera le nom de la fonction.

- <lvar> est une liste (éventuellement vide) qui sera la liste des paramètres formels de la fonction. Elle est de la forme:

```
<lvar> = ( {<varObl>}*
         [&optional {<var> | (<var> [<val-init> [<statut-var>]] )}*]
         [&rest <varFac>]
         [&key {<var> | ( {<var> | (<mot-clé> <var>)} [<val-init> [<svar>]] )}*]
              ... )
```

- <forme> sont des formes ou des atomes évaluables qui constitueront le corps de la fonction.

L'appel de la fonction <symb> se fera par :
(<symb> <arg1> <arg2> ... <argP>)

Le nombre P de paramètres effectifs devra être obligatoirement égal au nombre de paramètres formels <varObl>, s'il n'est pas prévu de paramètres facultatifs <varFac>.

Pour évaluer cette forme, l'interpréteur procèdera de la manière suivante :

Les <argi> seront évalués et les variables définies dans <lvar> prennent les valeurs correspondantes.

On remarquera que les <argi> sont évalués avant d'entrer dans le corps de la fonction. Là réside une différence essentielle entre les fonctions de type EXPR et celles de type MACRO.

Notons enfin que DEFUN est elle-même une fonction spéciale et qu'elle retourne le symbole <symb> comme valeur.

Exemples :

```
?(DEFUN AJOUT (ATOME LISTE)
          (IF   (MEMBER ATOME LISTE)
                LISTE
                (CONS ATOME LISTE)))
=AJOUT

?(AJOUT 2 '(1 2 3))
=(1 2 3)
?(AJOUT 4 '(1 2 3))
=(4 1 2 3)
?(AJOUT 2 (1 2 3))
=ERREUR

?(DEFUN TOC (X)
(+ X 1))
=TOC
```

```
?(SETF X 4)
=4

?(TOC 2)                ;X prend 2 pour valeur.
=3
?X                      ;X a repris son ancienne valeur.
=4

?(SETF X 4)
=4
?(TOC (SETF X 2))       ;La sauvegarde de la valeur X se
=3                      ;fait après l'évaluation des arguments
?X                      ;X garde la valeur affectée au moment de l'appel de TOC.
=2
```

La syntaxe utilisée ci-dessus nous permet de définir des fonctions ayant un nombre fixe de variables. En changeant légèrement cette syntaxe, nous pouvons définir des fonctions ayant un nombre quelconque (non fixé à l'avance) d'arguments.

b) Fonctions ayant un nombre non fixé d'arguments

*** (DEFUN <symb> (&rest <var>) <s1> <s2> ... <sN>)**

Lors de l'évaluation de la forme (<symb> <arg1> <arg2> ... <argP>), tout se passe comme avant sauf que <var> prend comme valeur la liste des valeurs des <argi>.

Exemple :

```
?(DEFUN FIRST (&rest X)
      (CAR X))
=FIRST

?(FIRST 'A 'B 'C)              ?(FIRST (CDR '(A B C))        ?(FIRST)
=A                             =(B C)                        =NIL
```

*** (DEFUN <symb> <lvar> <s1> <s2> ... <sN>)**

Avec <lvar> de la forme (x1 x2 ... xL **&rest** xL+1).
Dans ce cas, l'appel de la fonction pourra se faire avec un nombre d'arguments supérieur ou égal à L. Les L premiers arguments seront liés normalement aux L premiers paramètres et le dernier paramètre prendra pour valeur la liste formée des arguments restants.

Exemple :

```
?(DEFUN TEST (X Y &rest Z)
       (LIST X Y Z))
=TEST

?(TEST (1+ 1) (1+ 2) (1+ 3) (1+ 4))
=(2 3 (4 5))
```

c) Fonctions ayant des paramètres optionnels

- Utilisation du mot-clé &optional dans la définition de la fonction.

Exemples:

```
(DEFUN bar (a &optional (b 2)) (+ a (* b 3)))
```

```
(bar 4 5 )        --->  19
(bar 4)  --->  10


(DEFUN foo (&optional (a 2 b) (c 3 d)) (list a b c d) )
```

Les paramètres b et d permettent de savoir comment ont été obtenues les valeurs de a et c.

```
(foo)          --->  (2 NIL 3 NIL)        ; NIL indique que la valeur est celle par défaut
(foo 6 4)      --->  (6 T 4 T)            ; T indique une valeur donnée par un argument


(DEFUN foo (a b &optional (c 4 d)) (list a b c d))

(foo 1 2)          ---> (1 2 4 NIL)
(foo 1)  ---> ERREUR "pas assez d'arguments"
(foo 1 2 3)        ---> (1 2 3 T)
```

- Utilisation du mot-clé &key dans la définition de la fonction. Ceci permet de fournir des arguments optionnels dans l'ordre souhaité en précisant le mot-clé qui les désigne.

Exemples:

```
(DEFUN  foo1 (a b &key c d) (list a b c d))

(foo1 1 2)                  ---> (1 2 NIL NIL)
(foo1 1 2 :d 8 :c 6)        ---> (1 2 6 8)
(foo1 1 2 :d 8)   ---> (1 2 NIL 8)
```

d) Fonctions retournant plusieurs valeurs

En CommonLisp, il est possible qu'une fonction retourne plusieurs valeurs qu'il sera possible de récupérer à l'aide de la fonction MULTIPLE-VALUE-BIND. La fonction doit contenir l'évaluation d'une forme VALUES regroupant les valeurs à retourner.

(VALUES &rest <args>)

(MULTIPLE-VALUE-BIND ({var}*) <forme> <expr>)
où (<var>) indique la liste des variables qui recevront les valeurs multiples (ou non) retournées par l'évaluation de <forme>; <expr> permet de préciser sous quelle forme on veut recupérer le ou les résultats de cette évaluation.

Exemple:

```
(DEFUN polar (x y)
        (VALUES (SQRT (+ (* x x) (* y y))) (atan y x)))

?(MULTIPLE-VALUE-BIND (r theta)
        (polar 3.0 4.0)
        (list r theta))
=(5.0  0.9272952)
```

III.8.2 - Définition de fonctions anonymes

Dans certains cas, il n'est pas nécessaire de donner un nom à une fonction. Lisp permet de décrire des fonctions anonymes: elles sont nommées Lambda expressions à cause de leur origine liée au Lambda calcul de Church.

(LAMBDA <lvar> [{<déclaration>}* | <documentation>] {<forme>}*)

Définit une fonction anonyme dont la liste des paramètres formels est <lvar> et le corps <forme>.

Exemple :

```
?((LAMBDA (X Y) (* X Y)) 2 3)
= 6
```

D'autres exemples d'utilisation des fonctions anonymes seront donnés lors de l'étude des fonctions d'application APPLY, FUNCALL et MAPCAR.

III.8.3 - Déclaration de variables globales

Les variables utilisées dans les fonctions sont locales et la liaison entre paramètres formels et paramètres réels à l'"entrée" dans une fonction est lexicale, c'est-à-dire fixée à la définition de la fonction. Cela concerne surtout les variables utilisées dans une fonction et n'apparaissant pas dans la liste des paramètres formels. Il est cependant possible de définir des variables globales qui sont dynamiques.
Attention: la valeur d'une telle variable n'est pas réinitialisée en cas de redéclaration, par exemple lorsque l'on recharge le fichier.

(DEFVAR <nom> [<valeur-init> [<documentation>]])

Exemple:

```
(DEFVAR *TABLE-X* 0)
```

Exemple illustrant la différence de comportement entre une variable à liaison lexicale et une variable à liaison dynamique:

```
? (defvar a 5)          ; liaison dynamique

? (setf b 30)           ; liaison lexicale

? (defun f1 () (print (list a b)))      ; a et b ne sont pas définis en tant que paramètres formels
  ; affichage d'un message signalant que b n'est pas déclarée

? (defun f2 (a) (f1))         ; (f2 10)  renvoie (10 30) car a est liée avec la valeur de
  ; "a" en tant que variable locale à f2.

? (defun f3 (b) (f1))         ; (f3 10)  renvoie (5 30) car b est liée à la valeur de "b" en
  ; tant que variable de l'environnement de définition de f1.

? (let ((a 15)) (f1))         ; renvoie (15 30)

? (let ((a 15) (b 2)) (f1))   ; renvoie (15 30). b reste une variable locale au let.
```

III.8.4 - Fonctions avec état local

Il est possible d'associer à une fonction ou à plusieurs fonctions un environnement qui sera mémorisé d'un appel à l'autre de l'une de ces fonctions. Cette fonctionnalité est réalisée à l'aide de la fonction "let". Voir la définition de la fonction "let" dans le §III.11.

Exemples:

```
? (let (x (y 1))
      (defun sql (z) (setq x y) (setq y z) (*x x)))
? (sql 2)
= 1
? (sql 3)
```

```
= 4
? (sql 1)
= 9
? (sql 1)
= 1

? (let ((x 0))
      (setq compteur #'(lambda () (setq x (1+  x)))))
? (funcall compteur)
= 1
? (funcall compteur)
= 2
```

III.9 - Fonctions d'entrée-sortie

III.9.1 - Les fonctions d'entrée de base

Toutes les lectures sont réalisées dans le flux d'entrée qui est associé soit au terminal, soit à un fichier sélectionné par la fonction INCHAN.

a) (READ &optional <flux d'entrée> <eof-error-p> <eof-value>)

Lit une S-expression (atome ou liste) et la retourne en valeur sans l'évaluer. Si le flux d'entrée est le terminal, valeur par défaut, cette fonction attend une entrée au clavier.
Dans le cas d'une lecture dans un fichier, <eof-error-p> permet de préciser si l'on souhaite déclencher une erreur en fin de fichier (NIL si non, T si oui) et <eof-value> indique la valeur retournée en cas de détection de fin de fichier (par exemple on peut choisir la valeur de retour 'eof).

Exemple :

```
?(SETF X (READ))
?(+ 1 2)                    ;Réponse de l'utilisateur (le caractère d'invite, ici "?" dépend
                            ; de l'implémentation.
=(+ 1 2)                    ;Valeur retournée.
?X
=(+ 1 2)
```

b) (READ-CHAR &optional <flux d'entrée>)

Lit un caractère sur le flux d'entrée et retourne un objet "caractère". La fonction de sortie correspondante est WRITE-CHAR.

Exemple :

```
?(READ-CHAR)
 3
=#\3
```

c) (READ-LINE &optional <flux d'entrée>)

Lit la ligne suivante sur le flux d'entrée et retourne une chaîne des caractères de cette ligne. La fonction de sortie correspondante est WRITE-LINE.

Exemple :

```
?(READ-LINE)ABC  DE
="ABC  DE"
```

```
?(READ-LINE)
AB CD 34
="AB CD 34"
```

d) (READ-FROM-STRING <string> &optional <eof-error-p> <eof-value> &key :start :end :preserve-whitespace)

Cette fonction retourne deux valeurs : l'objet lu et le numéro du premier caractère qui n'est pas lu. Les mots-clés :start et :end permettent de délimiter la portion de la chaîne que l'on veut traiter. Cette fonction permet en particulier de créer des symboles à partir d'une chaîne de caractères.

Exemples :

```
? (set (read-from-string "titi") 3)
= 3
? titi
= 3
? (setq a (read-from-string "x1")
= X1
? (set a 5)
= 5
? x1
= 5
? (setq i 1)
= 1
? (set (read-from-string (format nil "R~a" i)) 7)
= 7
? R1
= 7
```

III.9.2 - Les fonctions de sortie de base

Il existe une fonction très générale WRITE permettant de nombreuses options; ici seules sont fournis quelques fonctions simples de sortie.

a) (PRINT <s> &optional <flux de sortie>)

Edite les valeurs de <s>, effectue un saut de ligne et retourne la valeur de <s> .

Exemple :

```
 ?(PRINT "il fait beau")

"il fait beau"
"il fait beau"
```

b) (PRIN1 <s> &optional <flux de sortie>)

Idem PRINT sans saut de ligne.

c) (PRINC <s> &optional <flux de sortie>)

Idem PRIN1, mais les chaînes de caractères sont affichées sans guillements.

d) (TERPRI &optional <flux de sortie>)
Ethymologie : TERminer PRInt.

Effectue un saut de ligne et Vide le tampon de sortie,.

e) (FORMAT <destination> <chaîne-de-contrôle> &rest <args>)

Cette fonction permet d'envoyer dans le flux de sortie <destination> les caractères correspondants aux arguments dans le but de les afficher ou de les stocker dans un fichier. La chîne de contrôle précise le format d'impression.

Si <destination> = NIL, une chaîne est créée.
Si <destination> = T , les données sont envoyées sur le flux de sortie standard.
Si <destination> correspond à un fichier ouvert, les données sont envoyés vers ce flux.

En standard on peut utiliser le format ~a pour afficher toute valeur.
Liste des formats: ~d pour les décimaux; ~e et ~f pour les flottants; ~s pour les s-expressions; ~a pour les ascii; ~c pour les caractères; ~r pour les radix.

Exemples:

```
(SETF X 5 Y 7)
(FORMAT t "x= ~a ; y= ~a" x y)              --->  x= 5 ; y= 7
(FORMAT nil "x= ~a ; y= ~a" x y)            --->  "x= 5 ; y= 7"
```

Remarque: ~& et ~% font sauter une ligne.

III.9.3 - Les fonctions sur les flux d'entrée/sortie

CommonLisp gère, en tant que flux d'entrée-sortie, le terminal et un certain nombre de fichiers en entrée ou en sortie.

a) (WITH-OPEN-FILE (<flot> <chaine-chemin-fichier> :direction :input|:output|:io)
{<forme>}*)

Cette fonction permet d'ouvrir un fichier en lecture ou en écriture , et d'évaluer les formes dans ce contexte.

Exemples:

Lecture d'un fichier texte contenant une seule liste: lecture globale par un READ:

```
(WITH-OPEN-FILE (fic-animaux "animaux.dat" :direction :input)
    (setf liste (read fic-animaux)))
```

Lecture d'un fichier texte contenant des données à lire en plusieurs fois:

```
(WITH-OPEN-FILE (fic-regles "regles.dat" :direction :input)
    (let ((regle (read fic-regles nil 'eof)))
        (loop until (eq regle 'eof ) do
            ...
            (setf regle (read fic-regles nil 'eof))))
```

b) (LOAD <file>)

Permet de charger (en évaluant toutes les expressions qui s'y trouvent) le fichier lisp de nom <file>, par exemple (load "tp1.lsp") ou (load 'tp1). L'extension standard d'un fichier Lisp est ".lsp".

III.9.4 - Les fonctions d'accès aux commandes système

(SHELL <strg>) envoie au système d'exploitation hôte la ligne de commande <strg>.

(CD <strg>) permet de changer de répertoire courant.

(EXECUTE <string>) exécute des programmes autres que les commandes systèmes.

(EXIT) sort de Commonlisp.

III.9.5 - Les fonctions de compilation

a) (COMPILE <name> &optional <definition>)

Cette fonction permet de compiler une fonction définie par DEFUN ou une fonction anonyme.

Exemples:

```
?(defun foo ...)
?(compile 'foo)    ; maintenant "foo" s'exécute plus vite

? (setf f (compile nil '(lambda (a b c) (- (* b b) (* 4 a c)))))
#<COMPILED-CLOSURE NIL>
? f
#<COMPILED-CLOSURE NIL>
? (funcall f  2 4 5)
-24
```

b) (COMPILE-FILE <fichier> &key :output-file :verbose :print)

Cette fonction permet de compiler un fichier de fonctions Lisp et de mettre le résultat dans le fichier dont le nom est précisé après le mot clé :output-file.

III.10 - Fonctions d'ordre supérieur

Ce sont des fonctions qui prennent en entrée une ou plusieurs fonctions.

a) (APPLY <fn> <l>)

<fn> a pour valeur un symbole fonctionnel ou une lambda-expression, <l> a pour valeur une liste dont les éléments serviront d'arguments réels à la valeur de <fn>. APPLY applique la valeur de <fn> aux éléments de <l> et retourne le résultat de cette application.

Exemples :

```
?(APPLY  'CONS  '(a (b c)))
=(A B C)
?(APPLY 'LIST '(U (V W)  X))
=(U  (V W)  X)
?(APPLY #'(LAMBDA (X) (+ X X))  '(3))                ; #' est équivalent à (function ...)
=6
```

La fonction APPLY ne s'applique pas sur les formes spéciales:

```
?(APPLY 'SETQ '(X a Y b Z c))
=ERREUR
```

La fonction APPLY permet la construction des fonctions récursives dont le paramètre formel est une S-expression quelconque. Le lecteur est invité à examiner de près la liaison des paramètres dans les exemples suivants :

```
?(DEFUN DER (&rest X)                                ; dernier élément de X
      (IF (NULL (CDR X))
          (CAR X)
          (DER (CDR X))))
=DER
?(DER 1 2 3)
=(2 3)                                    ; pourquoi ?

?(DER 'a 'b 'c 'd)
```

```
=(b c d)

Par contre :
?(DEFUN DER (&rest X)
      (IF (NULL (CDR X))
          (CAR X)
          (APPLY 'DER (CDR X))))
=DER
?(DER 1 2 3)
=3

?(DER 'a 'b 'c 'd)
=d


De même :
?(DEFUN DER (X &rest Y)
      (IF (NULL Y)
          X
          (APPLY 'DER Y)))          ; ici la fonction donne le résultat escompté
```

b) (FUNCALL <fn> <s1> ... <sN>)

Cette fonction est équivalente à :

(APPLY <fn> (LIST <s1> ... <sN>))

et peut être définie par :

```
(DEFUN FUNCALL (&rest L)
    (APPLY  (CAR L)  (CDR L)))
```

Exemples:

```
?(FUNCALL  'CAR  '(a b c))
=A

?(FUNCALL  'LIST  'u  'y  'w)
=(U Y W)

?(FUNCALL  #'(LAMBDA (X Y)  (SUBST '+ X Y))  'a  '((b a) a))
=((B +) +)

?(SETQ F 'LIST)
=LIST

?(FUNCALL F  'a 'b 'c 'd)
=(A B C D)
```

La fonction FUNCALL peut être utilisée pour définir les fonctions d'ordre supérieur. Soit par exemple, à définir une fonction PREMIER, prenant en argument un prédicat PRED et une liste LISTE=(X1 ...XN) et retournant la première valeur PRED(Xi) qui est différente de NIL.

```
?(DEFUN PREMIER (PRED LISTE)
      (COND
         ((NULL LISTE) NIL)
         ((FUNCALL PRED (CAR LISTE)) (CAR LISTE))
         (T (PREMIER PRED (CDR LISTE)))))
=PREMIER

?(PREMIER  'NUMBERP  '(c (a b) 12 3 d))
```

```
=12

?(PREMIER  #'(LAMBDA(X) (> X 0))  '(-1 -5 0 2 6 -8))
=2
```

c) (MAPCAR <fn> <l1> ... <lN>)

<fn> a pour valeur un symbole fonctionnel ou une lambda-expression à N arguments (autant que de <li>). a pour valeur une liste <Li>. MAPCAR applique la valeur de <fn> avec les CAR des <Li>, puis avec les CADR des <Li> et ainsi de suite jusqu'à ce que l'une des listes <Li> s'épuise. MAPCAR retourne la liste des valeurs des applications successives.

Exemples :

```
?(MAPCAR  'CAR '((a b)(c d e)(d e)))
=(a c d)

?(MAPCAR  'CONS '(1 2 3) '(2 3))
=((1 . 2) (2 . 3))

?(MAPCAR  'LIST  '(a (b c) d) '((b b) c (a d))  '(1 4 5 6))
=((A (B B) 1)  ((B C) C 4)  (D (A D) 5))

?(MAPCAR  'MAPCAR  '(CAR CADR CADDR)
      '(((a b c)) ((b c d)) ((d e f))))
=((A)(C)(F))
```

On observera les mêmes limitations que dans le cas de APPLY dans l'application d'une fonction spéciale

```
?(MAPCAR 'SETQ '(X Y Z) '('a 'b 'c))
=ERREUR
```

Par contre :

```
?(MAPCAR 'SET '(X Y Z) '(a b c))
=(A B C)
?X              ?Y              ?Z
=A              =B              =C
```

L'utilisation de MAPCAR et APPLY permet d'écrire des fonctions dans un style fonctionnel concis:

```
?(DEFUN PRODUIT-SCALAIRE (VEC1 VEC2)
        (APPLY '+ (MAPCAR '* VEC1 VEC2))

?(DEFUN DIAGONALE (MAT)       ;MAT=((a11...a1n)...(an1...ann))
     (IF MAT
         (CONS (CAAR MAT)     ;Ajouter a11 (le premier
                              ;élément de MAT)
               (DIAGONALE     ;à la diagonale de la matrice
                  (MAPCAR 'CDR (CDR MAT))
               ))))           ;carrée obtenue  en  enlevant  la
                              ;première ligne et la première
                              ;colonne de MAT
```

d) (MAPC <fn> <l1> ... <lN>)

Procède de la même manière que MAPCAR sauf qu'elle retourne toujours NIL (alors que la version actuelle sur IBM retourne <l1>).

Exemple :

```
?(MAPC 'PRIN1 '(1 2 3))
123=NIL
```

e) (MAPCAN <fn> <l1> ... <lN>)

MAPCAN est identique fonctionnellement à MAPC. Cependant, chaque application doit retourner une liste en valeur et MAPCAN retourne la liste des valeurs des applications qui sont rassemblées au moyen d'un NCONC. Si une valeur n'est pas une liste, elle est ignorée.

Exemples :

```
?(MAPCAN #'(LAMBDA (X Y) (LIST (1+ X) (1- Y)))
                '(1 2 3) '(1 2 3))
=(2 0 3 1 4 2)

?(MAPCAN #'(LAMBDA (X) (IF (> X 5) (LIST X) X))  '(3 4 5 6 7 5))
=(6 7)
```

III.11 - Fonctions manipulant l'environnement

Ces fonctions vont permettre de changer l'environnement de façon temporaire. Le terme environnement est pris ici comme l'ensemble des liaisons variable-valeur.

a) (LET <l> <s1> ... <sN>) [spéciale]

<l> est une liste de liaisons de la forme ((<var1> <val1>) ... (<varN valN)).
La fonction LET va lier dynamiquement et en parallèle toutes les variables <vari> aux valeurs <vali> puis lancer l'évaluation des formes <s1> ... <sN>. Elle permet d'effectuer des calculs dans un environnement local. <var1> ... <varN> sont des variables locales.

b) (LET* <l> <s1> ... <sN>) [spéciale]

Cette fonction est identique à la fonction LET, mais les variables <vari> sont liées séquentiellement aux valeurs <vali>. Ceci permet de faire apparaitre les variables déjà liées dans les valeurs <vali>.

Exemple:

```
(DEFUN F (a b)
        (LET* ((x (+ a b)) (y (* x 2))) ...)
)
```

c) (LABELS ({ (<nom> <liste-lambda>
[[{<déclaration>}* | <documentation>]] {<forme>}*) }*) [spéciale]

Permet de créer des fonctions temporaires nommée.
Les fonctions définies par DEFUN sont dites statiques. Cela signifie qu'une fois définies, ces fonctions font partie de l'interprète jusqu'à la réinitialisation de celui-ci ou leur destruction provoquée explicitement par l'utilisateur. Nous savons par contre, que les fonctions définies par LET et LAMBDA ont une durée de vie moins longue: leur existence prend fin après l'évaluation de la forme définie par LET ou LAMBDA. Nous disons que ces fonctions sont définies de manière dynamique. Il s'agit maintenant de pousser l'étude de ce genre de fonctions plus loin, et de nous donner les moyens de définir des fonctions locales tout à fait générales, pouvant être appelées récursivement.

LABELS fonctionne comme LET. La seule différence est que l'on peut définir plusieurs fonctions locales qui pourront s'appeler récursivement.

Exemples :

```
?(DEFUN integer-power (n k)             ; calcul de n puissance k
  (LABELS ((expt0 (x k a)
             (COND ((ZEROP k) a)
                   ((EVENP k) (expt1 (* x x) (FLOOR k 2) a))
                   (T (expt0 (* x x) (FLOOR k 2) (* x a)))))
           (expt1 (x k a)
             (COND ((EVENP k) (expt1 (* x x) (FLOOR k 2) a))
                   (T (expt0 (* x x) (FLOOR k 2) (* x a))))))
    (expt0 n k 1)))
```

La fonction INTER définie ci-dessous retourne les éléments communs à deux listes L1 et L2.

```
(DEFUN INTER (L1 L2)
    (COND
        ((NULL L2) NIL)
        ((NULL L1) NIL)
        ((MEMBER (CAR L1) L2)
             (CONS (CAR L1) (INTER (CDR L1) L2)))
        (T (INTER (CDR L1) L2))))
```

Il faut éviter de faire le test (NULL L2) dans le corps de la fonction récursive. En effet, d'un appel à l'autre, INTER laisse L2 invariant mais réduit la taille de L1 et amorce l'étape de retour une fois L1 épuisé. Voici une autre version de INTER prenant mieux en compte ces particularités:

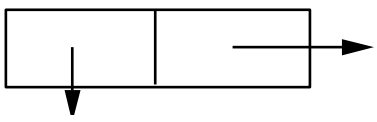
```
(DEFUN INTER (L1 L2)
    (IF L2
        (LABELS ((INTAUX (L)
                   (COND
                     ((NULL L) NIL)
                     ((MEMBER (CAR L) L2)
                         (CONS (CAR L) (INTAUX (CDR L))))
                     (T (INTAUX (CDR L))))))
            (INTAUX L1))
    ))

?(inter '(a b c)  '(e f b a))
=(A B)
```

IV - LA STRUCTURE INTERNE DE LISP

Les objets LISP sont trop complexes pour pouvoir être rangés dans un seul mot mémoire. Ils sont en général représentés par un pointeur. La création des doublets de liste et des autres objets Lisp se fait le plus souvent dynamiquement. Un dispositif général appelé GARBAGE-COLLECTOR récupère automatiquement les doublets et objets inutilisés .

IV.1 - Pointeurs et doublets

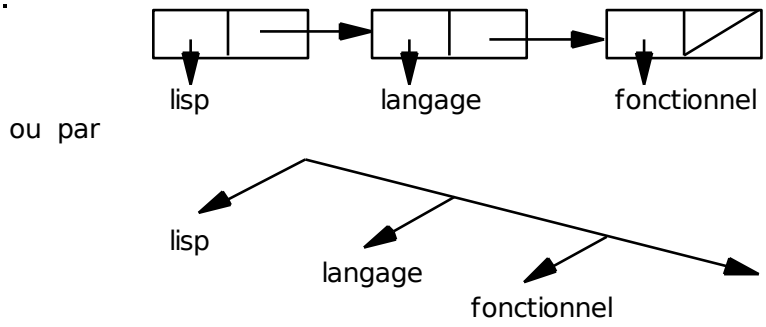
On appelle doublet (ou cons) un ensemble de deux pointeurs. Nous le représenterons par un diagramme en "boîtes et flèches":



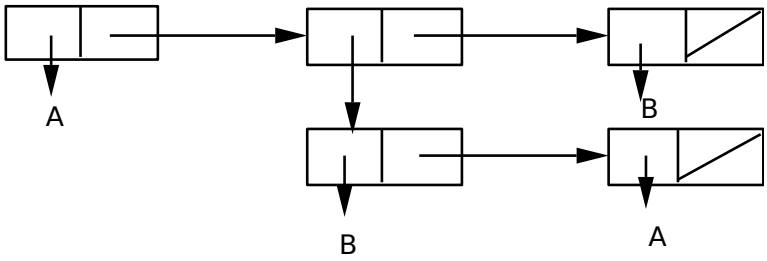
Une liste peut être considérée comme un ensemble ordonné de deux éléments, le premier s'appelant le CAR et le second le CDR de la liste. On représente donc une liste par un doublet dont la première partie pointe vers le CAR et la seconde vers le CDR de la liste.

Exemples :

a) Représentation en mémoire de la liste (LISP LANGAGE FONCTIONNEL).



b) Représentation de la liste (A (B A) B) par un diagramme en boîtes et flèches.



Notons enfin, qu'on représente quelquefois les listes par des "paires pointées". La paire pointée (A . B) représente une liste dont le CAR est A et le CDR est B.

Exemple :

En notation pointée, la liste (A (B A) B) serait représentée par:

(A . ((B . (A . NIL)) . (B . NIL)))

On peut représenter (et manipuler) de cette manière une liste dont le CDR est un atome et qui n'est pas représentable par la notation habituelle.

IV.2 - Symboles atomiques

Nous savons déjà que les symboles sont utilisés comme identificateurs pour désigner des variables et des fonctions. D'ailleurs, le même symbole peut désigner à la fois une variable et une fonction; suivant la manière dont il est appelé, il remplira tel ou tel rôle. Un symbole est créé implicitement dès sa lecture dans le flux d'entrée ou explicitement par certaines fonctions telles que MAKE-SYMBOL, GENSYM, etc.

Un symbole atomique est donc un sujet composé. En fait, c'est un pointeur sur une zone mémoire divisée en plusieurs parties. Nous aurons accès à des parties différentes suivant la manière dont nous utilisons le symbole. Nous dirons que chaque partie correspond à une propriété du symbole.

Voici quelques unes de ces propriétés que nous avons manipulées jusqu'ici :

Symbol-Value

Contient la valeur associée au symbole, considéré comme une variable. Les fonctions SET et SETQ sont utilisées pour affecter une valeur au symbole, et ainsi modifier le contenu de la Symbol-Value du symbole. On peut également utiliser la fonction SYMBOL-VALUE.

Exemple: (SYMBOL-VALUE 'INTER) retourne la valeur de INTER , ou un message d'erreur si INTER n'a pas de valeur.

Symbol-Function

Contient la valeur associée au symbole considéré comme une fonction. Cette valeur peut être une adresse machine ou une S-expression si la fonction n'est pas encore compilée.
L'accès à cette propriété se fait par la fonction SYMBOL-FUNCTION.

Exemple: (SYMBOL-FUNCTION 'INTER) --> le texte correspondant à la fonction

Symbol-Name (print-name)

Contient l'adresse de la chaîne de caractères représentant le nom du symbole. La fonction SYMBOL-NAME retourne cette chaîne.

Exemple: (SYMBOL-NAME 'INTER) --> "INTER"

Symbol-Plist (properties-list)

Voir les fonctions agissant sur cette propriété dans le chapitre VI.

Symbol-Package

On distingue deux types de symboles : les symboles internes et les symboles non internes. Un symbole interne est indexé par son nom dans un catalogue appelé "package", alors qu'un symbole non interne n'appartient à aucun catalogue particulier et est imprimé comme #: suivi du print-name. Par défaut, le nom du package courant est dans la variable Common-Lisp-User.

Notation des packages: SYSTEM::DEFINITION est le symbole DEFINITION du package SYSTEM.

Exemple :

```
(SETQ FOO 250)
(DEFUN FOO (X)  (IF  (ATOM X)  X  (CAR X)))
```

En mémoire, on aura la configuration suivante :

```
Pointeur symbole  --> Symbol-Value     --> 250
                      Symbol-Function  --> #<CLOSURE FOO (A B ...
                      Symbol-Name      --> "FOO"
                      Symbol-Plist     --> (SYSTEM::DEFINITON (DEFUN FOO))
                      Symbol-Package   --> #<PACKAGE USER>
```

Certains symboles sont déjà connus à l'appel du système:

- les constantes symboliques (qui contiennent leur propre valeur en valeur) : T, NIL,
- les fonctions prédéfinies,
- les variables système; ex: *standard-input* qui contient le nom du flux d'entrée standard des données.

A chaque symbole peut être associée une documentation sous forme d'une chaîne de caractères. Cette documentation, qui est fournie au moment de la définition d'un symbole est rendue disponible en ligne par la fonction DOCUMENTATION.

(DOCUMENTATION <symbole> <type-doc>)

<type-doc> sert à préciser la nature de l'objet dont on veut la documentation.

- *variable* correspond aux constructions "defvar", "defparameter" et "defconstant"
- *function* correspond à "defun" et "defmacro"
- *structure* correspond à "defstruct"
- *type* correspond à "deftype" et "defclass"
- *setf* correspond à "defsetf"

IV.3 - La boucle du top-level

Normalement, on interagit avec Lisp au moyen d'une boucle top-level "read-eval-print". Cette boucle est ainsi nommée car elle se situe au plus haut niveau de contrôle et elle consiste en une boucle sans fin qui lit une expression, l'évalue et affiche le résultat. On ne modifie l'état du système Lisp qu'en invoquant des actions provoquant des effets de bord.

Chaque implémentation de CommonLisp peut présenter des différences entre les interfaces utilisateur: caractère d'invite, présentation du résultat et des résultats multiples, éditeur de ligne, etc.

V - LES FONCTIONS DE MODIFICATION PHYSIQUE

Toutes les fonctions décrites dans cette section doivent être utilisées avec précaution pour éviter de placer le système dans un état de confusion, car elles permettent de modifier physiquement les structures LISP. En particulier il faut prendre garde à la modification physique des listes partagées, c'est-à-dire des listes auxquelles on peut accéder par plusieurs "pointeurs".

Toutefois la possibilité d'une véritable "chirurgie" sur les représentations internes des listes confèrent à LISP la puissance des langages machines.

a) (RPLACA <l> <s>)
(Etymologie : REPLAce CAr)

Remplace physiquement le CAR de la valeur de son premier argument (qui doit donc être une liste) par la valeur de son second argument et retourne la liste ainsi modifiée.

Exemple :

```
?(SETQ  L  '(A (B C) D))
=(A (B C) D)
?(RPLACA  L  (CADR L))
=((B C) (B C) D)
?L
=((B C) (B C) D)
```

Attention à la modification des listes partagées

Exemples (suite):

```
?(RPLACA (CADR L) 'U)
=(U C)
?L
=((U C) (U C) D)

?(SETQ L '(A B C) LL L)
=(A B C)
?(RPLACA L 'D)
=(D B C)
?L
=(D B C)
?LL
=(D B C)
```

b) (RPLACD <l> <s>)
(Etymologie : REPLAce CDr)

Remplace physiquement le CDR de la valeur de son premier argument (qui doit donc être obligatoirement une liste) par la valeur de son second argument et retourne la liste ainsi modifiée.

Exemple :

```
?(SETQ L '(A B C) LL '(X Y Z))
=(X Y Z)
?(RPLACD L LL)
=(A X Y Z)
?L
=(A X Y Z)
```

Les mêmes précautions que celles prises pour RPLACA doivent être observées dans la modification par RPLACD des listes partagées.

c) (SETF <place1> <s1> ... <placeN> <sN>) [spéciale]

Cette fonction que nous avons vue au §III.1 permet également de faire ces modifications physiques.

Exemples:

```
?(SETF  L '(A B C)  LL '(X Y Z))
=(X Y Z)
?(SETF  (CAR L)  '4  (CDR LL)  '(5 6))
=(5 6)
?L
(4 B C)
?LL
(X 5 6)
```

d) (NCONC &rest <li>)

 ayant pour valeur une liste <Li>, NCONC concatène physiquement les <Li> en plaçant dans le CDR du dernier doublet de <Li> (se trouvant au premier niveau) un pointeur sur <Li+1>. NCONC retourne la liste ainsi modifiée.

Exemple :

```
?(SETF L1 '(A (B))  L2 '((C D))  L3 '(E F))
=(E F)
?(NCONC L1 L2 L3)
=(A (B) (C D) E F)
?L1
=(A (B) (C D) E F)
?L2
=((C D) E F)
?L3
=(E F)
```

Si L a pour valeur une liste <l>, (NCONC L L) retourne une liste circulaire en faisant pointer le dernier CDR de <l> vers <l>.

Exemple :

```
?(SETF L '(A B C)
=(A B C)
```

```
?(NCONC L L)
=(A B C A B C A B C ...  ;Faire CTRL/C pour interrompre l'édition, puis ABORT
```

e) (DELETE <item> <l>)

Supprime toutes les occurences de <item> dans la liste <l>. La valeur de <l> est modifiée physiquement. Attention au cas où la liste devient vide.

f) Exemple d'utilisation des fonctions de modification physique

La fonction INVERSE ci-dessous prend en argument une liste et retourne une copie inversée (au premier niveau) de cette liste.

```
(DEFUN INVERSE (L)
    (COND
       ((NULL (CDR L)) L)
       (T (APPEND (INVERSE (CDR L)) (LIST (CAR L))))))
```

Calculons le nombre M de doublets consommés par cette fonction lors de l'inversion d'une liste L de n éléments. APPEND consomme autant de doublets que renferme son premier argument au premier niveau (donc autant que (INVERSE (CDR L)) i.e. n-1) et LIST autant de doublets qu'il a d'arguments (donc 1). n doublets seront donc utilisés à la dernière étape et qui s'ajouteront à ceux des étapes antérieures; d'où :

M1 = 1 , Mn = n + Mn-1.

Ce qui donne Mn = n + (n-1) + (n-2) + ...+ 1 = n(n+1) / 2.

Une grande partie de ces doublets est consommée inutilement. La version ci-dessous de la fonction inverse est plus économique :

```
(DEFUN REVERSE (LISTE RES)
     (COND
        ((NULL LISTE) RES)
        (T (REVERSE (CDR LISTE) (CONS (CAR LISTE) RES)))))
```

Ici RES est une variable tampon qui doit prendre comme valeur NIL lors de l'appel de la fonction.

```
?(REVERSE  '((A B) C D)  ())
=(D C (A B))
```

La fonction REVERSE inverse une liste de n éléments en n étapes et à chaque étape elle utilise un seul doublet (par application de CONS). Le nombre total de doublets utilisés est donc n.

Il n'est pas difficile de se convaincre que si on désire garder la liste de départ intacte, la fonction REVERSE utilise le minimum possible de doublets.

La version ci-dessous de la fonction INVERSE n'utilise aucun doublet! Mais la liste de départ est perdue : ses doublets sont recyclés dans la construction de sa copie inversée.

```
(DEFUN NREVERSE (LISTE RES)
     (COND
        ((NULL LISTE) RES)
        (T (NREVERSE (CDR LISTE) (RPLACD LISTE RES)))))
```

f) Comment éviter les problèmes liés au partage des listes

Lorsque une liste est partagée et que l'on souhaite faire des modifications physiques sur cette liste pour des raisons d'efficacité, il faut la dupliquer au préalable et travailler sur cette copie.

Cependant, tous les problèmes ne sont pas résolus. En particulier les fonctions ci-dessous ne peuvent traiter les listes circulaires.

(COPY-LIST <l>)

Retourne une copie de la liste <l> au premier niveau.

(COPY-ALIST <l>)

Retourne une copie de la aliste <l>.

(COPY-TREE <s>)

Retourne une copie de <s> à tous les niveaux. Cette fonction copie n'importe quel type d'objet LISP.

VI - LES LISTES D'ASSOCIATIONS ET LES LISTES DE PROPRIETES

VI.1 - Introduction

Dans un tableau on accède à un élément par son rang. Dans une liste, les fonctions d'accès sont réalisées par une composition de CAR et de CDR. Dans une structure composée, on repère les composants par les noms des champs, ces noms étant figés lors de la déclaration de la structure.

Il est quelquefois intéressant de disposer d'une structure de données où l'accès à une composante se fait par un nom (ou clef) calculé ou lu. Par exemple, si l'on veut compter le nombre d'occurences des mots d'un texte, il sera commode de mémoriser ces nombres dans une structure où l'on peut avoir accès directement au mot dont on veut stocker le nombre d'occurences.

Une telle structure s'appelle table ou structure associative. Chaque élément de la structure est repéré par une clef (ou indicateur) à laquelle est associée une valeur.

Pour rendre aisée l'utilisation de cette structure de données, les fonctions suivantes devront être définies:

a) Une fonction CREE qui initialise une structure associative à partir d'une liste d'indicateurs et d'une liste de valeurs.

b) Une fonction ACCES1 qui, pour un indicateur IND, retourne la valeur associée.

c) Une fonction ACCES2 qui, pour une valeur VAL, retourne le couple (IND VAL).

d) Une fonction AJOUT qui ajoute dans la structure un couple (IND VAL) d'indicateur-valeur.

e) Une fonction MISE-A-JOUR qui, dans la structure, change la valeur d'un indicateur IND par une nouvelle valeur NVAL.

f) Une fonction DESTRUCTION qui enlève un indicateur IND et sa valeur de la structure.

En CommonLisp, il existe 3 façons de réaliser la structure de données dont on vient de définir les fonctionnalités : les A-listes (ou listes d'association), les P-listes (ou listes de propriétés) et les tables hachées. Leurs représentations internes sont totalement différentes et les fonctions disponibles sur ces structures présentent des particularités et des restrictions par rapport aux définitions ci-dessus.

VI.2 - Les A-Listes (ou listes d'association)

En Lisp, les A-listes sont des listes qui possèdent la structure suivante:
((clef1 . val1) (clef2 . val2) ... (clefN . valN))

Tous les éléments de cette liste particulière doivent être des listes; le CAR de ces listes est la clef, et le CDR est la valeur. Le point placé entre la clef et la valeur n'apparait que si la valeur n'est pas une liste. La clef peut être soit un atome (symbole ou nombre), soit une S-expression quelconque; cependant les fonctions d'accès ne sont pas les mêmes dans les deux cas (prédicat EQ dans le cas d'une clef atome et prédicat EQUAL dans l'autre cas).

Les A-listes peuvent être utilisées comme les listes ordinaires, en particulier on peut les affecter à un symbole par les fonctions SET et SETQ. Dans ce cas, la A-liste sera la "Symbol-Value" du symbole.

Les principales fonctions qui permettent la manipulation des A-Listes sont les suivantes:

a) (ACONS <clé> <donnée> <al>)

Rajoute en tête un élément à la A-liste <al>. L'élément est composé de la clef <clé> et de la valeur <donnée>. Et retourne la nouvelle A-liste.

```
? (ACONS 'a 10 '((b . 11) (c 2 3)))
= ((a . 10) (b . 11) (c 2 3))
?(SETQ l (ACONS 'a 10 '((b . 11) (c 2 3)))
= ((a . 10) (b . 11) (c 2 3))
?l
= ((a . 10) (b . 11) (c 2 3))
```

b) (ASSOC <item> <al> &key :test :test-not :key)

Retourne le premier élément de la A-liste <al> dont la clef est <item> en utilisant par défaut le prédicat EQ, ou le prédicat précisé avec le mot-clé :test; s'il n'existe pas un tel élément la fonction ramène NIL.

```
?(ASSOC 'a '((b . 1) (a . 10)))
=(a . 10)
?(ASSOC '(a b) '((b . 1) ((a b) . 5) (a . z)))
=NIL

?(ASSOC '(a b) '((b . 1) ((a b) . 5) (a . z)) :test 'EQUAL)
= ((A B) . 5)
```

c) (RASSOC <s> <al> &key :test :test-not :key)

Retourne le premier élément de la A-liste <al> dont la valeur est égale à <s>.

```
?(RASSOC 'c  '((a . b) (a . c) (b . c)))
=(A . C)
?(RASSOC '(b c)  '((a . (b c))))
=NIL
?(RASSOC '(b c)  '((a . (b c)))  :test 'EQUAL)
=(A B C)
```

d) (SUBLIS <al> <s>)

Retourne une copie de l'expression <s> dans laquelle toutes les occurences 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.

```
?(SUBLIS '((a . z) (b 2 3))  '(a (b a c) d b . b)))
=(z  ((2  3)  z  c)  d  (2  3)  2  3 )
```

VI.3 - Les P-Listes

La structure citée ci-dessus peut également être réalisée en LISP par une P-Liste (ou liste de propriétés). Une P-Liste est une liste de la forme (<ind1> <val1> ... <indN> <valN)> où <indi> est un indicateur de la P-Liste et <vali> la valeur dans la P-Liste de <indi>. Les <indi> sont des atomes (symboles ou nombres) et les <vali> des S-expressions. La recherche d'un indicateur dans une P-Liste utilise le prédicat EQ.

La P-Liste elle-même est "stockée" dans le champ "Symbol-Plist" d'un symbole donné. L'accès à une P-liste fera intervenir systématiquement le nom de ce symbole. Le champ "Symbol-Plist" contient à la création du symbole l'adresse de la liste vide. Une fois la P-Liste créée, il contiendra l'adresse de la liste correspondante.

Les fonctions qui permettent la manipulation des P-Listes sont les suivantes :

a) (SYMBOL-PLIST <pliste>)

<pliste> a pour valeur un symbole S. SYMBOL-PLIST retourne la valeur de la P-LIST de S Cette fonction est aussi utilisée pour désigner une "place" pour SETF.

```
?(SETF (SYMBOL-PLIST 'CAPITALES)  '(FRANCE PARIS ITALIE ROME))
=(FRANCE PARIS ITALIE ROME)
?(SYMBOL-PLIST 'CAPITALES)
=(FRANCE PARIS ITALIE ROME)
?(GETF (SYMBOL-PLIST 'CAPITALES) 'FRANCE)
=PARIS
```

b) (GET <pliste> <ind>)

Retourne la valeur dans la P-Liste de <pliste> de l'indicateur <ind>. Si ce dernier n'est pas un indicateur de <pliste>, GET retourne NIL.
<pliste> doit avoir pour valeur un symbole.

```
?(GET 'CAPITALES 'FRANCE)
=PARIS
?(GET 'CAPITALES 'SUISSE)
=NIL
?(SETF (GET 'CAPITALES 'ESPAGNE) 'MADRID)
=MADRID
?(SYMBOL-PLIST 'CAPITALES)
=(ESPAGNE MADRID FRANCE PARIS ITALIE ROME)
```

c) (GETF <place> <ind>)

Cette fonction est identique à GET, mais elle est plus générale car elle permet d'utiliser une <place> pour indiquer sur quelle P-liste on veut faire la recherche.

```
?(GETF (SYMBOL-PLIST 'CAPITALES) 'FRANCE)
=PARIS
```

d) (GET-PROPERTIES <place> <liste-ind>)

Cette fonction est identique à GETF, mais la recherche dans la P-liste utilise la liste d'indicateurs au lieu du seul indicateur. Cette fonction retourne trois valeurs: le premier indicateur trouvé dans la P-liste, la valeur, la liste de propriétés à partir de l'indicateur.

e) (REMPROP <pl> <ind>)

Enlève de la P-Liste de <pl> l'indicateur <ind> et sa valeur (dans la P-Liste) et retourne la sous-liste de l'ancienne liste des indicateurs-valeurs de la P-Liste de <pl> commençant par <ind>.

```
?(REMPROP 'CAPITALES 'FRANCE)
=(FRANCE LYON ITALIE ROME)
?(GET 'CAPITALES 'FRANCE)
=NIL
```

f) (REMF <place> <ind>)

Idem REMPROP, mais en fournissant une <place> pour accéder à la P-Liste.

Remarques

1) SETF, REMPROP, REMF agissent par effet de bord. On peut utiliser aussi PUSH.

2) Un indicateur ne peut avoir qu'une valeur dans une P-Liste.

3) Un symbole qui reçoit une P-Liste la garde jusqu'à sa modification explicite par l'utilisateur. Aucun mécanisme de liaison par valeur à l'entrée d'une fonction n'agit sur le champ SYMBOL-PLIST d'un symbole.

Exemple :

```
?(DEFUN ASS (X)
   (SETF (SYMBOL-PLIST X) '(UN 1)))
=ASS
?(SYMBOL-PLIST 'ALPHA)
=NIL
?(ASS 'ALPHA)
=(UN 1)
?(SYMBOL-PLIST 'ALPHA)
=(UN 1)
```

g) Un exemple d'utilisation des P-listes : les mémo-fonctions

Une mémo-fonction est une fonction qui mémorise les valeurs qu'elle calcule et consulte sa mémoire avant de se lancer dans le calcul effectif d'une valeur.

Exemple : la suite de FIBONNACI.

La suite FIBO(n) est définie par:
FIBO(n) = FIBO(n-1) + FIBO(n-2).
On peut l'écrire en LISP par la fonction suivante :

```
(DEFUN FIBO (N)
    (COND
        ((= N 0) 0)
        ((= N 1) 1)
        (T (+ (FIBO (1- N)) (FIBO (- N 2))))))
```

Rien que dans le calcul de (FIBO 5), cette fonction calcule trois fois (FIBO 2) et deux fois (FIBO 3). Alors que si les valeurs calculées étaient mémorisées, ces répétitions inutiles n'auraient pas lieu. Voici une fonction MEM-FIB qui mémorise ses résultats calculés.

```
(DEFUN MEM-FIB (N)
   (COND
     ((= N 0) 0)   ;Retourner 0 pour 0.
     ((= N 1) 1)   ;et 1 pour 1.
     ((GET 'MEM-FIB N))  ;si la valeur recherchée
                         ;se trouve en mémoire,
                         ;la retourner.
     (T (SETF
          (GET 'MEM-FIB N)     ;sinon, la calculer et
```

```
            (+ (MEM-FIB (1- N)) ;la mémoriser.
               (MEM-FIB (- N 2))))))
    ))
```

VII - LES FONCTIONS DE TYPE MACRO

Un autre type de fonction reconnu par Common Lisp est le type MACRO. L'originalité des MACRO-fonctions réside principalement dans la manière dont la dernière forme présente dans le corps de ces fonctions est évaluée. Cette forme subit deux évaluations : une fois dans l'environnement local de la fonction (i.e. quand les paramètres formels et les variables locales sont encore liés à leurs valeurs) et une deuxième fois hors de l'environnement local, après l'étape de la déliaison. C'est cette deuxième valeur qui est retournée comme le résultat de l'évaluation de la MACRO-fonction. Les MACRO ne sont pas réellement des fonctions et ne peuvent pas être arguments de fonctions telles que APPLY, FUNCALL et MAP.

VII.1 - La définition des macro-fonctions

La définition des fonctions de type MACRO se fait par la fonction DEFMACRO. La syntaxe de la définition est :

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

où <symb> est un symbole, <lvar> est une S-expression formée de variables et <s1>, ..., <sN> sont des formes ou atomes évaluables.

La fonction DEFMACRO est "spéciale" et retourne <symb> en valeur. Après l'évaluation de la forme ci-dessus, <symb> acquiert une valeur fonctionnelle de type MACRO. L'appel se fait par la forme (<symb> <arg1> ... <argk>). Les étapes ci-dessous se déroulent alors dans l'ordre.

1) <lvar> et la liste (<arg1> ... <argk>) sont mis en liaison

2) Macro-expansion : <s1>, ..., <sN> sont évalués et la valeur de <sN> mémorisée.

3) Macro-évaluation : la valeur <sN> est à nouveau évaluée et retournée.

On constate que la valeur retournée n'est pas la valeur de la dernière forme ou atome présent dans le corps de la fonction, mais la valeur d'une seconde évaluation de cette forme opérée dans l'environnement de la fonction appelante.

VII.2 - Quelques exemples

Le mécanisme de la double évaluation de la dernière forme présente dans le corps d'une MACRO-fonction change radicalement l'esprit dans lequel ces fonctions doivent être abordées. On ne cherchera pas à retourner immédiatement une valeur, mais à construire une forme qui, évaluée, retourne cette valeur. Il faudra donc avoir présente à l'esprit l'expansion de la MACRO.

Exemples :

a)

```
?(DEFMACRO MCAR (&rest L)      ; appel par (MCAR <L>)
                               ; On prépare
     (CONS 'CAR L))            ; l'expansion (CAR <L>)
=MCAR
?(MCAR '(a b c))               ; L <-- '(a b c)
=a

?(SYMBOL-FUNCTION 'MCAR)
=(SYSTEM::MACRO . #<CLOSURE ... >)
```

b) On sait que la forme (LET ((VAR1 VAL1) ... (VARn VALn)) s1...sp) est équivalente à ((LAMBDA (VAR1 ... VARn) s1 ...sp) VAL1 ...VALn). On pourra donc essayer d'écrire une MACRO-fonction qui transforme dans son étape d'expansion la première forme en seconde.

```
?(DEFMACRO MLET (LV &rest CORPS)
                            ;LV ((VAR1 VAL1)...(VARn VALn))
                            ;CORPS (s1 ...sp)
     (CONS
        (CONS 'LAMBDA
              (CONS (MAPCAR 'CAR LV) CORPS))
        (MAPCAR 'CADR LV))
   )
=MLET
?(MLET ((A 2) (B 4)) (PRINT (+ A B)) (PRINT (- A B)))
6
-2
=-2
```

c) L'une des raisons d'être des MACRO-fonctions est l'écriture des fonctions de contrôle conditionnnelles. Ecrivons à titre d'exemple la fonction OR en MACRO. L'appel se faisant par (OR s1...sn), l'expansion devra être:

```
(COND (s1)
      (t (OR s2 ...sn)))
```

```
?(DEFMACRO MAC-OR (&rest CORPS)
     (IF CORPS
         (LIST 'COND
               (LIST (CAR CORPS))
               (LIST 'T (CONS 'MAC-OR (CDR CORPS))))))
=MAC-OR
```

d) Dans le même ordre d'idée, voici la fonction WHILE. L'appel se fait par (WHILE s s1 ... sn). L'expansion sera:

(IF s (PROGN s1 ... sn (WHILE s s1 ... sn)))

```
?(DEFMACRO MWHILE (&rest CORPS)
    (LIST 'IF
          (CAR CORPS)
          (APPEND (CONS 'PROGN (CDR CORPS))
                  (LIST (CONS 'MWHILE CORPS))))
 )
=MWHILE
?(SETQ A 3)
=3
?(MWHILE (> A 0) (PRINT A) (SETQ A (1- A)))
3
2
1
=NIL
```

VII.3 - Le phénomène de capture et les macro-fonctions

Les macro-fonctions servent à construire dynamiquement (c'est-à-dire à l'aide d'un programme, à partir de données) des corps de fonctions, puis à les évaluer. Malencontreusement, il se produit parfois des erreurs dûes aux homonymies entre noms des arguments de la fonction et noms des objets que l'on veut manipuler. Nous allons voir que le mécanisme de double évaluation résoud en partie le masquage des arguments. Etudions ce phénomène sur un exemple.

La fonction UNLESS est l'une des diverses formes de la structure si-alors-sinon. (UNLESS s1 s2 ... sn) retourne nil si la valeur de s1 est différente de nil, sinon elle évalue dans l'ordre s2 ... sn et retourne la valeur de sn. On peut remarquer que UNLESS est équivalent à:

(IF (NOT s1) (PROGN s2 ...)) et l'écrire à l'aide d'une MACRO par:

```
?(DEFMACRO M-UNLESS (TEST &rest CORPS)
   (LIST 'IF (LIST 'NOT TEST) (CONS 'PROGN CORPS)))
=M-UNLESS

?(SETQ TEST NIL)
=NIL
?(M-UNLESS TEST 1 2 3)
=3
```

L'évaluation de la forme (IF (NOT TEST) (PROGN 1 2 3)) (l'expansion de la macro-fonction) ne se fait pas dans l'environnement de la fonction, mais en dehors de cet environnement, quand la variable TEST a retrouvé sa valeur initiale (i.e. NIL).

Cependant, certaines formes de masquages ou d'homonymies peuvent apparaître. Notamment quand la MACR0-expansion construit des formes où interviennent des variables locales. Voici quelques exemples que le lecteur est invité à analyser.

```
?(DEFMACRO EXCHANGE (VAR1 VAR2)      ;échange la valeur de 2
     (LET ((R ())) ;variables
          (LIST 'SETQ 'R VAR1  ;appel par (EXCHANGE X1 X2)
                VAR1 VAR2     ;et expansion par
                        ;(SETQ R X1 X1 X2 X2 R)
                  VAR2 'R)))
=EXCHANGE

?(SETQ R 1 S 2)
=2

?(EXCHANGE R S)
=2                ;expansion par (SETQ R R R S S R)
?R      ?S
=2      =2
```

Autre exemple où une variable locale masque une variable globale:

```
?(DEFMACRO MOYENNE (&rest LNBR)
 ;calcule la moyenne de n nombres
 ;appel par (MOYENNE n1 ... nk)
 ;expansion par :
 ; (LET ((N (LENGHT '(n1 ... nk)))
 ;       (/ (+ n1 ... nk) N))
    (LIST 'LET
          (LIST (LIST 'N (LIST 'LENGHT (LIST 'QUOTE LNBR))))
          (LIST '/
                (CONS '+ LNBR)
                'N)))
=MOYENNE
?(SETQ N 10)
=10
?(MOYENNE N 3 5)
=11/3                        ; (3+3+5) / 3
```

En fait, on constate que l'expansion de (MOYENNE N 3 5) donne:

```
(LET ((N (LENGHT '(N 3 5))))
     (DIVIDE (+ N 3 5) N))
```

On aurait pu éviter l'emploi d'une variable locale intervenant dans l'expansion et écrire par exemple :

```
?(DEFMACRO MOYENNE (&rest LNBR)
    (LET ((N (LENGHT LNBR)))
         (LIST '/
               (CONS '+ LNBR)
               N)))
=MOYENNE
?(MOYENNE N 3 5)
=6                              ; (10+3+5)/3
```

De même, pour la fonction EXCHANGE :

```
?(DEFMACRO EXCHANGE (VAR1 VAR2)
     (LIST 'SETQ VAR1 VAR2 VAR2 (LIST 'QUOTE (EVAL VAR1))))
=EXCHANGE
?(SETQ S 'A)
=A
?(SETQ R 'B)
=B
```

L'expansion de la forme d'appel (EXCHANGE S R) donne alors:
(SETQ S R R 'A)

VII.4 - Les fonctions d'aide à la construction de listes complexes

Afin d'éviter les successions de CONS, LIST et APPEND dans la construction de listes et en particulier dans l'écriture des MACRO-fonctions, LISP met à notre disposition des MACRO-caractères qui rendent cette construction plus aisée et plus lisible. Un MACRO-caractère est un caractère qui a une valeur fonctionnelle. A la lecture de ce caractère, la valeur retournée par la fonction associée à ce MACRO-caractère remplace le MACRO-caractère lu. Nous connaissons déjà le MACRO-caractère QUOTE. Nous allons voir les MACRO-caractères '`' (BACKQUOTE), ',' (virgule) et '@' (arrobas). Ces MACRO-caractères n'ont pas d'arguments. La transformation provoquée par leur valeur fonctionnelle s'opère sur la première S-expression qui les succède dans le flux d'entrée.

a) Utilisation du caractère "backquote":

`(x1 ... xp) est lu comme '(x1 ... xp)

b) Utilisation du caractère "virgule" associé au caractère "backquote":
Si xi1, ..., xip ont une valeur, à la lecture de `(x1 ... ,xi1 ... ,xip ...xn) les xij sont remplacés par leur valeur.

Exemples :

```
?(SETQ X '(a b c) Y '(1 2 3))
=(1 2 3)

?`(X ,X ,Y Y)
=(X (a b c) (1 2 3) Y)
?`((CAR ,X) ,(CDR X) (Y ,(CDR Y)))
=((CAR (a b c)) (b c) (Y (2 3)))
```

c) Utilisation du caractère "arrobas" associé aux caractères "backquote" et "virgule": Si X a pour valeur une liste, dans `(.. ,@X ...) ,@X est remplacé par la suite des éléments de X.

Exemples :

```
?(SETQ X '(a b c) Y '(1 2 3))
=(1 2 3)
?`(,X ,@Y)
=((a b c) 1 2 3)
?`(,@(CDR X) (,(CAR X) ,@Y) fin)
=(b c (a 1 2 3) fin)
```

```
?`((LAMBDA ,X (APPLY 'LIST (QUOTE ,X))) ,@Y)
=((LAMBDA (a b c) (APPLY 'LIST '(a b c))) 1 2 3)
```

Voici quelques exemples de fonctions de type MACRO écrites à l'aide des MACRO-caractères étudiés ci dessus :

```
?(DEFMACRO MLET (LV &rest CORPS)
   `((LAMBDA ,(MAPCAR 'CAR LV) ,@CORPS) ,@(MAPCAR 'CADR LV)))


?(DEFMACRO EXCH (VAR1 VAR2)
     `(SETQ ,VAR1 ,VAR2 ,VAR2 (QUOTE ,(EVAL VAR1))))


?(DEFMACRO MAC_OR (&rest CORPS)
     (IF CORPS
         `(COND (,(CAR CORPS))
         (T (MAC_OR ,@(CDR CORPS))))))


?(DEFMACRO MSETQ (&rest CORPS)
     ;appel par (MSETQ X1 VAL1 ... Xn VALn)
     ;CORPS prend la valeur (X1 VAL1 ...Xn VALn)
     ;expansion par :
     ;           (PROGN (SETQ (QUOTE X1) VAL1)
     ;                  (MSETQ X2 VAL2 ... Xn VALn))
     (COND
       ((NULL CORPS) NIL)      ;MSETQ retourne toujours NIL
       ((NULL (CDR CORPS)) 'ERREUR)
                    ;un nombre impair d'éléments dans CORPS
       (T `(PROGN (SETQ (QUOTE ,(CAR CORPS))
                        ,(CADR CORPS))
                  (MSETQ ,@(CDDR CORPS))))))
```

VII.5 - La mise au point des macro-fonctions

a) (MACROEXPAND-1 <s>)

<s> a pour valeur une forme dont le CAR est une fonction de type MACRO. MACROEXPAND-1 retourne l'expansion de cette forme. (Cette fonction ne retourne pas cette valeur sur l'IBM).

Exemples :

```
?(MACROEXPAND-1 '(MAC_OR (CDDR L) (CADR L) (CAR L)))
=(COND ((CDDR L)) (T (MAC_OR (CADR L) (CAR L))))

?(MACROEXPAND-1 '(LET ((X '(a b c))) (PRINT (CAR X))))
=(LAMBDA (X) (PRINT (CAR X))) '(a b c))
```

b) (MACROEXPAND <s>)

Permet d'expanser dans la forme qui est la valeur de <s> tous les appels de MACRO à tous les niveaux. (Même remarque que pour MACROEXPAND-1)

VIII - LES SYMBOLES, LES CHAÎNES DE CARACTERES, LES VECTEURS, LES SéQUENCES, LES TABLES DE HACHAGE, LES STRUCTURES

Outre les fonctions de manipulation des nombres et des listes, Common Lisp propose des fonctions de gestion des symboles, de manipulation des chaînes de caractères et des vecteurs, et de création de structures. Ne sont présentées dans ce paragraphe que les principales fonctions appartenant à ces catégories.

Notons que CommonLisp propose aussi des fonctions s'appliquant à des séquences (une séquence est une liste ou un vecteur), des tables de hachage, des "readtables" et des "streams". Pour une information sur ces fonctions, voir le livre de G.L STEELE, cité dans la bibliographie.

VIII.1 - Les fonctions de gestion des symboles

(MAKE-SYMBOL <print-name>)

```
?(make-symbol "y")
=#:|y|
?=(make-symbol (concatenate 'string "a" "b")))
=#:|ab|
```

(GENSYM &optional <x>)

retourne à chaque appel un nouveau symbole créé par concaténation de "g" ou d'un paramètre optionnel <x> et de la valeur de la variable système *gensym-counter*

```
? (gensym)
= g101
```

(GENTEMP &optional <prefixe-string> <package>)

retourne un nouveau symbole dans le package <package>.

```
? (gentemp (format nil "~s" (setq b 'a)))
= a0
? (gentemp)
= T1
```

Le compteur associé à gentemp s'incrémente de 1 à chaque appel.

Pour créer des symboles, voir aussi la fonction READ-FROM-STRING.

VIII.2 - Les fonctions de manipulation des chaînes de caractères

Une chaîne de caratères est une collection de caractères accessibles par leur index (numéro). L'index du premier caractère d'une chaîne est égal à 0. La représentation externe d'une chaîne est:

```
"xxxxxxxxxxxxx"
```

Les fonctions de base sont : LENGTH, CHAR, AREF, ELT.

La fonction de conversion des chaînes est: STRING.

Les fonctions de comparaison des chaînes sont: STRING=, STRING< , ...STRING/=
(STRING= <ch1> <ch2> &key :start1 :end1 :start2 :end2)

Les fonctions de création des chaînes sont: CONCATENATE, MAKE-STRING, SUBSTRING.

D'autres fonctions sont disponibles pour accéder aux caractères d'une chaîne, pour modifier physiquement une chaîne ou pour rechercher des sous-chaînes.

Exemples:

```
?(CHAR "Floor and" 0)
=#\F
?(LENGTH "foo")
=3
?(MAKE-STRING 3 :initial-element '#\s)
="sss"
?(CONCATENATE 'STRING "a" "bc")
="abc"
?(STRING 'A)
="A"
```

VIII.3 - Les fonctions de manipulation des vecteurs et des tableaux

Un vecteur est une collection d'objets Lisp accessibles par leur index ou numéro. Ces objets Lisp sont des S-expressions pouvant avoir des types différents. Les index des vecteurs débutent à la valeur 0.

La représentation externe d'un vecteur est:

#(<sexpr0> <sexpr1> ... <sexprN>)

et d'un tableau à 2 dimensions est:

#2A((<sexpr> ... <sexpr>) ... (<sexpr> ... <sexpr>))

La dimension d'un tableau peut être quelconque.

(MAKE-ARRAY <dimensions> &key :element-type :initial-element)

fabrique un nouveau tableau de <n> éléments. Chacun de ces éléments est initialisé à la valeur fournie après le mot-clé :initial-element.

```
? (MAKE-ARRAY  5)
= #(NIL NIL NIL NIL NIL)
? (MAKE-ARRAY  5  :type-element 'fix  :initial-element 0)
= #(0 0 0 0 0)

?(MAKE-ARRAY '(2 2))
=#2A((NIL NIL) (NIL NIL))
```

(VECTOR <s1> <s2> ... <sN>)

fabrique un vecteur de taille N, initialisé avec les valeurs des S-expressions <sI>.

```
?(VECTOR 1 2 3 4)
=#(1 2 3 4)
```

Les principales fonctions manipulant les vecteurs sont: AREF, VECTOR-PUSH, VECTOR-POP, ADJUST-ARRAY.

```
?(SETQ  A (VECTOR 1 2 3 4))
=#(1 2 3 4)
?(AREF A 1)     ; accès en lecture
=2
?(SETF (AREF A 1) 5)  ; accès en écriture
=5
?A
=#(1 5 3 4)
```

VIII.4 - Les séquences

Le type "Séquence" recouvre les types "liste" et "vecteur". Les fonctions décrites ci-dessous sont fondées sur la propriété commune à ces deux types: ce sont des ensembles ordonnés d'éléments. Outre les fonctions déjà vues, telles que ELT, LENGTH, CONCATENATE, voici quelques fonctions utiles:

(FIND <item> <séquence> :test <test> :key <clé>)

(SUBSEQ <séquence> <debut> &optional <fin>)

(MAP <type-résultat> <fonction> &rest <séquences>)

(SOME <prédicat> <séquence> &rest <séquences>)

```
? (some #'(lambda (x) (eql 'a x)) '(d v a c))
T
? (some #'(lambda (x) (if (eql 'a x) x)) '(d v a c))
A
? (some #'(lambda (x) (if (eql 'a (car x)) x)) '((d) (v b) (a e f) (c)))
(A E F)
```

(EVERY <prédicat> <séquence> &rest <séquences>)

```
? (every #'(lambda (x y) (equal x y)) '(a b) '(a b))
T
? (every #'(lambda (x y) (equal x y)) '(a b) '(a c))
NIL
```

(NOTANY <prédicat> <séquence> &rest <séquences>)

(NOTEVERY <prédicat> <séquence> &rest <séquences>)

(REMOVE <item> <séquence> &key :from-end :test :test-not :start :end :count :key)
Retourne une copie de la séquence modifiée.

(DELETE <item> <séquence> &key :from-end :test :test-not :start :end :count :key)
Modification physique de la séquence.

(SORT <séquence> <prédicat> &key :key)
Modification physique de la séquence selon le prédicat de comparaison de 2 éléments et le moyen d'accès à la donnée s'il s'agit d'une liste ou d'une structure quelconque.

```
? (sort '(("cc" 1) ("aa" 2) ("bb" 3)) #'string-lessp :key #'car)
(("aa" 2) ("bb" 3) ("cc" 1))
```

(MERGE <type-résultat> <séquence1> <séquence2> <prédicat> &key :key)
Retourne une copie de la fusion des séquences déja triées.

VIII.5 - Les tables de hachage

CommonLisp propose des fonctions spécialisées dans la création, la modification et l'accès aux tables de hachage;

(MAKE-HASH-TABLE &key :test :size :rehash-size :rehash-threshold)
crée une table de hachage, dont on peut préciser le test d'accès aux éléments, la taille, etc.

(GETHASH <clé> <table de hachage> &optional <valeur par défaut>)
permet d'accéder à la valeur associée à la clé. Par conséquent permet de modifier la table par le biais de la fonction SETF.

Exemple: (SETF (GETHASH 'X table) 3) associe la valeur 3 à la clé X.

(REMHASH <clé> <table de hachage>)
supprime un élément de la table.

Autres fonctions: MAPHASH, CLRHASH

VIII.6 - Les structures

Ce sont des entités analogues aux records en C ou aux objets en C++. Common Lisp permet en plus de créer une sorte de hiérarchie de types de structures.

1) Définition de nouveaux types d'objets structurés

Ils sont formés d'un certain nombre de champs nommés et typés ou non, pouvant avoir pour valeur un objet-LISP quelconque.

La consultation et la modification des champs se font au moyen de fonctions d'accès aux champs définies automatiquement 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.

(DEFSTRUCT <nom-struct> <documentation> <desc-champ 1> ... <desc-champ N>)
[MACRO]

<nom-struct> est un symbole devenant le nom de la structure et qui est un nouveau type d'objet-LISP, ou une liste comportant le nom de la structure et des options

<desc-champ i> est de la forme:

(<nom-champ> <valeur-par-défaut> <option1> <valeur1> ... <optionK> <valeurK>)

Par exemple, la définition:

```
(defstruct  homme
            âge
            (poids 75))
= homme
```

crée le type homme
les fonctions d'accès aux champs
homme-âge
et homme-poids
et la fonction de création des instances:
make-homme

Extension de structure

```
(defstruct (chercheur (:include homme))
           projet
           articles)
=chercheur
```

Cette structure a 4 champs:
âge
poids
projet
article

2) Création d'instances

(MAKE-<nom-struct>)

```
?(SETQ paul (MAKE-HOMME))
=#S(HOMME  :AGE NIL  :POIDS 75)
?(SETQ jean (MAKE-CHERCHEUR))
=#S(CHERCHEUR  :AGE NIL  :POIDS 75  :PROJET NIL  :ARTICLE NIL)
```

3) Accès aux champs

(<nom-struct>-<champ> <o>) ; <o> est le nom d'une instance

```
 ?(HOMME-AGE paul)                    ; accès en lecture
 =NIL
 ?(SETF (HOMME-POIDS paul) 80)        ; accès en écriture
 =80
```

4) Tests de type

(TYPE-OF <s>) calcule le type de <s>

```
  ? (TYPE-OF paul)
  = HOMME
```

(SUBTYPEP <type1> <type2>)
avec <type1> et <type2> représentant des types
est vrai si <type1> est sous-type de <type2>

Exemples:

```
   ?(SUBTYPEP 'CHERCHEUR 'HOMME)
   = T
   ?(SUBTYPEP 'SYMBOL 'SYMBOL)
   = T
   ?(SUBTYPEP 'INTEGER 'NUMBER)
   = T
```

(TYPEP <s> <type>)
est vrai si le type de <s> est un sous type du type <type>

Exemples:

```
   ? (TYPEP (MAKE-HOMME) 'HOMME)
   = T
   ? (TYPEP paul 'homme)
   = T
```

IX - LES TYPES: création, conversion, vérification

En Common Lisp, il existe une liste de types prédéfinis (cf la table ci-dessus) à laquelle on peut adjoindre de nouveaux types définis à l'aide de spécifieurs. Grâce à ces spécifieurs, il est possible de raffiner la description de la valeur d'un objet, ce qui est très pratique pour effectuer des vérifications automatiques de types et ajouter des informations sémantiques sur une entité.

Table des symboles représentant les types prédéfinis:

array	fixnum	package	simple-string
atom	float	pathname	simple-vector
bignum	function	random-state	single-float
bit	hash-table	ratio	standard-char
bit-vector	integer	rational	stream

character	keyword	readtable	string
common*	list	sequence	string-char*
compiled-function	long-float	short-float	symbol
complex	nil	signed-byte	t
cons	null	simple-array	unsigned-byte
double-float	number	simple-bit-vector	vector

Remarque: Les types notés par des symboles suivis d'un astérisque ne sont pas définis dans la norme X3J13.

a) Spécifieurs de types sous forme de symbole

Les symboles des types prédéfinis sont énumérés dans la table ci-dessus.
En outre, chaque fois qu'un type structure est défini par DEFSTRUCT, le nom du type de la structure devient un symbole valide représentant ce type.

b) Spécifieurs de types sous forme de liste

Une liste peut représenter la spécification d'un type. Dans ce cas, le "car" de la liste est un symbole, et le reste de la liste est une information subsidiaire qui peut être non spécifiée. Les items de cette liste qui ne sont pas spécifiés sont notés par *.

Par exemple, pour définir complètement un type vecteur, il faut indiquer le type des éléments et la longueur du vecteur. Cependant il peut être possible de n'indiquer que l'une de ces informations.

Exemples:
(vector double-float 100) décrit un vecteur de 100 doubles flottants

(vector double-float *) décrit un vecteur de doubles flottants
ce qui peut être aussi noté:
(vector double-float)

(vector * 100) décrit un vecteur de 100 éléments

(vector double-float 100) est un sous-type de (vector double-float).

c) Spécifieurs de types sous forme de prédicat

Une liste de la forme (SATISFIES <nom-prédicat>) fait référence à un ensemble d'objets vérifiant le prédicat <nom-prédicat> défini par une fonction globale à un argument. Les prédicats sous forme de lambda-expression ne sont pas autorisés.

Exemple:

Le type (satisfies numberp) est identique au type *number*

d) Spécifieurs de types qui combinent des types

- (MEMBER <objet1> <objet2> ...) fait référence à un ensemble d'objets dont les noms appartiennent à la liste <objet1> <objet2> ...

- (EQL <objet>) est identique à (MEMBER <objet>) , mais peut être utilisée en tant que spécialiseur de paramètre dans une méthode CLOS.

- (NOT <type>) fait référence à tous les objets qui ne sont pas du type <type>.

- (AND <type1> <type2> ...) décrit l'intersection des types <type1> <type2> ...

- (OR <type1> <type2> ...) décrit l'union des types <type1> <type2> ...

Exemples:

(and integer (satisfies primep)) permet d'éviter l'appel de la fonction primep sur un argument qui ne serait pas un entier. Les types sont vérifiés de gauche à droite.

(or null cons) est équivalent au type *list*.

e) Spécifieurs de types qui spécialisent des types

- (ARRAY <type-élément> <dimensions>) fait référence aux tableaux dont les éléments vérifient le type <type-élément> et dont les dimensions correspondent à <dimensions>.

- (SIMPLE-ARRAY <type-élément> <dimensions>) indique de plus que les objets de ce type sont des tableaux simples.

- (VECTOR <type-élément> <taille>) fait référence à des tableaux à une dimension.

- (SIMPLE-VECTOR <taille>) décrit des vecteurs généraux simples.

- (complex <type>) fait référence à des nombres complexes dont la partie réelle et la partie imaginaire sont du type <type>.

- (FUNCTION (<type-arg1> <type-arg2> ...) <type-valeur>) est utilisé uniquement lors de déclarations de fonctions. <type-valeur> peut être décrit à l'aide du spécifieur VALUES pour indiquer les types des valeurs multiples.

- (VALUES <type-val1> <type-val2> ...) est utilisé pour déclarer les types des valeurs multiples retournées par une fonction.

Exemples:

```
(array integer 3)              ; tableaux d'entiers de dimension 3
(array integer (* * *))  ; tableaux d'entiers de dimension 3
(array * (4 5 6))          ; tableaux de dimensions 4 x 5 x 6
(array character (3 *))  ; tableaux de caractères ayant 3 lignes
(vector (mod 32) *)            ; vecteurs d'entiers compris entre 0 et 31
```

f) Spécifieurs de types qui fournissent des abréviations de types

- (INTEGER <val-min> <val-max>) fait référence aux entiers compris entre <val-min> et <val-max>.

- (MOD <n>) fait référence aux entiers non négatifs inférieurs à <n>.

- (SIGNED-BYTE <s>) fait référence aux entiers pouvant être représentés avec <s> bits.

- (UNSIGNED-BYTE <s>) fait référence aux entiers non négatifs pouvant être représentés avec <s> bits.

- (RATIONAL <val-min> <val-max>) fait référence aux rationnels compris entre <val-min> et <val-max>.

- (FLOAT <val-min> <val-max>) fait référence aux réels compris entre <val-min> et <val-max>.

- (STRING <taille>) est identique à (ARRAY string-char (<taille>)).

- (BIT-VECTOR <taille>) est identique à (ARRAY bit (<taille>)).

g) Définition de nouveaux spécifieurs de types

Il existe deux façons de définir de nouveaux types: DEFSTRUCT et DEFTYPE. DEFSTRUCT est utilisé pour définir de nouveaux types sous la forme de structures, et DEFTYPE est utilisé pour définir de nouvelles abréviations de types.

(DEFTYPE <nom> <lambda-liste>
[[{<déclaration>}* | <chaîne-de-caractères-documentation>]]
{<forme>}*)

Exemples:

```
(deftype mod (n) `(integer 0 (,n)))
? (typep 10 '(mod 20))
T
? (typep 10 '(mod 5))
NIL
```

```
(deftype list () '(or null cons))

(deftype matrice-carree (&optional type taille)
        "MATRICE-CARREE inclut tous les tableaux carrés à 2 dimensions"
        `(array ,type (,taille ,taille)))

(matrice-carree short-float 7)     est équivalent à (array short-float (7 7))

Une autre définition des matrices carrées consiste en:

(defun equidimensionnel (a)
        (or     (< (array-rank a) 2)       "array-rank" fournit le nombre de
                        dimensions d'un tableau
                (apply #'= (array-dimensions a))))        "array-dimensions" retourne une
                                                liste des différentes dimensions des
                                                éléments du tableau

(deftype matrice-carree (&optional type taille)
        `(and (array ,type (,taille ,taille))
                (satisfies equidimensionnel)))
```

h) Fonctions de conversion de type

Pour convertir des données d'un type à un autre, il faut utiliser la fonction COERCE.

Exemples:

```
(coerce '(a b c)  'vector)          --->  #(a b c)
(coerce "a"  'character)  --->  #\a
(coerce 7/2  'float)                --->  3.5
```

i) Détermination et vérification du type d'un objet

La fonction TYPE-OF permet d'obtenir le type d'un objet. Le résultat peut dépendre de la version de Common Lisp.

La fonction CHECK-TYPE permet de vérifier si une donnée est bien d'un type particulier. Dans le cas contraire, un message d'erreur s'affiche et le programme s'interrompt.

(CHECK-TYPE <place> <spécifieur de type>) [MACRO]

Exemple:
(check-type cube1 cube) ne provoque pas d'erreur si cube1 vérifie le type cube, par exemple est instance de la classe cube. Cette fonction retourne NIL.

La fonction SUBTYPEP permet de vérifier qu'un type est un sous-type d'un autre type, ceci est généralisé aux classes.
(SUBTYPEP <type1> et <type2> est vrai si <type1> est sous-type de <type2>

La fonction TYPEP permet de vérifier que le type d'un objet est un sous-type du type <type>, ce qui se généralise aux classes pour vérifier que la classe d'une instance est une sous-classe d'une autre classe.

X - CLOS : Common Lisp Object System

Notations:
* indique 0 ou plusieurs occurrences
+ indique 1 ou plusieurs occurrences
{} indique un regroupement

Remarque: Avant d'utiliser les fonctions spécifiques de CLOS, évaluer l'expression
(use-package "CLOS").

X.1 - Définition de classes

(DEFCLASS <nom-classe> (<nom-de-superclasse>*)
(<description-slot>*)
(<option-classe>*))

La description d'un slot (ou champ) est de la forme (<nom-slot> <option-slot>*), où chaque option est un mot clé suivi d'un nom, d'une expression ou d'une autre forme.

Les options sont:

:READER <nom-fonction> : nom de la fonction pour l'accès en lecture au champ
:WRITER <nom-fonction> : nom de la fonction pour l'accès en écriture au champ
:ACCESSOR <nom-fonction> : nom de la fonction pour l'accès en lecture/écriture
:INITFORM <expression> : valeur initiale du champ par défaut
:INITARG <symbole> : symbole utilisé comme mot-clé dans la liste des arguments lors de la création d'une instance
:DOCUMENTATION <chaîne de caractères> : description sémantique du champ
:TYPE <spécification-type> : indique que le contenu du champ est toujours du type spécifié
:allocation <allocation-type> : spécifie le mode d'allocation du slot : ":class" ou ":instance" selon que la valeur est partagée par les instances de la classe ou non

Les options sont facultatives. Cependant il est souvent pratique de fournir le nom des fonctions d'accès au champ, ainsi que le nom du mot-clé permettant de fournir une valeur initiale.

L'option principale de description de classe est :DOCUMENTATION <string>. Comme toutes les options de documentation, elle permet d'avoir une information automatique en ligne à l'aide de la fonction DOCUMENTATION.

Ex: (DOCUMENTATION '<nom-classe> 'type)

DEFCLASS ressemble à DEFSTRUCT. La syntaxe est légèrement différente, et on dispose de plus de contrôle sur la façon de nommer les fonctions permettant de manipuler les champs.

- Par exemple, comparons une définition avec DEFSTRUCT et une autre avec DEFCLASS:

(DEFSTRUCT personne (nom 'paul) (age 10))

La création d'instance s'appelle automatiquement MAKE-PERSONNE et les fonctions d'accès aux champs sont PERSONNE-NOM et PERSONNE-AGE. L'initialisation des champs se fait ainsi au moment de la création des instances:

```
(make-personne  :nom  'pierre  :age 12)
```

- La définition avec DEFCLASS qui produirait le même effet serait:

```
(DEFCLASS personne ()
  ((nom   :accessor        personne-nom
                  :initform        'paul
                  :initarg :nom)

  (age            :accessor        personne-age
                  :initform        10
                  :initarg :age)
  ))
```

Il est possible de nommer les fonctions d'accès aux champs comme on le souhaite (par exemple "nom" et non "personne-nom"). Les conventions peuvent être plus sémantiques et moins rigides qu'avec DEFSTRUCT.

Il n'est pas nécessaire de fournir toutes les options de chaque champ lors de la définition d'une classe.

Notons que les classes elles-mêmes sont des objets instances de la classe STANDARD-CLASS. Pour obtenir le nom interne de la classe d'un objet représentant une classe, on peut utiliser (FIND-CLASS <nom>).

Exemple:

```
  ? (setq classe-cube (defclass cube () (...)))
  = #<standard-class cube>
  ? (find-class 'cube)
  = #<standard-class cube>
```

La fonction CLASS-NAME permet d'accéder au nom externe de la classe:

```
  (class-name classe-cube)                ---->     cube
```

X.2 - Création des instances

Pour créer des instances d'une classe il faut utiliser la fonction MAKE-INSTANCE:

(MAKE-INSTANCE <classe> {valeurs d'initialisation des champs}*)

Exemple:

```
  (DEFCLASS personne ()
          ((nom   :accessor        nom
                          :initarg :nom)
          (age            :accessor        age
                          :initform        10
                          :initarg :age)
  ))

  (make-instance 'personne :age 100)
```

Par défaut le champ "age" de cette instance est 10.

Il est souvent commode de définir ses propres fonctions de construction d'instance plutôt que d'appeler la fonction MAKE-INSTANCE directement; ainsi on peut cacher certains détails d'implantation et on n'a pas besoin d'utiliser les mots-clés pour initialiser les champs.

Par exemple, on peut définir la fonction suivante:

```
(defun make-personne (nom age)
  (make-instance 'personne :nom nom :age age) )
```

si on veut fournir le nom et l'âge comme paramètres de cette fonction plutôt que de fournir les valeurs en les faisant précéder des mots-clés :nom et :age.

Exemple:

```
? (setq p1 (make-instance 'personne :nom 'gilles :age 30))
= #<personne <adresse-memoire> >

? (nom p1)
= gilles
```

La création d'une instance s'effectue en deux temps:
1) création physique de la structure correspondant à l'instance
2) initialisation des champs de l'instance; la méthode INITIALIZE-INSTANCE est automatiquement appelée avec les arguments du "make-instance". Cette décomposition en 2 étapes est très intéressante car elle permet, en redéfinissant la méthode INITIALIZE-INSTANCE, de pouvoir déclencher des réflexes accompagnant la création d'instances.

Exemple:

```
? (defclass rectangle ()
        ((x  :accessor x  :initarg :x   :initform 0)
         (y  :accessor y  :initarg :y   :initform 0)
         (perimetre  :accessor perimetre  :initarg :perimetre))
        (:documentation "la classe des rectangles"))
= #<clos:standard-class rectangle>

? (defmethod initialize-instance :after ((oself rectangle) &rest args)
        (declare (ignore args))
        (setf (perimetre oself) (* 2 (+ (x oself) (y oself)) )) )

? (setq r1 (make-instance 'rectangle :x 2  :y 3))
= #<rectangle ..... >

? (perimetre r1)
= 10
```

X.3 - Affichage des objets CLOS

Tous les renseignements portant sur une instance peuvent être obtenus en appelant la fonction DESCRIBE:

```
?(describe p1)
= #<personne...> est une instance de la classe
  #<clos:standard-class personne...>:
   les valeurs des champs sont:
  nom Gilles
  age 30
```

On peut remarquer que lorsque l'on utilise DEFCLASS, les instances sont affichées avec la notation #<...>, plutôt que #s(personne ...). Mais il est possible de changer la façon dont les instances sont affichées en définissant des méthodes surchargeant la fonction générique PRINT-OBJECT.

X.4 - Accès en lecture et en écriture aux champs d'une instance

Par défaut, l'accès aux champs (ou slots) se fait en utilisant :

(SLOT-VALUE <instance> <nom-slot>)

```
? (slot-value p1 'nom)                    ; accès en lecture
= gilles

? (setf (slot-value p1 'nom) 'Jérome)     ; accès en écriture
= Jérome
```

L'exécution de la fonction SLOT-VALUE ne fait jamais appel aux réflexes "before", "after", etc. pouvant être définis lors des redéfinitions des méthodes d'accès aux champs (voir le paragraphe portant sur la combinaison des méthodes). La fonction SLOT-BOUNDP permet de savoir si un champ a déjà une valeur, et la fonction SLOT-MAKUNBOUND supprime la valeur d'un champ (qui ne devient pas pour autant égale à NIL).

Les fonctions d'accès définies dans "DEFCLASS" peuvent être utilisées pour lire ou modifier les valeurs des champs:

```
? (age p1)
= 30

? (setf (age p1) 31)
= 31

? (age p1)
= 31
```

X.5 - Héritage des options des champs

La classe "personne" décrite ci-dessus n'a pas fait l'objet d'un héritage de classe lors de sa définition. En fait elle a une superclasse qui est la classe prédéfinie STANDARD-OBJECT.

Lorsque l'on référence des superclasses lors de la définition d'une classe, on peut malgré tout indiquer un champ qui a déjà été spécifié dans une superclasse. Dans ce cas les informations sur les options des champs sont combinées.

Pour les options :ACCESSOR et :INITARG, le résultat est l'union des valeurs (l'accès au champ peut être fournie au moyen de l'ancienne valeur de l'option ou de la nouvelle, au choix); pour l'option :INITFORM, c'est la nouvelle valeur qui est prépondérante.

Voici quelques définitions de sous-classes:

```
?(defclass professeur (personne)
  ((matiere         :accessor professeur-matiere
                    :initarg :matiere )))
=#<clos:standard-class professeur @...>

?(defclass professeur-math (professeur)
  ((matiere         :initform "mathematiques")
   (nom    :accessor nom-prof-math))
=#<clos:standard-class professeur-math @...>

?(setq p2 (make-instance 'professeur-math :nom 'Jean :age 34))
=#<professeur-math @...>

? (describe p2)
...
nom       Jean
age       34
matiere   mathematiques

? (nom-prof-math p2)
```

= Jean
? (nom p2)
= Jean

X.6 - Héritage multiple

Une classe peut avoir plus d'une superclasse. Avec l'héritage simple (une seule superclasse) il est facile d'ordonner les superclasses des plus spécifiques aux moins spécifiques.

Pour ordonner les superclasses dans le cas de l'héritage multiple, on procède à tous les niveaux de la façon suivante:
Pour chaque classe, on retient la liste fournie à sa création, ordonnée par priorité décroissante, de ses superclasses directes. Cette liste définit la précédence locale.
A partir des superclasses directes, on peut déterminer toutes les superclasses à quelque niveau que ce soit.
Lors d'un appel de fonction générique, le graphe d'héritage est linéarisé pour définir la précédence globale.

Pour chaque classe, la hiérarchie de superclasses est donc linéarisée, de la plus spécifique à la plus générique, et les méthodes sont recherchées dans cette liste des superclasses ordonnées. Toutes les classes ordinaires héritent de standard-object, qui elle-même hérite automatiquement de T (le type plus général que tous les autres).

Deux règles sont utilisées pour cette linéarisation:

Règle 1 = Chaque classe est plus spécifique que ses superclasses.

Supposons que nous ayons:
(defclass A (B C)...)

La classe A est plus spécifique que B ou C; il reste à expliciter comment faire pour départager B et C si une option de champ est définie dans B et dans C ? Laquelle est prépondérante? La règle en CLOS est que les superclasses énoncées les premières sont plus spécifiques que les suivantes.

Règle 2 = Pour une classe donnée, les superclasses énoncées les premières sont plus spécifiques que celles énoncées ensuite.

Ces règles sont utilisées pour déterminer une relation d'ordre linéaire pour une classe et toutes ses superclasses, des plus spécifiques aux moins spécifiques. La "liste de précédence des classes" d'une classe reflète cet ordre.

Cependant, les deux règles ne sont pas toujours suffisantes pour déterminer un ordre unique, CLOS dispose d'un algorithme pour départager les ex-acquo. Ceci assure que toutes les implémentations fournissent le même ordre. Mais on considère habituellement que c'est une mauvaise méthode de travail que de compter sur cet ordonnancement. Si l'ordre des superclasses est important, il est préférable de l'expliciter directement dans la définition des classes.

Algorithme de CLOS:

Spécification de l'algorithme de calcul de la précédence entre les classes en cas d'héritage multiple:

R1 : respecter la profondeur
si C1 est une superclasse de C2, alors C2 précède C1 dans toutes les listes de précédence où elles apparaissent ensemble.

R2 : respecter l'ordre gauche-droite de la précédence locale
si C1 précède C2 dans la liste de définition des superclasses d'une classe, alors C1 précède C2 dans la liste de précédence.

R3 : respecter le focus

si C1 est placé avant C2 dans une liste de précédence, alors toutes les superclasses de C1 doivent être placées avant C2 si c'est compatible avec les règles R1 et R2.

Il n'y a qu'une manière de linéariser le graphe en respectant ces 3 règles.

Exemples:

1) Il est possible d'écrire un ensemble de définitions de classe qui ne peuvent pas être ordonnées. Dans ce cas, CommonLisp signale une erreur lorsqu'il essaie de calculer la liste de précédence.

```
(1) (DEFCLASS fruit () ())
(2) (DEFCLASS pomme (fruit) ())
(3) (DEFCLASS  pomme-fruit (fruit pomme) ())

(MAKE-INSTANCE 'pomme-fruit)      -->  Erreur car "pomme" précède
                                       "fruit" dans (2) et lui succède dans (3)
```

2) Exemple de configuration semblant conflictuelle, mais étant traitée correctement par l'algorithme de CLOS.

```
(DEFCLASS pomme () ())
(DEFCLASS cannelle () ())
(DEFCLASS tarte (pomme cannelle) ())
(DEFCLASS pâte (cannelle pomme) ())
```

La liste de précédence des classes pour "tarte" est
(tarte pomme cannelle standard-object t)

et celle de "pâte" est :
(pâte cannelle pomme standard-object t)

Il n'est cependant pas possible de construire une nouvelle classe héritant de "tarte" et de "pâte".

X.7 - Les fonctions génériques et les méthodes

La particularité de la programmation objet est de concevoir les opérations et messages entre objets à l'aide de fonctions génériques, fonctions dont le comportement spécifique pour les instances de classes particulières sera défini à l'aide de méthodes.

(DEFGENERIC <nom-fonction> <lambda-liste>) est utilisé pour définir une fonction générique. Il n'est pas obligatoire d'appeler la fonction DEFGENERIC avant de faire appel à DEFMETHOD. En effet, dans le cas où une fonction générique n'aurait pas été créée avant cet appel, elle serait définie automatiquement. Pour autant, il est souvent préférable d'utiliser DEFGENERIC pour déclarer qu'une certaine opération existe et qu'elle nécessite certains paramètres.

Une méthode est définie par:
(DEFMETHOD <nom-fonction-générique> <liste-spécialisée> <forme>*)

Cette définition ressemble beaucoup à celle obtenue par DEFUN, en particulier on peut y trouver les mots-clés &OPTIONAL, &REST, &KEY. La différence réside dans la <liste-spécialisée>. Chaque paramètre apparaissant dans cette liste peut être spécialisé, c'est à dire qu'il peut être défini comme étant d'un certain type ou d'une certaine classe.
La <liste-spécialisée> peut être de la forme:
((var1 type1) (var2 type2) ... varN ...)

La spécialisation est optionnelle. L'omettre signifie que la méthode peut être appliquée aux instances de n'importe quelle classe ou aux données de n'importe quel type.

Par exemple:

```
(defmethod perimetre ((c cercle)) (* 2 pi (cercle-rayon c)))
```

```
(defmethod perimetre ((s carré)) (* 4 (carre-cote s)))

(defmethod perimetre ((tr triangle))
        (+ (triangle-cote1 tr) (triangle-cote2 tr) (triangle-cote3 tr)))


(defmethod change-matiere ((prof professeur) nouvelle-matiere)
  (setf (professeur-matiere prof) nouvelle-matiere))
```

Ici la valeur de "nouvelle-matiere" peut être de n'importe quel type. Si on souhaite restreindre son type, on peut écrire cette définition:

```
(defmethod change-matiere ((prof professeur) (nouvelle-matiere string))
  (setf (professeur-matiere prof) nouvelle-matiere))
```

On peut également définir des classes pour les matières.

Dans la programmation objet "classique", les méthodes ne spécialisent qu'un seul paramètre. En CLOS, il est possible d'en spécialiser plus d'un. Dans ce cas, la méthode est parfois appelée multi-méthode.

Une méthode définie pour une classe C est prépondérante sur une méthode de même nom définie pour une superclasse de C. La méthode pour C est "plus spécifique" que la méthode pour la superclasse, car C est plus spécifique que les classes dont elle hérite.

Pour les multi-méthodes, déterminer quelle est la méthode la plus spécifique est plus complexe. Les paramètres sont considérés de gauche à droite.

Dans l'exemple qui suit, "integer" est plus spécifique que "number".

Exemples:

```
(defmethod test   ((x number) (y number))   '(num num))
(defmethod test   ((x integer) (y number))   '(int num))
(defmethod test   ((x number) (y integer))   '(num int))

(test 1 1) ∅  (int num)     et non (num int) ;  C'est la 2ème méthode qui est appliquée
(test 1 1/2)        ∅  (int num)
(test 1/2 1)        ∅  (num int)
(test 1/2 1/2)      ∅  (num num)
```

Cas particulier des méthodes d'accès aux champs

Il est possible de redéfinir les méthodes d'accès en lecture et en écriture aux champs automatiquement créées lors de la création d'une classe.

Pour l'accès en lecture: (defmethod <nom-champ> [<mot-clé>] ((<id-var> <id-type>)) ...)

Pour l'accès en écriture: (defmethod (setf <nom-champ>) [<mot-clé>]
(<nouvelle-valeur> (<id-var> <id-type>)) ...)

Le mot-clé peut être :before, :after ou :around (voir le § suivant).

X.8 - Combinaison de méthodes

Quand plus d'une classe définit une méthode pour une fonction générique et que plus d'une méthode est applicable pour un ensemble donné d'arguments, les méthodes applicables sont combinées en une seule "méthode effective". Chaque définition des méthodes individuelles devient alors une partie de la définition de la méthode effective.

CLOS propose un type de combinaison de méthodes, appelé combinaison standard de méthodes. Il est aussi possible de définir d'autres sortes de combinaison de méthodes.

La combinaison standard de méthodes met en jeu quatre sortes de méthodes:

* Les méthodes primaires forment le corps principal de la méthode effective. Seule la méthode primaire la plus spécifique est appelée, mais il peut être fait référence à la méthode primaire spécifique suivante dans la hiérarchie des classes en appelant la fonction:
(call-next-method)

* Les méthodes ":BEFORE" sont toutes appelées avant la méthode primaire; l'appel de la méthode :BEFORE la plus spécifique étant fait en premier.

* Les méthodes ":AFTER" sont toutes appelées après la méthode primaire; l'appel de la méthode :AFTER la plus spécifique étant fait en dernier.

* Les méthodes ":AROUND" s'exécutent à la place des autres méthodes.

Les méthodes ":BEFORE", ":AFTER", et ":AROUND" sont spécifiées en plaçant le mot clé correspondant comme qualifieur dans la définition de la méthode. Les méthodes :BEFORE et :AFTER sont les plus faciles à utiliser.

Exemples:

```
(defclass plat      () ())

(defmethod cuisiner ((x plat))
  (print "Préparer un plat")
)
(defmethod  cuisiner  :before ((p plat))
  (print "Un plat va être cuisiné"))

(defmethod  cuisiner  :after ((p plat))
  (print "Un plat a été cuisiné"))

(defclass tarte (plat)
  ((garniture :accessor tarte-garniture :initarg :garniture :initform 'poire)))

(defmethod cuisiner ((x tarte))
  (print "Préparer une tarte")
  (setf (tarte-garniture x) (push 'cuite (tarte-garniture x) ))
)

(defmethod  cuisiner  :before ((x tarte))
  (print "Une tarte va être préparée"))

(defmethod  cuisiner  :after ((x tarte))
  (print "Une tarte a été preparee"))

(setf tarte-1 (make-instance 'tarte :garniture 'pomme))
```

Et maintenant:

```
? (cuisiner tarte-1)
"Une tarte va être preparee"
"Un plat va être cuisiné"
"Préparer une tarte"
"Un plat a été cuisiné"
"Une tarte a été préparée"
(pomme cuite)
```

X.9 - Apports d'un méta-protocole

CLOS propose un méta-protocole. Les classes sont elles même des instances de métaclasses, et il est possible de créer de nouvelles métaclasses. Un méta-protocole offre des outils extrêmement précieux d'introspection du code. Ils permettent d'examiner une application au cours de son fonctionnement même, et par conséquent d'améliorer ce dernier.

Un méta-protocole est aussi un moyen d'étendre le langage sans pour autant le dénaturer. Les extensions ne s'y substituent pas : comme elles reposent sur la spécialisation et le masquage éventuel, elles coexistent avec le fonctionnement ordinaire du langage. G. Kiczales et al. (1991) donnent de nombreux exemples suggestifs des extensions possibles et souhaitables : cohabitation de calculs distincts de listes de précédences, possibilité de pouvoir protéger davantage l'accès à des champs (à la C++), champs notant les accès qui leur sont faits, etc.

CLOS apparaît alors comme l'un des seuls langages qui intègre les moyens de son évolution et de l'expérimentation de pistes nouvelles, et qui offre un protocole pour le faire.

X.10 - Liste des fonctions définies dans le package CLOS

- Outils pour la programmation objet simple:

call-next-method	initialize-instance
change-class	make-instance
defclass	next-method-p
defgeneric	slot-boundp
defmethod	slot-value
generic-flet	with-accessors
generic-function	with-added-methods
generic-labels	with-slots

- Fonctions sous-jacentes aux macros utilisées souvent:

add-method	reinitialize-instance
class-name	remove-method
compute-applicable-methods	shared-initialize
ensure-generic-function	slot-exists-p
find-class	slot-makunbound
find-method	slot-missing
function-keywords	slot-unbound
make-instances-obsolete	update-instance-for-different-class
no-applicable-method	update-instance-for-redefined-class
no-next-method	

- Outils pour combiner déclarativement des méthodes

call-method	method-combination-error
define-method-combination	method-qualifiers
invalid-method-error	

XI - Les OUTILS DE MISE AU POINT

Parmi les outils de mise au point fournis avec Common Lisp, on distingue :
- les outils de pistage: trace et exécution pas-à-pas
- les points d'arrêt et la mise en place de boucles d'inspection pour examiner l'environnement en cas d'erreur.

Quand une erreur se déclenche, un message est imprimé et il est possible d'examiner l'environnement à l'aide de commandes que l'on peut obtenir en tapant ?. CTRL/D ou abort permettent de revenir au Top Level.

Menu de BREAK:

```
Help        = ce menu-ci
Abort       = arrêt, retour au niveau supérieur
Unwind      = arrêt, retour au niveau supérieur
Mode-1      = examiner tous les éléments de la pile
Mode-2      = examiner tous les <frames>
Mode-3      = examiner uniquement les <frames> lexicaux
Mode-4      = examiner uniquement les <frames> EVAL et APPLY (par défaut)
Mode-5      = examiner uniquement les <frames> APPLY
Where       = examiner ce <frame>
Up          = examiner un <frame> supérieur
Top         = examiner le <frame> le plus élevé
Down        = examiner un prochain <frame> plus récent (inférieur)
Bottom      = examiner le <frame> le plus récent (le plus bas)
Backtrace-1 = montrer tous les éléments de la pile
Backtrace-2 = montrer tous les <frame>
Backtrace-3 = montrer tous les <frame> lexicaux
Backtrace-4 = montrer tous les <frame> EVAL et APPLY
Backtrace-5 = montrer tous les <frame> APPLY
Backtrace   = montrer la pile en mode actuel
Break+      = placer un point d'interception dans le <frame> EVAL
Break-      = enlever le point d'interception du <frame> EVAL
Redo        = réévaluer la forme dans le <frame> EVAL
Return      = quitter le <frame> EVAL avec certaines valeurs
```

Les fonctions de pistage sont TRACE (et UNTRACE) et STEP:

a) (TRACE <symb1> ... <symbN>)

Permet de pister les fonctions décrites par les arguments <symb1>, ... <symbN>. Ces arguments ne sont pas évalués.

Exemple d'utilisation du pisteur

```
?(defun rv (s res)
?    (if (null s)
?    res
?    (rv (cdr s) (cons (car s) res))))
=rv

?(rv '(1 2 3) ())
=(3 2 1)

?(TRACE rv)
=(rv)

?(RV '(1 2 3) ())
1. TRACE: (RV '(1 2 3 'NIL)
2. TRACE: (RV '(2 3) '(1))
3. TRACE: (RV '(3) '(1 2))
4. TRACE: (RV '(5) '(1 2 3))
4. TRACE: RV => (3 2 1)
3. TRACE: RV => (3 2 1)
2. TRACE: RV => (3 2 1)
1. TRACE: RV => (3 2 1)
=(3  2  1)
```

b) (UNTRACE <sym1> ... <symN>)

Enlève le pistage pour les différentes fonctions de nom <sym1>, <symN>. Si aucun argument n'est fourni, UNTRACE enlève l'ensemble des pistages actifs.

c) (STEP <s>)

Evalue l'expression <s> en mode pas à pas. Retourne la valeur de l'évaluation de <s>.

L'exécution en mode pas à pas permet de s'arrêter à chaque appel interne de l'évaluateur et donc de suivre très finement le déroulement d'un programme.

A chaque arrêt, différentes actions (que l'on peut obtenir en tapant ?) sont possibles:

Menu de STEP:

```
Help        = ce menu-ci
Abort       = arrêt, retour au niveau supérieur
Unwind      = arrêt, retour au niveau supérieur
Mode-1      = examiner tous les éléments de la pile
Mode-2      = examiner tous les <frame>
Mode-3      = examiner uniquement les <frame> lexicaux
Mode-4      = examiner uniquement les <frame> EVAL et APPLY (par défaut)
Mode-5      = examiner uniquement les <frame> APPLY
Where       = examiner ce <frame>
Up          = examiner un <frame> supérieur
Top         = examiner le <frame> le plus élevé
Down        = examiner un prochain <frame> plus récent (inférieur)
Bottom      = examiner le <frame> le plus récent (le plus bas)
Backtrace-1 = montrer tous les éléments de la pile
Backtrace-2 = montrer tous les <frame>
Backtrace-3 = montrer tous les <frame> lexicaux
Backtrace-4 = montrer tous les <frame> EVAL et APPLY
Backtrace-5 = montrer tous les <frame> APPLY
Backtrace   = montrer la pile en mode actuel
Break+      = placer un point d'interception dans le <frame> EVAL
Break-      = enlever le point d'interception du <frame> EVAL
Redo        = réévaluer la forme dans le <frame> EVAL
Return      = quitter le <frame> EVAL avec certaines valeurs
Step        = step into form: Évaluer cette forme petit à petit
Next        = step over form: Évaluer cette forme en bloc
Over         = step over this level: Évaluer tout le reste jusqu'au prochain
retour
Continue    = continue: Évaluer tout le reste en bloc
Step-until, Next-until, Over-until, Continue-until:
           de même, avec spécification d'une condition d'arrêt
```

XII - DÉFINITIONS DÉTAILLÉES DE QUELQUES FONCTIONS

Définition des structures:

(DEFSTRUCT <nom-struct> <documentation> <desc-champ 1> ... <desc-champ N>)
[MACRO]

<nom-struct> est un symbole devenant le nom de la structure et qui est un nouveau type d'objet-LISP, ou une liste comportant le nom de la structure et des options:

:conc-name pour fournir un préfixe pour les noms des fonctions d'accès aux champs (par défaut <nom-struct>-<champ>)

:constructor pour donner un nom spécial à la fonction de construction d'une instance (par défaut MAKE-<nom-struct>)

:copier pourr donner un nom spécial à la fonction de copie d'une instance (par défaut COPY-<nom-struct>)

:predicate pour donner un nom de prédicat pour tester le type d'un objet (par défaut <nom-struct>-P)

:include pour construire une structure comme étant une extension d'une autre structure

:print-function pour écrire les structures avec un style particulier (par défaut #S(...))

:type spécifie la représention de la structure (vector, ou list)

:named indique si la structure est nommée (oui par défaut, sauf si une option :type a été définie)

:initial-offset pour affiner la représentation interne dans le cas où une option :type est précisée.

<desc-champ i> est de la forme:

(<nom-champ> <valeur-par-défaut> <option1> <valeur1> ... <optionK> <valeurK>)

Liste des options possibles:

:type spécifie le type de la valeur du champ

:read-only indique que le champ ne peut pas être modifié

(DEFCLASS *class-name* ({*superclass-name* }*)
({*slot-specifier* }*) [[↓*class-option*]]) [MACRO]

class-name ::= *symbol*
superclass-name ::= *symbol*
slot-specifier ::= *slot-name* | ((*slot-name* [[↓*slot-option*]])
slot-name ::= *symbol*
slot-option ::= { :reader *reader-function-name* } *

| {:writer *writer-function-name* } *

| {:accessor *reader-function-name* } *

| {:allocation *allocation-type* }

| {:initarg *initarg-name* } *

| {:initform *form* }

| {:type *type-specifier* }

| {:documentation *string* }

reader-function-name ::= *symbol*
writer-function-name ::= *function-name*
function-name ::= {*symbol* | (setf *symbol*)}
initarg-name ::= *symbol*
allocation-type ::= :instance | :class
class-option ::= (:default-initargs *initarg-list*)
| {:documentation *string* }
| {:metaclass *class-name* }
initarg-list ::={*initarg-name default-initial-value-form* }*

Définition de fonctions s'adaptant aux types des arguments

(DEFGENERIC *function-name lambda-list* [MACRO]
[[↓*option* | {*method-description* }*]])

function-name ::= {*symbol* | (setf *symbol*)}
lambda-list ::= ({*var*}*
[&optional { *var* | (*var*) }*]
[&rest *var]*
[&key {*keyword-parameter*}* [&allow-other-keys]])

keyword-parameter ::= *var* | ({ *var* | (*keyword var*) })
option ::= (:argument-precedence-order {*parameter-name*}+)
| (declare {*declaration*}+)
| (:documentation *string*)

| (:method-combination *symbol* {*arg*}*)
| (:generic-function-class *class-name*)
| (:method-class *class-name*)

method-description ::= (:method {*method-qualifier*}*
specialized-lambda-list
[[{*declaration*}* | *documentation]]*
{*form*}*)

method-qualifier ::= *non-nil-atom*

specialized-lambda-list ::=
({*var* | (*var parameter-specializer-name*) }*
(&optional {*var* | (*var [initform* [*supplied-p-parameter*)] }*]
[&rest *var]*
[&key {*specialized-keyword-parameter*}* [&allow-other-keys]]
[&aux {*var* | (*var [initform]*)}*])

specialized-keyword-parameter ::=
var | ({ *var* | (*keyword var*) } *[initform* [*supplied-p-parameter*]])

parameter-specializer-name ::= *symbol* | (eq1 *eql-specializer-form*)

(DEFMETHOD <nom-fonction> <lvar-spéciale> [{déclaration}* | <documentation>]
{<forme>}*)

où:

- <nom-fonction> est un <symbole> ou une liste de la forme (setf <symbole>)

- <lvar-spéciale> = ({var | (var <spécialisation du paramètre>)}*
[&optional <paramètres-optionnels>]
[&rest var]
[&key <paramètre definis par mots cles>]
[&aux {var | (var [forme d'initialisation])}*])

<spécialisation du paramètre> permet de préciser le type du paramètre et d'adapter la méthode à des types de paramètres particuliers. Le type peut être le nom d'un type ou une expression telle que (eql <domaine du paramètre>).

XIII - RÉFÉRENCES BIBLIOGRAPHIQUES

"Langage d'un autre type : Lisp", Christian QUEINNEC, Eyrolles (France), 1982.

"Le Langage Lisp", Miche CAYROL, Cepadues Editions, Toulouse, 1983.

"Lisp" , Patrick H. WINSTON et Berthold K. HORN, Addison Wesley, 1988, 3rd edition.

"Common Lisp, the Language", Guy L. STEELE, Burlington MA (USA), Digital Press, 1990, 2nd edition.

"Les langages Lisp", Christian QUEINNEC, InterEditions (France), 1994.

"Objectif : CLOS", Benoît Habert, Masson, 1996.

"The Art of the MetaObject Protocol", Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow, The MIT Press, 1991.

"Advances in Object-Oriented Metalevel Architectures and Reflection", Chris Zimmermann, CRC Press, 1996.

Adresse de la documentation en ligne :
http://e3000.ensicaen.ismra.fr/HyperSpec/FrontMatter