TABLE DES MATIERES

1. PROLEGOMENES 1
   1.1 EXERCICES 4

2. LES FONCTIONS DE BASE : QUOTE, CAR, CDR, CONS 5
   2.1 EXERCICES 9

3. UN AUTRE POINT DE VUE SUR LES LISTES 10
   3.1 EXERCICES 13

4. LA DEFINITION DE FONCTIONS 15
   4.1 EXERCICES 19

5. DES PREDICATS ET DE LA SELECTION 20
   5.1 QUELQUES PREDICATS 20
      5.1.1 Exercices 23
   5.2 LA FONCTION IF 23
      5.2.1 Exercices 26

6. LA REPETITION 28
   6.1 LA FONCTION REVERSE 28
   6.2 LA FONCTION APPEND 31
   6.3 LA FONCTION EQUAL 33
   6.4 LA FONCTION DELETE 37
   6.5 EXERCICES 38

7. L'ARITHMETIQUE 40
   7.1 LES FONCTIONS ARITHMETIQUES DE BASE 40
   7.2 LES PREDICATS NUMERIQUES 42
   7.3 LA PROGRAMMATION NUMERIQUE 43
   7.4 EXERCICES 50

8. LES P-LISTES 52
   8.1 LES SYMBOLES 52
   8.2 L'ACCES AUX P-LISTES 53
   8.3 EXERCICES 56

9. LES MEMO-FONCTIONS 58
   9.1 EXERCICES 61

10. LES ENTREES / SORTIES (PREMIERE PARTIE) 63

10.1 LA FONCTION GENERALE D'IMPRESSION : PRINT 63
10.2 LES CHAINES DE CARACTERES 68
10.3 LES FONCTIONS PRIN1, TERPRI ET PRINCH 70
10.4 L'ACCES AUX NOMS DES ATOMES 78
10.5 EXERCICES 85

11. QUELQUES REMARQUES SUR LES MECANISMES DE LIAISON 88
11.1 LES DIVERS TYPES DE FONCTIONS UTILISATEURS EN Le_LISP : DE et DF 88
11.1.1 LA FONCTION DE 88
11.1.2 LA FONCTION DF 91
11.1.3 EXERCICES 92
11.2 QUELQUES DETAILS SUR LES DEFINITIONS DE FONCTIONS EN CommonLISP 92
11.2.1 EXERCICES 99

12. LES FONCTIONS EVAL ET APPLY 100
12.1 LA FONCTION EVAL 101
12.2 LA FONCTION APPLY 103
12.3 EXERCICES 106

13. LES ENTREES / SORTIES (DEUXIEME PARTIE) 107
13.1 LA FONCTION GENERALE DE LECTURE : READ 107
13.2 LES AUTRES FONCTIONS D'ENTREE / SORTIE : READ-CHAR, PEEK-CHAR 109
13.3 LES DIVERS TYPES DE CARACTERES 118
13.3.1 détermination du type d'un caractère 118
13.3.2 les macros-caractères 123
13.3.2.1 Définition de macro-caractères en Le_LISP 123
13.3.2.2 Définition de macro-caractères en CommonLISP 124
13.3.2.3 Un exemple plus élaboré 127
13.4 COMMENT CHANGER DE FICHIERS D'ENTREE/SORTIE 132
13.5 EXERCICES 139

14. LE FILTRAGE (PREMIERE PARTIE) 141
14.1 PREMIERE VERSION D'UNE FONCTION DE FILTRAGE 142
14.2 EXERCICES 145

15. RETOUR VERS LES LISTES 147
15.1 L'ORGANISATION DE LA MEMOIRE LISP 147
15.2 LA ZONE LISTE 148
15.3 LES FONCTIONS DE MODIFICATION DE LISTES 155
15.4 LES FONCTIONS EQ ET EQUAL 161
15.5 LES LISTES CIRCULAIRES 163
15.6 LES AFFECTATIONS 166
15.7 EXERCICES 170

16. UN MICRO-PROJET 172
16.1 LES ECHAPPEMENTS 180
16.2 RETOUR VERS LES MACHINES DE TURING 184

17. LES MACRO-FONCTIONS 191
17.1 EXERCICES 199

18. LES DIVERSES FORMES DE REPETITION 200
18.1 APPEL DE FONCTION IMPLICITE (OU LE BRANCHEMENT) 200
18.2 LES ARGUMENTS FONCTIONNELS 201
18.3 QUELQUES ITERATEURS SUPPLEMENTAIRES 209
18.4 EXERCICES 213

19. LE FILTRAGE (DEUXIEME PARTIE) 214

20. UN MINI-PROJET 235
20.1 QUELQUES REMARQUES SUR LA CORRECTION DE PROGRAMMES 235
20.2 LA LOGIQUE DE HOARE 236
20.2.1 La Règle d'Affectation 238
20.2.2 La Première Règle de Conséquence 239
20.2.3 La Règle du Conditionnel 240
20.2.4 La Deuxième Règle de Conséquence 241
20.2.5 La Règle de l'Itération 243
20.2.6 Les Règles Concernant des Procédures 245
20.3 LE PROGRAMME GCV 249

21. SOLUTIONS AUX EXERCICES 263
21.1 CHAPITRE 1 263
21.2 CHAPITRE 2 263
21.3 CHAPITRE 3 264
21.4 CHAPITRE 4 265
21.5 CHAPITRE 5 266
21.5.1 chapitre 5.1.1 266
21.5.2 chapitre 5.2.1 267
21.6 CHAPITRE 6 267
21.7 CHAPITRE 7 270
21.8 CHAPITRE 8 272
21.9 CHAPITRE 9 278
21.10 CHAPITRE 10 280
21.11 CHAPITRE 11 284
21.11.1 chapitre 11.1.3 284
21.11.2 chapitre 11.2.1 285
21.12 CHAPITRE 12 286
21.13 CHAPITRE 13 287
21.14 CHAPITRE 14 290
21.15 CHAPITRE 15 292

21.16 CHAPITRE 17 294
21.17 CHAPITRE 18 295

22. BIBLIOGRAPHIE 297

23. LES FONCTIONS LISP STANDARD 301
23.1 Le_LISP 301
23.2 CommonLISP 312

24. INDEX 323

AVANT-PROPOS

LISP est un des plus anciens langages de programmation : ses premières implémentations ont vu le jour à la fin des années 60, quelques années seulement après FORTRAN.

LISP a été conçu par John McCarthy pour le traitement d'expressions symboliques. Dès son origine, il a été utilisé pour écrire des programmes de calcul symbolique différentiel et intégral, de théorie des circuits électriques, de logique mathématique et de programmation de jeux.

LISP est également un des langages de programmation les plus répandus : il existe pratiquement sur tous les ordinateurs, de toute taille et de toute origine.

LISP est un des langages de programmation les plus vivants. Le nombre étonnant de versions différentes, telles que MACLISP, INTERLISP, CommonLISP, VLISP, LE_LISP, etc., en témoigne. De plus, signalons l'existence de machines utilisant une architecture spécialement conçue pour l'implémentation de LISP, par exemple les ordinateurs de Symbolics ou de la LISP-Machine Company, ou, en France, l'ordinateur Maia, développé à la CGE.

En LISP, de même que dans les langages-machine, la représentation des données et celle des programmes est identique : ainsi un programme LISP peut construire d'autres programmes LISP ou se modifier au fur et à mesure de son exécution. De plus, cette identité de représentation permet d'écrire LISP en LISP même.

LISP est un langage interactif avec un environnement de programmation intégré. Ceci implique non seulement que tous les programmes et toutes les données soient accessibles, modifiables et analysables en LISP même, sans sortir du langage, mais également que la construction des programmes s'accélère considérablement, le cycle :

édition de texte
↓
compilation
↓
édition de liens
↓
chargement
↓
exécution

des langages compilés classiques se réduit au cycle.

lecture→ évaluation

Ces vertus ainsi que la simplicité de la syntaxe de LISP, la possibilité de programmer immédiatement un problème sans passer par des stades de déclarations de variables ou de types, et finalement le fait que LISP soit fondé sur la récursivité, font de LISP un excellent langage d'apprentissage et d'enseignement de la programmation.

L'apprentissage de LISP est le sujet de ce livre d'introduction à la programmation. Il ne présuppose aucune connaissance de la part du lecteur. A travers des exemples commentés il introduit le langage LISP, ses structures de données, ses structures de contrôle, et sa programmation.

Comme nous l'avons indiqué ci-dessus, LISP se présente dans des versions diverses : bien qu'il y ait régulièrement des efforts pour trouver un standard de LISP, le développement de ce magnifique langage semble difficile à arrêter et le programmeur LISP travaille dans des

dialectes spécifiques. Jusqu'à récemment, LISP était principalement confiné à des environnements universitaires ou de recherche. Ce n'est que grâce au succès commercial des systèmes experts et de quelques programmes d'intelligence artificielle que la pression est devenue suffisamment forte pour obliger la communauté LISP à développer un noyau LISP standardisé. Ce noyau du langage LISP s'appelle ComonLISP. Les dialectes utilisés dans ce livre sont d'une part CCL, une implémentation de CommonLISP développée à l'université Paris VIII (Vincennes) par Partick Greussay, et, d'autre part, LE_LISP, le dialecte issu de VLISP et de Common LISP développé par Jérôme Chailloux à l'INRIA. Ces deux dialectes sont les versions de LISP les plus répandues en France. Lorsque des différences existent entre ces deux dialectes, elles sont indiquées clairement.

La plupart des chapitres se compose de deux parties : la première consiste en une présentation des nouveaux concepts abordés dans le chapitre, la deuxième est une suite d'exercices dont toutes les solutions sont exposées à la fin du livre. Ces exercices font partie intégrante de chaque chapitre. Souvent les chapitres présupposent une compréhension des exercices des chapitres précédents.

Voici le plan du livre :

- Les trois premiers chapitres introduisent les structures de données standard de LISP, atomes et listes, ainsi que les fonctions de base permettant leur accès et leur construction. Sans une compréhension de ces structures et de ces fonctions nous ne pouvons écrire aucun programme.

- Au chapitre 4 nous introduisons la possibilité pour l'utilisateur de se définir ses propres fonctions. C'est la base de toute écriture de programme LISP. Nous y abordons également les questions de liaisons des variables.

- Le chapitre 5 présente les prédicats les plus courants ainsi que la fonction de sélection **IF**.

- Le chapitre 6 introduit à travers quatre exemples la notion de récursivité, la forme de répétition la plus standard de LISP.

- Le chapitre 7 présente les fonctions arithmétiques et les bases de la programmation numérique en LISP.

- Aux chapitre 8 et 9 nous reprenons la notion d'atome et introduisons l'utilisation des P-listes pour attacher à un atome de multiples valeurs. Nous y traitons également une application particulière des P-listes : les fonctions mémorisant leurs activités.

- Le chapitre 10 traite toutes les questions concernant les fonctions d'impression.

- Les chapitres 11 et 12 exposent les divers mécanismes de liaisons des variables utilisées en LISP et la différence entre les fonctions de type EXPR et de type FEXPR. Nous y examinerons également les fonctions **EVAL** et **APPLY**.

- Au chapitre 13 nous reprenons des questions concernant les fonctions d'entrée/sortie ainsi que le rôle des divers caractères spéciaux de LISP.

- Au chapitre 14 nous construisons un petit programme de filtrage.

- Le chapitre 15 examine en détail la structure des listes et les fonctions modifiant physiquement ces structures. Nous y verrons également les fonctions d'affectation.

- Au chapitre 16 nous développons un simulateur de machines de Turing. Nous y introduisons les fonctions d'échappements non locaux.
- Le chapitre 17 présente un nouveau type de fonction : les macro-fonctions.
- Le chapitre 18 présente diverses formes de répétitions itératives.
- Au chapitre 19 nous reprenons l'exemple de la fonction de filtrage. Nous y construisons une fonction de filtrage très puissante, intégrant toutes les notions abordées précédemment.
- Au chapitre 20 nous abordons des questions de vérification de programmes à travers la construction d'un générateur de conditions de vérification pour des programmes LISP simples. Ce chapitre clôt notre apprentissage de LISP.
- Le chapitre 21 donne les solutions à l'ensemble des exercices donnés dans ce livre.
- Finalement, le chapitre 22 donne une bibliographie assez exhaustive de LISP ; le chapitre 23 résume les diverses fonctions Le_LISP et CommonLISP utilisées dans ce livre. C'est vers ce chapitre que vous devez vous tourner quand vous avez oublié la syntaxe ou la sémantique d'une fonction standard. Nous terminons au chapitre 24, comme tout livre, par un index.

Nous ne couvrons pas l'ensemble des fonctions disponibles en LISP : seules les plus importantes sont exposées. Pourtant le sous-ensemble de fonctions traitées est largement suffisant pour construire n'importe quel programme sur n'importe quel système.

Nous n'abordons pas non plus des questions d'implémentation de LISP, de sa compilation et — surtout — de son application en intelligence artificielle. Ces parties seront traitées dans un deuxième volume. Néanmoins, dans la bibliographie, à la fin de ce livre, nous citons également des livres concernant ces autres aspects de LISP.

Le contenu de ce livre correspond à un cours d'introduction à la programmation en langages évolués que l'auteur assure depuis un certain nombre d'années à l'université Paris VIII (Vincennes). Je tiens à remercier les étudiants, qui, par leurs remarques critiques, leurs questions et leur enthousiasme, ont largement influencé l'écriture de ce livre. C'est à eux que ce livre est dédié.

Ce livre a été saisi et mis en page par l'auteur à l'IRCAM sur ordinateur DEC CAX 780 sous système UNIX, grâce à la bienveillance de Pierre Boulez et David Wessel. Sa réalisation n'aurait pas été possible sans le soutien de l'équipe technique de l'IRCAM, notamment de Michèle Dell-Prane et de Patrick Sinz.

Evidemment, ce livre a bénéficié des suggestions, critiques, corrections et remarques d'un grand nombre de personnes. Qu'il me soit permis ici de remercier particulièrement Jean-Pierre Briot, Florence Caroff-Krief, Annette Cattenat, Frédéric Chauveau, Jacques Ferber, Mario Furnari, Jean-François Gay, Patrick Greussay, Daniel Goossens, Eric Halle, Philippe Krief, Jean Mehat, Christian Margale, Danièle Mathé, Gérard Nowak, Mourad Ouldmohand, Gérard Paul, Yves-Daniel Pérolat, Jean-François Perrot, Stéphane Rideau, Christian Riesner, Nicole Rocland, Bernard-Paul Serpette, Jacqueline Signorini, Patrick Sinz, Ellen Ann Sparer, Roger Tanguy.

AVANT-PROPOS pour la deuxième édition

Cette deuxième édition est complètement revisée et considérablement élargie. J'ai changé les dialectes utlisés. Si, dans la première édition le dialecte principal était VLISP et le dialecte secondaire. Le_LISP, dans cette nouvelle édition le dialecte principal est CommonLISP et le dialecte secondaire est Le_LISP. Ceci est dû d'une part à la très large disponiblilité de CommonLISP, qui semble s'installer de plus en plus comme une sorte de standard de LISP, deuxièmement au quasi arrêt des recherches portant sur VLISP : Le Groupe d'intelligence artificielle de l'université Paris VIII, principal responsable du développement de VLISP s'est tourné, au fur et à mesure des années vers d'autres sujets : les machines parallèles, les environnements de programmation, Smalltalk, etc. Avec la tendance à la standardisation de LISP, les recherches directement liées au développement de ce langage ne semblent plus si attractives qu'il y a quelques années. Heureusement que les intérêts de recherche progressent.

Cette deuxième édition contient deux nouveaux chapitres, chacun d'eux exposant en détail un projet de programmation.

Les nouveaux sujets abordés dans cette édition incluent des concepts aussi variés que les échappements, la liaison dynamique et lexicale, les déclarations d'objets LISP, les machines de Turing ou la vérification de programmes.

Néanmoins, ce livre reste, même si son volume s'est agrandi (peut-être est-ce un signe de l'avance de cette discipline), un livre d'introduction et peut être lu et étudié par toute personne intéressée par la programmation LISP.

Harald Wertz

à Ellen, Arik et Esther

1. PROLEGOMENES

Le langage utilisé dans ce livre d'introduction à la programmation est LISP.

LISP est un acronyme de **LIS**t **P**rocessor. Comme son nom l'indique, LISP traite des LISTES. Une liste en LISP est quelque chose qui commence par une parenthèse ouvrante "(" et qui se termine par une parenthèse fermante ")". C'est simple. Voici quelques exemples de listes :

(DO RE MI)
(OH LA LA)
(LE CUBE EST SUR LA TABLE)

Syntaxiquement une liste est définie comme :

$$\text{liste} ::= (\ \{qqc_1\ qqc_2 \ldots qqc_n\}\ )$$

et *qqc* est défini comme :

qqc ::= liste |atome

Le signe ::= veut dire *est défini comme*, et le signe | veut dire *ou*. Les choses entre accolades, { et }, sont optionnelles. Cela veut dire qu'il peut y en avoir ou pas suivant les cas. La définition précédente se lit donc "qqc est défini comme une liste ou un atome".

Continuons nos définitions :

atome	::=	nombre |nom
nombre	::=	0 |1 |2 |3 |.... |1024 |... |-1 |-2 |... |-1024 |... (ce qui veut dire qu'il peut y avoir des nombres positifs ou négatifs)
nom	::=	suite de caractères alphanumériques contenant au moins une lettre et ne contenant aucun *séparateur*
séparateur	::=	. |espace |(|) |[|] |' |tabulation |; |retour chariot ce sont des caractères qui ont un rôle particulier

Quelques exemples d'atomes :

DO
RE
MI
CECI-EST-UN-ATOME-TRES-LONG
CUBE
CUBE1
1A

Voici donc quelques exemples d'objets LISP :

```
128                                   ; un nombre ;
-32600                                ; un nombre négatif ;
HAHA          ; un atome nom (atome alphanumérique) ;
()                                    ; une liste à 0 élement ;
(HAHA)                                ; une liste à 1 élément ;
(UNE LISTE)                           ; une liste à 2 éléments ;
((UNE) LISTE)   ; c'est une liste à 2 éléments, le premier ;
                  ; élément est une liste à 1 élément, le ;
                    ; deuxième élément est un atome ;
(UNE (LISTE))     ; encore une liste à deux éléments ;
```

Voici trois exemples de choses qui ne sont pas des objets possibles en LISP :

```
)                          ; rien ne peut commencer par une parenthèse fermante
(TIENS (TIENS (TIENS                  ; ici, il y'a un tas de parenthèses
                                        ; fermantes manquantes
(...)                        ; malheureusement, le caractère "point"
                           ; n'est pas permis en LISP (pour l'instant)
```

Il est possible de compter les éléments d'une liste, on parle alors de la *longueur* d'une liste. La *longueur* d'une liste est le nombre de ses éléments.

Voici quelques listes de longueur 2 :

```
(1 2)
(TIENS TIENS)
( ((AHA)) (ENCORE UNE LISTE) )
(()())
```

et voici encore quelques listes et leurs longueurs :

listes	longueur
()	0
(JOHN GIVES MARY A BOOK)	5
((SUJET)(VERBE)(OBJET))	3
(((X + Y) + Z) --> (X + (Y + Z)))	3
(QUELQU-UN AIME MARY)	3
(DU PASSE FAISONS TABLE RASE)	5
(1 2 3 A B C C B A 3 2 1)	12

La liste vide, (), s'appelle **NIL**. Pour LISP l'atome **NIL** et la liste () ont la même valeur, c'est-à-dire : (). Vous pouvez l'écrire soit sous la forme d'un atome : **NIL**, soit sous la forme d'une liste : ().

Visiblement, les parenthèses sont toujours bien équilibrées : une liste, et donc également chaque sous-liste, comporte le même nombre de parenthèses ouvrantes et fermantes. De plus, en lisant une liste, caractère par caractère, de gauche à droite, le nombre de parenthèses fermantes est toujours inférieur au nombre de parenthèses ouvrantes rencontrées. Excepté, naturellement, à la fin de la liste où le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes.

Si vous avez des problèmes pour déterminer si un objet représente une liste bien formée, voici un petit algorithme qui peut vous aider :

> Si l'objet en question ne commence pas par une parenthèse ouvrante, bien évidemment, vous n'avez pas affaire à une liste et le tour est joué. Sinon, comptez les parenthèses de manière suivante : imaginez que vous ayez un compteur qui est initialement à zéro. Ensuite vous lisez l'objet de gauche à droite. A chaque rencontre d'une parenthèse ouvrante vous incrémentez votre compteur de 1 et vous marquez la parenthèse ouvrante par la valeur de votre compteur. Si vous rencontrez une parenthèse fermante, vous la marquez avec la valeur courante de votre compteur, ensuite vous le décrémentez de 1. Si, avant d'atteindre la fin de la liste vous obtenez (à un instant quelconque) un nombre inférieur ou égal à 0 dans votre compteur, alors l'objet n'est pas une liste bien formée. Sinon, si à la fin de l'objet votre compteur est égal à 0, alors l'objet est bel et bien une liste des plus correctes.

Voici quelques exemples d'application de cet algorithme :

- L'objet **(((AHA)) (ENCORE UNE LISTE))** peut être étiqueté comme suit :

  $$(_1(_2(_3\mathbf{AHA})_3)_2\ (_2\mathbf{ENCORE\ UNE\ LISTE})_2)_1$$

  La dernière parenthèse est étiquetée '1' : cet objet est donc bien une liste.

- Essayons pour l'objet que voici : **(HEIN? (QUOI?)))**

  $$(_1\mathbf{HEIN?}\ (_2\mathbf{QUOI?})_2)_1)_0$$

  Ici, il y a une parenthèse fermante en trop !

- finalement : **(((X + Y) + Z) -> (X + (Y + Z)))** donne

  $$(_1(_2(_3\mathbf{X} + \mathbf{Y})_3 + \mathbf{Z})_2 \text{ -> } (_2\mathbf{X} + (_3\mathbf{Y} + \mathbf{Z})_3)_2)_1$$

  Alors ? Est-ce une liste ?

Cette manière de numéroter les parenthèses permet également de reconnaître aisément les différents éléments d'une liste, les différents éléments des sous-listes, des sous-sous-listes, etc.

Prenons, par exemple la dernière liste :

$$(_1(_2(_3X + Y)_3 + Z)_2 \text{ -> } (_2X + (_3Y + Z)_3)_2)_1$$

C'est une liste à 3 éléments : les deux sous-listes entourées par les parenthèses étiquetées '2' et l'atome '->'. Pour être plus précis, le numérotage des parenthèses correspond à la *profondeur* de la liste : la liste $(X + Y)$ se trouve à la profondeur 2, la liste $(X + (Y + Z))$ se trouve à la profondeur 1, etc. Une liste entourée par des parenthèses numérotées x se trouve à la profondeur x-*1*, et tous les éléments à l'intérieur de cette liste se trouvent à la profondeur x.

1.1 EXERCICES

1. Déterminez pour chacun des objets ci-dessous s'il s'agit d'une liste, d'un atome ou d'un objet inadmissible en LISP :

   123
   (EIN (SCHOENES) BUCH)
   (AHA (ETONNANT . . . . !))
   (((((1) 2) 3) 4) 5)
   -3Aiii
   T
   (((((ARRRGH)))))

2. Déterminez le nombre d'éléments des listes suivantes :

   (EIN (SCHOENES) BUCH)
   (((((1) 2) 3) 4) 5)
   (((((ARRRGH)))))
   (UNIX (IS A) TRADEMARK (OF) BELL LABS)

3. Pour chacune des listes de l'exercice 2, quel est l'élément de profondeur maximale et sa profondeur.

2. LES FONCTIONS DE BASE : QUOTE, CAR, CDR, CONS

Les listes et les atomes sont donc les objets sur lesquels vous travaillez en LISP. Il existe des listes spéciales, nommées des *formes*, qui servent à indiquer à la machine-LISP qu'elle doit faire quelque chose. "Faire quelque chose" se dit, en LISP, *appeler une fonction*. Voyons dans l'ordre : d'abord la définition d'une forme :

forme ::= (*nom-de-fonction* {argument$_1$ argument$_2$ ... argument$_n$})

Un *appel de fonction* ou une *évaluation* d'une forme *ramène* (ou *retourne*) une valeur. Toute fonction LISP ramène une valeur quand on l'appelle.

Voici la définition d'une fonction TRES importante, la fonction **QUOTE** :

(**QUOTE** arg_1) → arg_1

Elle sert à dire à la machine sur quel objet on veut travailler; elle ramène *en valeur* l'objet LISP donné en argument (de la fonction) tel quel. *Ramène en valeur* sera noté avec le signe " → ".

Quelques exemples d'appels de la fonction **QUOTE** :

(QUOTE TIENS)	→	TIENS
(QUOTE (A B C))	→	(A B C)
(QUOTE (THE (PUMPKIN (EATER))))	→	(THE (PUMPKIN (EATER)))

La fonction **QUOTE** est d'une utilité extrême dans la programmation en LISP : puisque la majorité des fonctions LISP évalue ses arguments, nous utilisons la fonction **QUOTE** pour les arguments que nous ne voulons pas évaluer. L'évaluation des arguments des fonctions peut ainsi être réalisée sans crainte : **QUOTE** nous ramène l'argument *tel quel*.

La fonction **QUOTE** est tellement importante et tellement souvent utilisée, qu'il existe une notation abrégée :

'TIENS	→	TIENS
'(A B C)	→	(A B C)
'(THE (PUMPKIN (EATER)))	→	(THE (PUMPKIN (EATER)))

Cette notation n'est qu'une abréviation : la machine, elle, comprend toujours la même chose, c.à.d. un appel de la fonction **QUOTE**. La fonction **QUOTE** peut être comprise comme la fonction d'identité.

Puisqu'on a des listes, et puisqu'on peut énumérer les éléments d'une liste, il existe des fonctions pour accéder aux différents éléments d'une liste. D'abord la fonction **CAR** :[1]

(**CAR** *arg*) → le premier élément de la liste *arg* donnée en argument.
arg doit obligatoirement être une liste

voici quelques exemples d'appels :

```
(CAR '(A B C))                          →  A
notez que l'argument a été QUOTE ! pourquoi ?
(CAR '(THE (PUMPKIN (EATER))))          →  THE
(CAR '(((O C N H)) P S))                →  ((O C N H))[3]
```

naturellement, les fonctions LISP peuvent être composées :

```
(CAR (CAR '(((O C N H)) P S)))          →  (O C N H)
(CAR (CAR (CAR '(((O C N H)) P S))))    →  O
```

d'ailleurs :

```
(CAR '(CAR '(((O C N H)) P S)))         →  CAR
```

Voyez-vous maintenant l'utilité de la fonction **QUOTE** ?

La fonction **CDR** peut être définie comme suit :

(**CDR** *arg*) → la liste *arg* donnée en argument sans le premier élément.
arg doit obligatoirement être une liste

CDR est la fonction complémentaire de la fonction **CAR**. Voici quelques exemples :

```
(CDR '(A B C))                          →  (B C)
(CDR '(THE (PUMPKIN (EATER))))          →  ((PUMPKIN (EATER)))
(CDR '(((O C N H)) P S))                →  (P S)
```

et en combinant des fonctions :

```
(CDR (CDR '(A B C)))                    →  (C)
(CAR (CDR '(A B C)))                    →  B
(CAR (CDR (CDR '(A B C))))              →  C
```

Certaines combinaisons des fonctions **CAR** et **CDR** sont tellement utiles, qu'il en existe des

1. Ce nom barbare ne subsiste que pour des raisons historiques : la première implémentation de LISP se faisait sur un ordinateur IBM-704. Un mot de cette machine etait divisé en une partie "adressse" et une partie "décrement". Le Contenue de la partie Adresse d'un Registre livrait alors le **CAR** d'une liste et le Contenu de la partie Décrement d'un Registre livrait le **CDR** d'une liste. En honneur de cette première implémentation de LISP, toutes les versions suivantes ont gardé ces deux noms.

3. O, C, N, H, P et S sont – bien entendu – les éléments chimiques nécessaires à l'emergence de la vie.

écritures abrégées :

(**CAR** (**CDR** liste)) est la même chose que (**CADR** liste)
(**CAR** (**CDR** (**CDR** liste))) est la même chose que (**CADDR** liste)

La fonction **CADR** ramène le deuxième élément de sa liste argument, et la fonction **CADDR** ramène le troisième élément.

Vérifiez vous-même qu'avec des combinaisons adéquates des fonctions **CAR** et **CDR** vous pouvez ramener *n'importe* quel élément de *n'importe* quelle liste !

Jusqu'à présent nous connaîssons une fonction ramenant son argument tel quel (**QUOTE**), une fonction qui ramène le premier élément d'une liste (**CAR**) et une (**CDR**) qui ramène le reste d'une liste, c'est-à-dire : la liste sans son premier élément. Il nous faut aussi une fonction pour *construire* une nouvelle liste, c'est la fonction **CONS**, définie comme :

(**CONS** *argument*$_1$ *liste*) → la liste *liste* avec la valeur de *argument*$_1$ comme nouveau premier élément

exemples :

```
(CONS 'A '(B C))                          →  (A B C)
(CONS '(A B) '(C D))                      →  ((A B) C D)
(CONS (CAR '(A B C))(CDR '(A B C)))       →  (A B C)        ;!!;
(CAR (CONS 'A '(B C)))                    →  A
(CDR (CONS 'A '(B C)))                    →  (B C)
```

Pour terminer cette première partie, voici un instantané d'une interaction avec la machine. La machine écrit un "?" quand elle attend que vous entriez quelque chose. Ce caractère s'appelle le *caractère de prompt*. Elle imprime la valeur de ce que vous avez demandé, précédée du signe "=", sur la ligne suivante.) :[4]

```
?  'A
=  A

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

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

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

4. Le caractère de prompt peut varier d'une implémentation à l'autre. Souvent c'est le signe '>' ou le signe '→' au lieu du point d'interrogation, et le signe '=' est souvent omis.

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

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

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

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

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

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

Remarque importante :

Par définition des fonctions **CAR** et **CDR**, le **CAR** de **NIL** est égal à **NIL**, et le **CDR** de **NIL** est également égal à **NIL**. Vous avez donc les relations suivantes :

```
(CAR NIL)  →  ()
(CAR ())   →  ()
(CDR NIL)  →  ()
(CDR ())   →  ()
```

Remarquez également :

(CONS () '(A B C)) = (CONS NIL '(A B C)) → (() A B C)

Bien évidemment, faire un **CONS** avec **NIL** et une liste revient à insérer une liste vide en tête de la liste donnée en deuxième argument. **NIL** en tant que deuxième argument d'un **CONS** correspond à mettre une paire de parenthèses autour du premier argument du **CONS** :

(CONS 'A NIL) = (CONS 'A ()) → (A)

2.1 EXERCICES

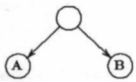
1. Donnez les résultats des appels des fonctions suivantes :

   a. (CAR '((A (B C)) D (E F))) → ?
   b. (CDR '((A (B C)) D (E F))) → ?
   c. (CADR (CAR '((A (B C)) D (E F)))) → ?
   d. (CADDR '((A (B C)) D (E F))) → ?
   e. (CONS 'NOBODY (CONS 'IS '(PERFECT))) → ?
   f. (CONS (CAR '((CAR A) (CDR A))) (CAR '(((CONS A B))))) → ?

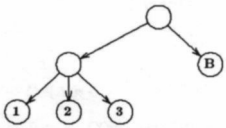
3. UN AUTRE POINT DE VUE SUR LES LISTES

Si vous avez encore des problèmes avec la notation des listes, imaginez les comme des *arbres* : vous descendez d'un niveau chaque fois que vous rencontrez une parenthèse ouvrante, et vous remontez d'un niveau lorsque vous rencontrez une parenthèse fermante. Voici quelques exemples de listes avec leur équivalent en forme d'arbre :

La liste **(A B)** peut être représentée comme :

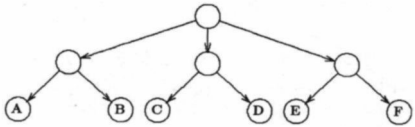


La liste **((1 2 3) B)** peut être représentée comme :

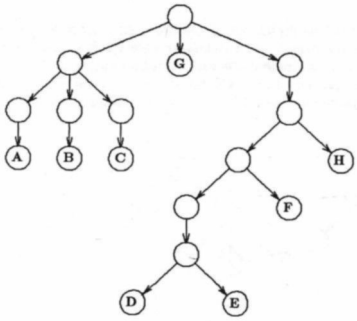


Cette représentation permet de reconnaître visuellement à la fois les éléments et leur profondeur. Une liste (ou sous-liste) est représentée comme un cercle vide. Les éléments d'une liste sont les cercles directement connectés au cercle de la liste, et la profondeur d'un élément se compte simplement par le nombre des flèches qu'il faut suivre en partant du sommet pour arriver à l'élément.

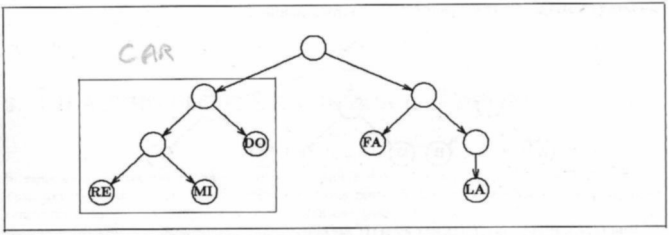
La liste **((A B) (C D) (E F))** peut être représentée comme :



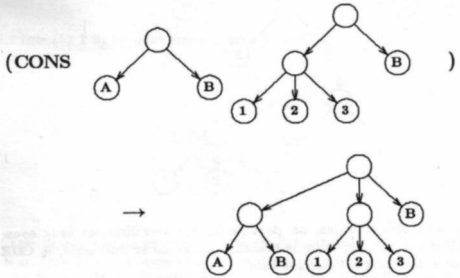
La liste **(((A)(B)(C)) G (((((D E)) F) H)))** peut être représentée comme :



Si l'on représente les listes sous forme d'arbres, on peut facilement visualiser les deux sous-parties **CAR** et **CDR**. Le **CAR** d'une liste est toute la branche gauche de la liste-arbre, le **CDR** d'une liste est l'arbre *sans* la branche gauche. Voici une liste :



Cet arbre est équivalent à la liste **(((RE MI) DO)(FA (LA)))**. Le **CAR** de cette liste correspond à la branche gauche contenue dans la boite intérieure et le **CDR** à l'arbre qui reste si l'on enlève de la boite extérieure la boite intérieure, c'est-à-dire si l'on enlève de l'arbre complet la branche gauche. Ainsi, il est clair que l'opération **CONS** sur des listes-arbres, correspond à l'insertion d'une nouvelle branche gauche dans un arbre, comme ci-dessous :



3.1 EXERCICES

1. Donnez les combinaisons de **CAR** et **CDR** nécessaires pour remplacer le signe "?" dans les appels de fonctions suivants

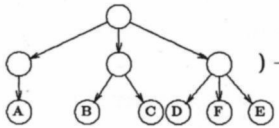
```
(? '(A B C D))                  →  D
(? '((A (B C)) E))              →  C
(? '(((DIEU) ENCORE) UNE))      →  DIEU
(? '(((DIEU) ENCORE) UNE))      →  ENCORE
```

2. Traduisez les listes du premier exercice sous forme d'arbres

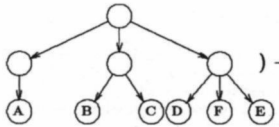
3. Que font les appels de fonctions suivants :

```
(CADR (CDR (CDR (CDR '(DO RE MI FA SOL LA SI)))))      →  ?
(CONS (CADR '((A B)(C D))) (CDDR '(A (B (C)))))        →  ?
(CONS (CONS 'HELLO NIL) '(HOW ARE YOU))                →  ?
(CONS 'JE (CONS 'JE (CONS 'JE (CONS 'BALBUTIE NIL))))  →  ?
(CADR (CONS 'TIENS (CONS '(C EST SIMPLE) ())))         →  ?
```

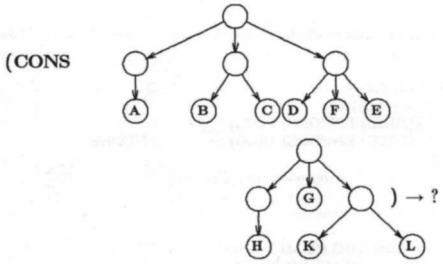
4. Donnez l'arbre résultant des opérations suivantes :

(CAR 

) → ?

(CDR 

) → ?



5. Traduisez les arbres de l'exercice précédent sous forme de listes.

((A)(BC)(DFE))

((H)G(KL)

4. LA DEFINITION DE FONCTIONS

Nous avons vu qu'il est possible de construire une liste grâce à la fonction **CONS**, de prendre le premier élément d'une liste grâce à la fonction **CAR**, de récupérer la liste sans son premier élément, grâce à la fonction **CDR**, et, par des combinaisons de ces fonctions, d'accéder à n'importe quel élément d'une liste et de combiner des éléments arbitraires de listes quelconques.

Maintenant imaginons que nous ayons un programme (un programme est, pour nous, un ensemble d'appels de fonctions) dans lequel nous ayons à calculer à plusieurs reprises le quatrième élément d'une liste, comme par exemple dans

```
(CAR (CDR (CDR (CDR '(A B C D E)))))
ou
(CAR (CDR (CDR (CDR '(1 2 3 4 5)))))
ou
(CAR (CDR (CDR (CDR '((0) (1) (1 0)(1 1) (1 0 0)(1 0 1)(1 1 0)(1 1 1))))))
```

Clairement, après un certain temps il devient lassant de taper et retaper la même suite d'appels **(CAR (CDR (CDR (CDR** . . . On aimerait bien, dans de tels cas, avoir la possibilité d'abréger cette suite d'appels : par exemple, pouvoir écrire

```
(4-IEME '(A B C D E))
```

au lieu de

```
(CAR (CDR (CDR (CDR '(A B C D E)))))
```

LISP permet de résoudre ce problème, et propose comme solution la possibilité de *définir* de nouvelles fonctions qui, une fois définies, ne pourront pas être distinguées des fonctions déjà existantes. Nous nommerons les fonctions existantes des *fonctions-standard*, et celles que l'utilisateur de LISP définit lui-même des *fonctions-utilisateurs*.

Mais regardons d'abord comment on peut définir de nouvelles fonctions. Voici un exemple de définition de fonction :

```
(DEFUN 4-IEME (L)
       (CAR (CDR (CDR (CDR L)))))
```

DEFUN est une fonction standard indiquant à la machine qu'on est en train de *définir* une nouvelle fonction. Naturellement, pour pouvoir ensuite s'en servir il faut lui associer un *nom* pour pouvoir s'y référer par la suite. Ce nom peut être quelconque, mais il est préférable de

choisir un nom mnémonique, c'est-à-dire un nom qui rappelle ce que la fonction doit faire. Dans l'exemple donné, nous avons choisi le nom **4-IEME**, qui dit bien ce que la fonction est censée faire : chercher le 4-ième élément d'une liste. Puisqu'il s'agit d'une liste il est nécessaire de donner un *argument* à la fonction. Un seul argument ici, puisque on ne veut calculer que le 4-ième élément d'*une* liste à la fois. Dans l'exemple, **L** est le nom de cet argument. C'est un paramètre de la fonction, c-à-d. le nom d'une *variable*. La deuxième ligne de la définition de la fonction utilisateur **4-IEME** indique ce que la fonction doit faire : calculer le **CAR** du **CDR** du **CDR** du **CDR** de son argument **L**.

Après cet exemple informel, regardons la définition syntaxique de la fonction standard **DEFUN** :[1]

(**DEFUN** *nom-de-la-fonction* ({var_1 var_2 ... var_n}) *corps-de-la-fonction*)

avec :

nom-de-la-fonction
: devrait être un nom nouveau, qui n'existe pas encore dans la machine. De préférence ne prenez pas alors des noms comme **CAR**, **CDR** ou **CONS**, ou plus généralement, des noms de fonctions standard, puisque si vous définissez une fonction du nom d'une des fonctions standard, vous perdez sa définition originale.

var_1, var_2 etc
: sont les noms des paramètres. Là aussi, choisissez de préférence des noms mnémoniques.

corps-de-la-fonction
: est la suite d'instructions décrivant le calcul que la fonction est censée réaliser quand on l'appelle.

Le premier argument de la fonction **DEFUN** est donc le nom de la fonction à définir, le deuxième argument est une liste de paramètres (vous avez remarqué les parenthèses dans la définition syntaxique de **DEFUN** ?!) Le reste de la définition d'une fonction est son corps, une suite d'appels de fonctions LISP standard ou de fonctions utilisateurs (dont vous devez également donner à LISP la définition avant une exécution d'un appel de la fonction). La valeur ramenée par une définition de fonction (par l'appel de la fonction **DEFUN**) est le *nom* de la fonction que vous avez définie. Ici dans la définition de la fonction **4-IEME** c'est donc l'atome **4-IEME** qui sera ramené en valeur. C'est une indication du langage LISP, signifiant qu'elle a enregistré votre définition et que vous pouvez vous en servir par la suite.

Une fois la fonction *définie*, on l'*appelle* (pour la tester ou pour l'exécuter). Un appel de la fonction **4-IEME** s'écrit comme suit :

(4-IEME '(A B C D E F))

1. En Le_LISP, la fonction permettant de définir des fonctions s'appelle **DE**. Sa définition est identique à celle de la fonction **DEFUN** de CommonLISP.

La valeur ramenée par un appel de fonction utilisateur est la valeur ramenée par l'évaluation du corps de la fonction *avec les variables paramètres liées aux valeurs des arguments de l'appel.* L'argument de l'appel exemple ci-dessus est **(QUOTE (A B C D E F))**. Le paramètre **L** de la fonction **4-IEME** sera donc lié à la liste **(A B C D E F)**, la valeur de l'évaluation de l'appel de la fonction **QUOTE** avec l'argument **(A B C D E F)**.

Cela peut paraître un peu lourd (par écrit !); mais, il est très important de distinguer entre la *définition* d'une fonction et son *appel.*

Le résultat de l'appel :

(4-IEME '(A B C D E F))

sera donc la valeur de l'expression :

(CAR (CDR (CDR (CDR L))))

avec **L** lié à la liste **(A B C D E F)**, ceci est équivalent à évaluer l'expression

(CAR (CDR (CDR (CDR '(A B C D E F)))))

Donc :

(4-IEME '(A B C D E F)) → D

Voici deux autres appels ainsi que le résultat de ces appels :

```
(4-IEME '(1 2 3 4 5))                          →   4
(4-IEME '((0) (1) (1 0)(1 1)(1 0 0)(1 0 1)(1 1 0)(1 1 1)))   →   (1 1)
```

Voici maintenant une fonction qui construit une liste de ses trois arguments :

```
(DEFUN LISTE3 (ARG1 ARG2 ARG3)
    (CONS ARG1 (CONS ARG2 (CONS ARG3 () ))))
```

et voici un appel de cette fonction :

(LISTE3 1 2 3) → (1 2 3)

Rappelons que l'évaluation de cette fonction se fait dans l'ordre suivant :

- d'abord la variable **ARG1** est liée à la valeur **1**
- ensuite la variable **ARG2** est liée à la valeur **2**
- ensuite la variable **ARG3** est liée à la valeur **3**

- ensuite le corps de la fonction est exécuté. Chaque fois qu'il y a une référence à une des trois variables **ARG1**, **ARG2** ou **ARG3**, le langage LISP calcule leur valeur respective. (La valeur d'une variable est la valeur à laquelle elle est liée.)

et voici encore deux appels de cette fonction :

```
(LISTE3 'TIENS 'TIENS 'TIENS)          →  (TIENS TIENS TIENS)
(LISTE3 'ENCORE '(UNE LISTE) 'OK?)     →  (ENCORE (UNE LISTE) OK?)
```

Que fait alors la fonction suivante :

```
(DEFUN TRUC (QQC)
       (CONS QQC (CONS QQC (CONS QQC ()))))
```

voici un appel :

```
(TRUC 'HMMM)  → ?
```

Examinons cela pas à pas : il s'agit d'une fonction, de nom **TRUC**, qui prend à l'appel un argument qui sera lié à la variable appelée **QQC**. Donc à l'appel de la fonction **TRUC**, LISP lie la variable **QQC** à l'argument **HMMM**, qui est la valeur de l'expression **(QUOTE HMMM)**. Ensuite, toute occurrence de la variable **QQC** à l'intérieur du corps de la fonction sera remplacée par cette valeur. Ce qui donne la même chose que si on avait écrit :

```
(CONS 'HMMM (CONS 'HMMM (CONS 'HMMM ())))
```

Ensuite, il ne reste qu'à évaluer cette expression. Comme toujours, LISP évalue d'abord les arguments de la fonction pour pouvoir calculer ensuite la valeur de la fonction sur ces arguments. Détaillons :

1. La première chose à faire est d'évaluer les deux arguments du premier **CONS** : l'expression **'HMMM** est le premier argument et **(CONS 'HMMM (CONS 'HMMM ()))** le deuxième. L'évaluation de **'HMMM** donne l'atome **HMMM** tel quel. Cet atome sera le nouveau premier élément de la liste résultant de l'évaluation de **(CONS 'HMMM (CONS 'HMMM ()))**.

2. Pour calculer cette liste, nous devons donc faire un **CONS** de **'HMMM** et de **(CONS 'HMMM ()))**. Comme précédemment, la valeur de l'expression **(QUOTE HMMM)** est l'atome **HMMM**. Cet atome sera le nouveau premier élément de la liste résultant de l'évaluation de **(CONS 'HMMM ()))**.

3. Ensuite il suffit de construire une paire de parenthèses autour de l'atome **HMMM**, ce qui donnera la liste **(HMMM)**.

4. Maintenant, enfin, il est possible de calculer le **CONS** que nous avions laissé en suspens en '2'. Il faut donc calculer le résultat du **CONS** de **HMMM** et de **(HMMM)**. Ce qui donne la liste **(HMMM HMMM)**.

5. En '1' il reste encore un **CONS** à calculer : le **CONS** de l'atome **HMMM** et de cette liste **(HMMM HMMM)**. Le résultat final est donc la liste

(HMMM HMMM HMMM)

La fonction **TRUC** nous retourne donc une liste avec trois occurrences de l'élément passé en argument ! C'est très puissant : après avoir défini une fonction il est possible de s'en servir de la même manière que des fonctions standard.

S'il subsiste encore des problèmes, relisez ce paragraphe (il est *très* important pour la suite !), et faites les exercices qui suivent.

4.1 EXERCICES

1. Voici la définition d'une fonction très intéressante :

```
(DEFUN TRUC1 (L1 L2)
   (CONS
      (CONS (CAR L1)(CDR L2))
      (CONS (CAR L2)(CDR L1))))
```

que fait-elle pour les appels suivants :

```
(TRUC1 '(A 2 3) '(1 B C))              →  ?
(TRUC1 '(JA CA VA) '(OUI ES GEHT))     →  ?
(TRUC1 '(A VOITURE) '(UNE CAR))        →  ?
```

2. Que sont les résultats des appels de fonctions suivants ?

```
(TRUC (4-IEME '(DO RE DO MI DO FA)))                    →  ?
(TRUC1 (LISTE3 'UN 'CUBE 'ROUGE)
   (TRUC1 (LISTE3 'SUR 'LA 'TABLE)(TRUC 'BRAVO)))       →  ?
```

3. Ecrivez une fonction sans argument qui ramène à l'appel la liste :

(BONJOUR)

4. Ecrivez une fonction à un argument qui ramène une liste contenant 4 occurrences de cet argument

5. Ecrivez une fonction à trois arguments qui ramène une liste contenant les trois arguments dans l'ordre inverse de l'appel. Exemple : si la fonction s'appelle **REVERSE3** un appel possible pourrait être :

(REVERSE3 'DO 'RE 'MI)

et la valeur ramenée serait alors la liste **(MI RE DO)**.

5. DES PREDICATS ET DE LA SELECTION

Avec les fonctions dont nous disposons, il est déjà possible d'écrire un grand nombre de programmes utiles. Reste toutefois que tous ces programmes sont *linéaires*. Par cela, nous entendons qu'ils se constituent de suites d'instructions (de suites d'appels de fonctions) qui seront exécutées l'une après l'autre, sans possibilité de faire des tests ou des répétitions. Dans ce chapitre des fonctions particulières seront introduites, les *prédicats*, et une fonction de *sélection* ou de *test* qui se sert de ces prédicats.

Rappelons que des programmes peuvent être considérés comme des descriptions d'activités, comme des recettes de cuisine. Naturellement, comme dans les recettes de cuisine, il doit y avoir des possibilités de tester l'état du monde, par exemple il faut une possibilité de tester si l'eau est bouillante ou pas. Des fonctions de test sont, en logique, appelées des *prédicats*. Ce sont des fonctions posant des questions auxquelles la réponse est soit oui, soit non. Elles testent la vérification d'une condition, et peuvent donc ramener *deux* valeurs possibles : *oui* ou *non*, comme à la question "est-ce que l'eau est bouillante ?" vous ne pouvez répondre que par oui ou par non (des réponses du style "pas encore" ne sont pas admises dans notre jeu), la machine ne peut, à l'évaluation d'un prédicat, que répondre par oui ou par non, *vrai* ou *faux*.

Non se dit en LISP **NIL** ou (), et *oui* se dit **T**. **T** est une abréviation de True, mot anglais pour *vrai*. Nil signifie en Latin et en 'colloquial english' *rien*, faux, nul. Cette association entre le mot 'faux' et le mot 'rien' se retrouve en LISP : rappelons que **NIL** est également un mot pour désigner la liste vide (). **NIL** a donc en LISP plusieurs rôles :

- c'est un atome. En LISP, **NIL** est un atome, mais pas une variable : on ne peut pas lui associer une valeur.
- c'est un nom pour la liste vide (la liste à 0 élément)
- et c'est le mot LISP pour *logiquement faux*.

5.1 QUELQUES PREDICATS

Voici la définition syntaxique d'un premier prédicat - la fonction **ATOM** :

(**ATOM** *arg*) → () si *arg* n'est pas un atome
→ **T** si l'argument est un atome

Cette fonction teste donc si la valeur de son argument est un *atome*. Voici quelques exemples :

```
(ATOM 'BRAVO)                    →  T
(ATOM NIL)                       →  T
                                    remarquez qu'il ne faut pas quoter
                                    l'atome NIL. De même, il ne faut
                                    pas quoter l'atome T et les nom-
                                    bres. On dit que de tels atomes
                                    sont des constantes
(ATOM ())                        →  T  ; par définition !!!! ;
(ATOM '(A B))                    →  ()  ; c'est une liste ! ;
(ATOM 1024)                      →  T
(ATOM '(GIRLS (ARE (COSIER))))   →  ()
```

On a un prédicat qui teste si son argument est un atome, mais naturellement il existe également des prédicats testant les autres types d'objets LISP. Ainsi le prédicat **NUMBERP** teste si son argument est un *nombre* et le prédicat **CONSP** teste si son argument est une *liste*. Voici leurs définitions syntaxiques et quelques exemples d'appels :

```
                    →  NIL si arg n'est pas une liste
(CONSP arg)
                    →  T si l'argument est une liste
```

quelques exemples de **CONSP** :

```
(CONSP '(A B C))      →  T
(CONSP 'BRAVO)        →  NIL
(CONSP ())            →  NIL   ; !!!! par définition !!!! ;
(CONSP '((A)(B C)))   →  T
(CONSP -1024)         →  NIL
```

et voici le prédicat **NUMBERP** :

```
                    →  NIL si arg n'est pas un nombre
(NUMBERP arg)
                    →  arg si l'argument est un nombre
```

Une première remarque s'impose : tout à l'heure il était signalé qu'en LISP *vrai* se dit **T**. Ce n'est pas tout à fait correct, il aurait fallu dire que *faux* se disait **NIL** en LISP, mais que LISP considère toute expression différente de **NIL** comme équivalente à **T**, c'est-à-dire *vrai*. C'est pourquoi dans la fonction **NUMBERP** le résultat peut être soit **NIL**, si l'argument n'est pas un nombre, soit, si l'argument est effectivement un nombre, la valeur de l'argument même. En Le_LISP, tous les prédicats qui permettent de ramener comme valeur *vrai* l'argument de l'appel même, vont adhérer à cette convention qui est fort utile si l'on veut utiliser, dans la suite du programme, la valeur de l'argument. En CommonLISP, la convention est que tout prédicat ramène soit l'atome **T**, si le prédicat est satisfait, soit la valeur **NIL**. Dans ce livre nous n'allons jamais utiliser cette possibilité de Le_LISP. Néanmoins, dans les exemples suivants, nous montrerons – de temps à autre – les résultats Le_LISP, juste pour s'habituer au fait que *tout* ce qui est différent de **NIL** est considéré comme logiquement vrai.

Nous y reviendrons à l'occassion. D'ailleurs, cette convention n'est pas possible pour la fonction **ATOM**, puisque l'appel :

(ATOM NIL)

doit ramener en valeur *vrai*, c-à-d. **T**. Elle ne peut décemment pas, dans ce cas, ramener la valeur de l'argument, qui est **NIL**, donc la valeur LISP disant 'logiquement faux'.

Voici quelques exemples d'appel de la fonction **NUMBERP** :

```
(NUMBERP 0)          →  T
(NUMBERP -110)       →  T
(NUMBERP NIL)        →  ()
(NUMBERP 'HMMM)      →  ()
(NUMBERP 3214)       →  T
(NUMBERP '(DO RE))   →  ()
```

Donnons tout de suite un prédicat supplémentaire : la fonction **NULL**. Cette fonction teste si son argument est une liste vide ou non. Voici sa définition :

(**NULL** *arg*)
- → **NIL** si *arg* n'est pas la liste vide
- → **T** si l'argument est la liste vide

NULL peut être traduite par la question "est-il vrai que la valeur de l'argument est égale à **NIL** ou () ?". Voici quelques exemples :

```
(NULL ())            →  T
(NULL NIL)           →  T
(NULL T)             →  ()
(NULL '(TIENS))      →  ()
(NULL 13)            →  ()
(NULL 'UN-ATOME)     →  ()
(NULL (NULL NIL))    →  ()
(NULL (NULL T))      →  T
```

Evidemment, puisque **NIL** est l'équivalent LISP du *faux* logique, ce prédicat peut être également traduit en "est-il vrai que la valeur de l'argument est faux ?" !

Ces prédicats peuvent être utilisés dans des fonctions comme toute autre fonction LISP. Ecrivons par exemple une fonction qui teste si le **CAR** (abus de langage ! **CAR** veut dire : le premier élément) d'une liste est un atome :

```
(DEFUN ATOM-CAR? (L)
       (ATOM (CAR L)))
```

Si nous donnons cette définition de fonction à la machine LISP, elle nous répondra en retournant l'atome **ATOM-CAR?**, indiquant ainsi que cette fonction lui est connue maintenant. On peut donc l'appeler ensuite :

```
(ATOM-CAR? '((DO RE)(RE DO)))          →  NIL
ou
(ATOM-CAR? '(DO RE RE DO))             →  T
ou encore
(ATOM-CAR? (CONS 'HELLO '(YOU THERE))) →  T
```

5.1.1 Exercices

1. Ecrivez une fonction **NUMBERP-CADR**, qui teste si le deuxième élément de la liste passée en argument est un nombre.

2. Ecrivez une fonction de nom **LISTE-CAR?** qui ramène la liste (**EST** x), avec x égal soit à **T** soit à **NIL**, suivant que le premier élément de sa liste argument est une liste ou non. Exemples d'appels :

```
(LISTE-CAR? '(A B C))     →  (EST NIL)
(LISTE-CAR? '((A) (B C))) →  (EST T)
(LISTE-CAR? '(AHA))       →  (EST NIL)
```

3. Quels sont les résultats de ces appels de fonctions :

```
(NUMBERP-CADR '(1 2 3))                  →  ?
(NUMBERP-CADR '(A -24 B))                →  ?
(NUMBERP-CADR '(256 PETITS BITS)         →  ?
(NUMBERP-CADR (CONS '1 '(1 2 3 5 8 13))) →  ?
```

avec **NUMBERP-CADR** définie comme :

```
(DEFUN NUMBERP-CADR (L)
       (NUMBERP (CADR (LISTE3 (CADDR L)(CAR L)(CADR L)))))
```

et **LISTE3** définie comme au chapitre précédent de cette introduction.

5.2 LA FONCTION IF

Revenons sur notre analogie de recettes de cuisine : si l'on teste la température de l'eau, c'est-à-dire si l'on se pose la question "l'eau est-elle bouillante ?", c'est probablement qu'on désire *agir différemment* suivant le résultat du test. On veut, par exemple, éteindre le feu ou verser l'eau dans la cafetière si l'eau est bouillante, et attendre que l'eau soit vraiment bouillante dans le cas contraire. Donc, on veut, suivant les résultats de l'application des tests, sélectionner entre différentes activités celle qui est adéquate.

En LISP, les prédicats servent exactement à la même chose : guider la suite de l'exécution du programme suivant les résultats de tests sur les données.

La fonction LISP responsable de ce *guidage* est la fonction **IF**. Donnons en tout de suite la définition syntaxique :[1]

(IF *test* *action-si-vrai*
{*action*$_1$*-si-faux*
action$_2$*-si-faux*
. . .
action$_n$*-si-faux*})

→ valeur de *action-si-vrai* si l'évaluation du test donne *vrai*

→ valeur de *action*$_n$*-si-faux* si l'évaluation du test donne *faux*

Cette définition peut se lire comme :

> **IF** est une fonction à nombre variable d'arguments. Le premier argument, *test*, est un prédicat. Le deuxième argument, *action-si-vrai*, est une expression LISP quelconque. Si le résultat de *test* est différent de **NIL** (donc, si le test donne le résultat *vrai*) la valeur ramenée du **IF** sera la valeur de l'expression *action-si-vrai*. Sinon, si le résultat de l'évaluation de *test* est égal à **NIL**, la machine LISP évalue séquentiellement une *action-si-faux* après l'autre et retourne la valeur de l'évaluation de *action*$_n$*-si-faux*.

Regardons un exemple. Voici un appel possible de **IF** :

(IF (NUMBERP 1) '(UN NOMBRE) '(PAS UN NOMBRE)) → (UN NOMBRE)

L'évaluation du test **(NUMBERP 1)** ramène l'atome **T** en valeur qui est bien évidemment différent de **NIL**. Donc le résultat de l'évaluation de *test* donne *vrai*. Ceci implique, après la définition de la fonction **IF**, que la valeur ramenée par le **IF** tout entier soit la valeur de *action-si-vrai*, donc la valeur de **'(UN NOMBRE)**, donc la liste **(UN NOMBRE)**. Si au lieu de donner le test **(NUMBERP 1)**, on avait donné à LISP l'appel suivant :

(IF (NUMBERP 'HELLO) '(UN NOMBRE) '(PAS UN NOMBRE)) → (PAS UN NOMBRE)

En effet, puisque l'atome **HELLO** n'est pas un nombre, la valeur ramenée est la liste **(PAS UN NOMBRE)**, la valeur de l'unique *action-si-faux*.

Donnons encore quelques exemples :

1. Quelques systèmes CommonLISP n'admettent qu'*une action unique* si l'évaluation de la partie *test* donne **NIL**.

```
(IF (NULL '(A B)) (CAR '(DO RE))(CDR '(DO RE)))      →  (RE)
(IF (CONSP (CONS NIL NIL)) (CONS NIL NIL) 'BRRR)     →  (NIL)
(IF NIL 1 2)                                         →  2
```

dans l'exemple précédent, le premier argument du **IF** n'est pas l'appel d'un prédicat. Toutefois, l'atome **NIL** se trouve en situation de prédicat, simplement par le fait qu'il se trouve dans la position de *test*. Le résultat de l'évaluation de *test* est donc égal à ().

Voici encore un exemple :

```
(IF '(A B C) 'AHA 'HMMM)   →   AHA
```

la même remarque que précédemment s'impose : la liste **(A B C)** n'est pas un prédicat en soi, mais de par sa situation en position de *test*, elle joue le rôle d'un *vrai* logique

Munis de cette fonction, nous pouvons commencer à écrire de vraies fonctions LISP. Construisons d'abord une nouvelle fonction **NUMBERP-CADR?** qui ramène la liste (*deuxième-élément* **EST UN NOMBRE**) si le deuxième élément de sa liste argument est un nombre, et qui, dans les autres cas, ramène la liste (deuxième-élément **N EST PAS UN NOMBRE**) :

```
(DEFUN NUMBERP-CADR? (L)
    (IF (NUMBERP (CADR L))
        (CONS (CADR L) '(EST UN NOMBRE))
        (CONS (CADR L) '(N EST PAS UN NOMBRE))))
```

et voici quelques appels de cette fonction :

```
(NUMBERP-CADR? '(1 2 3))      →  (2 EST UN NOMBRE)
(NUMBERP-CADR? '(DO RE MI))   →  (RE N EST PAS UN NOMBRE)
(NUMBERP-CADR? NIL)           →  (NIL N EST PAS UN NOMBRE)
```

encore quelques exemples :

```
(DEFUN CONS? (ELE L)
    (IF (CONSP L)
        (CONS ELE L)
        (CONS L '(N EST PAS UNE LISTE, MSIEUR))))
```

et voici quelques appels :

```
(CONS? 1 '(2 3 4))
        →  (1 2 3 4)
(CONS? 1 2)
        →  (2 N EST PAS UNE LISTE, MSIEUR)

(CONS? 'UNE (CONS? 'SOURIS (CONS? 'VERTE ())))
        →  (UNE SOURIS VERTE)

(CONS? 'UNE (CONS? 'SOURIS 'VERTE))
        →  (UNE VERTE N EST PAS UNE LISTE, MSIEUR)
```

Examinons la fonction suivante :

```
(DEFUN TYPE? (ARG)
   (IF (NUMBERP ARG) 'NOMBRE
      (IF (ATOM ARG) 'ATOME 'LISTE)))
```

Tous les arguments du **IF** pouvant être des expressions LISP quelconques, rien n'exclut que l'un ou plusieurs des arguments soient des appels de la fonction **IF** elle-même. De telles imbrications servent en général à distinguer entre plus de 2 possibilités. Ici, dans la fonction **TYPE?**, nous distinguons 3 cas possibles : soit l'argument donné à l'appel est un nombre, la machine répond alors avec la valeur **NOMBRE**, soit il est un atome, la machine ramène l'atome **ATOME**, soit l'argument n'est ni un nombre, ni un atome, alors la machine suppose que c'est une liste et ramène en valeur l'atome **LISTE**. Voici quelques appels :

```
(TYPE? '(((IRREN) IST) MENSCHLICH))   →  LISTE
(TYPE? '(ERRARE HUMANUM EST))         →  LISTE
(TYPE? (CAR '(1 + 2)))                →  NOMBRE
(TYPE? (CADR '(1 + 2)))               →  ATOME
(TYPE? '(CECI EST FAUX))              →  LISTE
```

5.2.1 **Exercices**

1. Ecrivez une fonction **3NOMBRES** qui ramène l'atome **BRAVO**, si les 3 premiers éléments de sa liste argument sont des nombres, sinon elle doit ramener l'atome **PERDANT**. exemple :

```
(3NOMBRES '(1 2 3))                          →  BRAVO
(3NOMBRES '(1 CAT))                          →  PERDANT
(3NOMBRES (CONS 1 (CONS -100 (CONS -1 ()))))  →  BRAVO
```

2. Ecrivez la fonction **REV** qui inverse une liste de 1, 2 ou 3 éléments. Exemples d'appels :

```
(REV '(DO RE MI))        →  (MI DO RE)
(REV '(EST IL FATIGUE))  →  (FATIGUE IL EST)
(REV '(JO GEHNS))        →  (GEHNS JO)
```

3. Voici une fonction bizarre :

```
(DEFUN BIZARRE (ARG1 ARG2)
   (IF (CONSP ARG2)
      (CONS ARG1 ARG2)
      (IF (CONSP ARG1)
         (CONS ARG2 ARG1)
         (IF (NULL ARG2)
            (IF (NULL ARG1) '(ALORS, BEN, QUOI)
                  (CONS ARG1 ARG2))
            (IF (NULL ARG1) (CONS ARG2 ARG1)
                  (CONS ARG1 (CONS ARG2 NIL)))))))
```

Que fait cette fonction ? Donnez les valeurs des appels suivants :

```
(BIZARRE ()())                                                    →  ?
(BIZARRE 1 '(2 3))                                                →  ?
(BIZARRE '(2 3) 1)                                                →  ?
(BIZARRE 'TIENS 'C-EST-TOI)                                       →  ?
(BIZARRE () (TYPE? 432))                                          →  ?
(BIZARRE (CDR (CONS? (TYPE? 'INTEL) 432)) (CAR '(DO RE MI)))      →  ?
```

Avant de continuer, assurez-vous que jusqu'ici vous avez bien compris. Si vous avez des problèmes, reprenez les exercices et les exemples. Et n'oubliez pas de les faire tourner sur une machine, vérifiez-les, modifiez-les légèrement et trouvez les erreurs.

6. LA REPETITION

6.1 LA FONCTION REVERSE

Tout langage de description de processus, donc tout langage de programmation, doit comporter des possibilités de décrire des activités répétitives. LISP exprime une répétition de manière particulièrement élégante, facile et efficace. Commençons par examiner l'exemple d'une fonction qui inverse les éléments d'une liste de longueur quelconque :

```
(DEFUN REVERSE (L RES)
   (IF (NULL L) RES
      (REVERSE (CDR L) (CONS (CAR L) RES))))
```

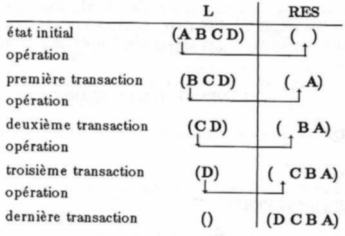
et voici quelques appels :

```
(REVERSE '(DO RE MI FA) NIL)                  →  (FA MI RE DO)
(REVERSE '(COLORLESS GREEN IDEAS) NIL)        →  (IDEAS GREEN COLORLESS)
(REVERSE '(A B (C D) E F) NIL)                →  (F E (C D) B A)
```

Pour comprendre l'évaluation d'un appel de cette fonction, donnons-en d'abord une description de son comportement : imaginons que nous ayons deux lieux, l'un nommé **L**, l'autre nommé **RES**, chacun dans un état initial :

L	RES
(A B C D)	()

Pour inverser la liste **(A B C D)** de **L**, on va transférer un élément après l'autre de **L** vers **RES** en introduisant à chaque instant l'élément sortant de **L** en tête de **RES**. Voici graphiquement une *trace* des états successifs de **L** et **RES** :



Le programme **REVERSE** se comporte exactement de la même façon : il transfère élément par élément de **L** vers l'accumulateur **RES**, et s'arrête quand la liste **L** est vide.

L'écriture de cette fonction peut être facilement comprise si l'on raisonne de la manière suivante : pour inverser une liste il faut d'abord vérifier si la liste est vide. Si oui, alors il n'y a rien à faire et le résultat sera la liste accumulée au fur et à mesure dans la variable **RES**. Ceci est naturellement vrai au début, puisque **RES** sera égal à **NIL**, la liste vide, au premier appel. La liste vide est bien l'inverse de la liste vide ! Ensuite, il faut prendre le premier élément de la liste **L** et le mettre en tête de la liste **RES**.

Reste ensuite à répeter la même chose avec les éléments restants, *jusqu'à ce que la liste* **L** *soit vide*. Si maintenant on trouvait une fonction pour faire ce travail, il suffirait de l'appeler. Rappelons-nous que nous avons une fonction qui satisfait à cette demande : la fonction **REVERSE** qu'on est en train de définir. Donc, appelons-la avec le reste de la liste **L** et l'accumulation, dans **RES**, du premier élément de **L** et de **RES**. Remarquons qu'il est nécessaire de modifier l'argument **L** de telle manière qu'il converge vers la satisfaction du test d'arrêt **(NULL L)**.

L'exécution de l'appel

(REVERSE '(DO RE MI) NIL)

se fait donc comme suit :

1. d'abord **L** est liée à la liste **(DO RE MI)** et **RES** à **NIL**. Ensuite le corps de la fonction est évalué avec ces liaisons. Première chose à faire : tester si **L** est vide. Pour l'instant ce n'est pas le cas. Il faut donc évaluer l'*action-si-faux* du **IF**, qui dit qu'il faut calculer **(REVERSE (CDR L)(CONS (CAR L) RES)**. Avec les liaisons valides pour l'instant, cela revient au même que de calculer

   (REVERSE '(RE MI) '(DO))

   Pour connaître le résultat de l'appel initial, il suffit donc de connaître le résultat de ce nouvel appel.

2. On entre alors de nouveau dans la fonction **REVERSE**, liant la variable **L** à la nouvelle valeur **(RE MI)** et la variable **RES** à **(DO)**. Comme **L** n'est toujours pas égale à **NIL**, il faut, pour connaître le résultat de cet appel, une fois de plus évaluer l'*action-si-faux* de la sélection. Cette fois ce sera

   (REVERSE '(MI) '(RE DO))

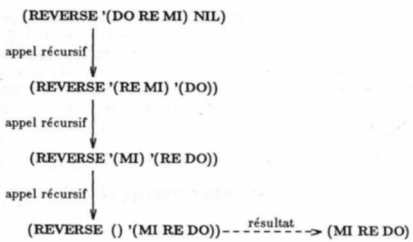
   c'est-à-dire le **CDR** de **L** et le résultat de **(CONS (CAR L) RES)** avec les liaisons actuelles.

3. la même opération se répète : nouvelles liaisons des variables, **L** n'étant toujours pas égale à **NIL** il faut re-appeler **REVERSE**

   (REVERSE () '(MI RE DO))

4. cette fois-ci la variable **L** est liée à **NIL** et le test **(NULL L)** est donc vérifié. Le résultat du dernier appel de **REVERSE** est donc la valeur de **RES**, la liste **(MI RE DO)**. Mais rappelons nous que cet appel n'avait lieu que pour connaitre le résultat de l'appel précédent, c'est donc également le résultat de l'appel précédent. Ainsi de suite, jusqu'à ce qu'on arrive à l'appel initial, et la machine peut ramener cette liste en valeur.

Graphiquement, on peut représenter l'évaluation de cet appel comme suit :



Des fonctions qui contiennent à l'intérieur de leur *corps* un appel à elles-mêmes sont dites des *fonctions-récursives*. La fonction **REVERSE** est donc une fonction récursive.

Voici une *trace* de l'exécution du même appel :

```
? (REVERSE '(DO RE MI) ())
  ---> REVERSE : ((DO RE MI) NIL)
    ---> REVERSE : ((RE MI) (DO))
      ---> REVERSE : ((MI) (RE DO))
        ---> REVERSE : (() (MI RE DO))
  <---------- REVERSE = (MI RE DO)
= (MI RE DO)
```

6.2 LA FONCTION APPEND

Prenons un deuxième exemple d'une fonction récursive : la fonction **APPEND** qui doit concaténer deux listes comme dans les exemples suivants :

```
(APPEND '(A B) '(C D))    →   (A B C D)
(APPEND '(A) '(B C))      →   (A B C)
(APPEND NIL '(A B C))     →   (A B C)
```

Cette fonction possède deux paramètres (pour les deux listes passées en argument) et elle retourne une liste qui est la concaténation de ces deux listes. Cela nous donne déjà le début de l'écriture de la fonction :

```
(DEFUN APPEND (LISTE1 LISTE2)
  . . .
```

Si nous regardons les exemples de la fonction **APPEND** (ces exemples peuvent être compris comme sa *spécification*), on observe que si le premier argument, **LISTE1**, est vide, le résultat est la liste passée en deuxième argument, c.à.d.: **LISTE2**, et si **LISTE1** a un seul élément, le résultat est la liste **LISTE2** avec comme nouveau premier élément, l'élément unique du premier argument. Malheureusement, nous savons déjà que nous ne pouvons pas *conser*[1] un élément après l'autre de la première liste avec la deuxième, puisque cette méthode était utilisée dans la fonction **REVERSE** : elle introduirait les éléments dans l'ordre inverse. Il faudrait pouvoir commencer avec le dernier élément de la liste **LISTE1**, l'introduire dans la liste **LISTE2**, pour ensuite y introduire l'avant dernier élément et ainsi de suite, jusqu'à ce que nous soyons au premier élément de **LISTE1**. On pourrait inverser d'abord la liste **LISTE1** et ensuite appliquer la fonction reverse avec comme arguments cette liste inversée et la liste **LISTE2**. Ce qui nous donne :

```
(DEFUN APPEND (LISTE1 LISTE2)
        (REVERSE (REVERSE LISTE1 NIL) LISTE2))
```

Mais, bien que l'écriture de cette fonction soit très brève, elle est terriblement inefficace, puisque la liste **LISTE1** est parcourue 2 (*deux* !) fois dans toute sa longueur. Imaginez le temps que cela prend si la liste est très, très longue.

1. du verbe "*conser*" (du premier groupe), qui se conjugue tout normalement : je conse, tu conses, .. etc et qui est synonyme de 'faire un **CONS**', ou 'allouer une cellule'.

On peut faire exactement la même chose plus efficacement, si on sépare le parcours de la liste **LISTE1** de la construction de la liste résultat : d'abord parcourons la liste premier argument, en mémorisant les différents éléments rencontrés, ensuite *consons*[2] chaque élément, en commençant par le dernier, à la liste **LISTE2**. Ce qui nous donne cette deuxième définition :

```
(DEFUN APPEND (LISTE1 LISTE2)
   (IF (NULL LISTE1) LISTE2
      (CONS (CAR LISTE1) (APPEND (CDR LISTE1) LISTE2))))
```

qui peut être paraphrasée comme suit :

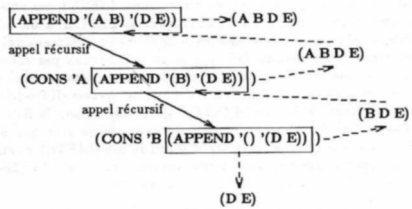
Pour concaténer deux listes, il faut distinguer deux cas :

1. si la première liste est vide, le résultat sera la deuxième liste
2. sinon, il suffit d'introduire le premier élément en tête de la liste résultant de la concaténation du reste de la première liste et de la deuxième liste, tout naturellement, puisqu'on avait déjà vu que la concaténation d'une liste à un seul élément est justement le résultat du **CONS** de cet unique élément et de la deuxième liste.

Graphiquement, l'exécution de l'appel

```
(APPEND '(A B) '(D E))
```

peut être représentée de la manière suivante :



On voit clairement qu'en *descendant* dans les appels récursifs successifs, la machine accumule des **CONS**'s qu'elle ne peut calculer qu'après avoir évalué les arguments (ici, un des arguments est toujours un appel récursif de **APPEND**). Le calcul des expressions **CONS** est effectué pendant la *remontée* de la récursion.

Voici une *trace* d'un appel de la fonction **APPEND** :

2. voir la note précédente.

```
? (APPEND '(A B C) '(D E F))                    ; l'appel initial ;
  ---> APPEND : ((A B C) (D E F))                    ; sa trace ;
    ---> APPEND : ((B C) (D E F))   ; le deuxième appel, premier appel récursif ;
      ---> APPEND : ((C) (D E F))                ; le troisième appel ;
        ---> APPEND : (NIL (D E F))              ; le quatrième appel ;
        <--- APPEND = (D E F)          ; le résultat du 4-ième appel ;
      <--- APPEND = (C D E F)          ; le résultat du 3-ième appel ;
    <--- APPEND = (B C D E F)          ; le résultat du 2-ième appel ;
  <--- APPEND = (A B C D E F)                ; le résultat du 1-er appel ;
= (A B C D E F)                                  ; le résultat final ;
```

En comparant les deux *traces* de **REVERSE** et de **APPEND**, nous constatons qu'il existe deux sortes de boucles récursives :

1. les boucles récursives qui font tout le calcul en *descendant*, c'est-à-dire : comme dans la fonction **REVERSE**, le calcul est effectué entièrement à travers les appels récursifs. Une fois arrivé au *test de sortie* de la récursivité (ici, le test est l'expression **(NULL L)**), le calcul est terminé et la fonction peut directement ramener cette valeur.

2. les boucles récursives qui font une partie du calcul en *descendant* les appels récursifs et l'autre partie en *remontant* les appels récursifs successifs. C'est le cas dans la fonction **APPEND** : en descendant, la fonction isole les différents éléments qu'elle *conse* ensuite, pendant la remontée, aux résultats des divers appels récursifs. Remarquez que les éléments isolés pendant la descente sont utilisés, pendant la remontée, dans l'ordre inverse, c'est-à-dire : le premier élément isolé sera le dernier élément utilisé.

Nous allons revenir plus loin sur la différence entre ces deux types de fonctions récursives. Pour l'instant, observons juste, que ces deux types sont aisés à distinguer dans l'écriture même des fonctions : dans le premier type (calcul exclusivement en descendant) le résultat de l'appel récursif n'est pas utilisé par une autre fonction, ou, plus précisément, l'appel récursif ne se trouve pas en position d'argument d'une fonction; dans le deuxième type (calcul en descendant et en remontant), le résultat est utilisé par une autre fonction. Dans la fonction **APPEND**, par exemple, le résultat de chaque appel récursif est utilisé comme deuxième argument d'un **CONS**.

Les appels qui n'engendrent un calcul qu'en descendant s'appellent des *appels récursifs terminaux*.

6.3 LA FONCTION EQUAL

Notre troisième exemple d'une fonction récursive sera la fonction **EQUAL**, une fonction qui doit tester l'égalité de ses deux arguments. A cette fin nous avons d'abord besoin d'apprendre un nouveau prédicat : le prédicat **EQ** qui teste l'égalité de deux *atomes*. Voici la définition de **EQ** :

(EQ *arg1 arg2*) → **T** si *arg1* et *arg2* sont le même atome

→ **NIL** si *arg1* et *arg2* sont des atomes différents ou si au moins l'un des deux est une liste

Ci-dessous quelques exemples d'utilisation de la fonction standard **EQ** :

```
(EQ 'LUC 'LUC)                              →  T
(EQ 'LUC 'DANIEL)                           →  NIL
(EQ (CAR '(DO RE))(CADR '(MI DO SOL)))      →  T
(EQ '(D P F T R) '(D P F T R))              →  NIL   ; !! ;[4]
```

Remarquons que **EQ** ramène **NIL**, donc *faux*, si les arguments sont des listes, même si les deux listes sont égales.[5] A priori, il n'existe pas de fonctions testant l'égalité de deux listes. D'autre part, nous savons que deux listes peuvent être considérées comme égales si (et seulement si) les éléments des deux listes sont égaux un à un. Tout ce que nous avons à faire alors, c'est comparer les éléments successifs des deux listes en y appliquant la fonction **EQ**. Ce raisonnement nous amène à la définition suivante :

```
(DEFUN EQUAL (ARG1 ARG2)
   (IF (ATOM ARG1)(EQ ARG1 ARG2)
      (IF (ATOM ARG2) NIL
         (IF (EQ (CAR ARG1)(CAR ARG2))
            (EQUAL (CDR ARG1)(CDR ARG2))
            NIL))))
```

Cette fonction teste aussi bien l'égalité de deux atomes (la première ligne dit que si **ARG1** est un atome, il suffit de tester son égalité – par un appel de **EQ** – avec **ARG2**) que l'égalité de deux listes (en testant d'abord l'égalité des deux premiers éléments et en répétant ce processus avec les **CDR**'s des deux listes).

Notons l'utilisation du prédicat **ATOM** : il sert à la fois à tester si l'on a affaire à un atome *et* de test d'arrêt de la récursivité. Ceci est possible puisque la liste vide est considérée comme l'atome **NIL**. Voici quelques exemples d'appels de cette fonction :

```
(EQUAL '(DO RE MI) '(DO RE MI))                    →  T
(EQUAL 'TIENS 'TIENS)                              →  T
(EQUAL '(DO RE MI) (CDR '(DO RE MI)))              →  NIL
(EQUAL '(LE NEZ (DE CLEOPATRE)) '(LE NEZ (DE CLEOPATRE)))  →  NIL[7]
```

Evidemment la fonction ne marche pas si les listes données en argument contiennent des sous-

4. "D.P.F.T.R." est un acronyme pour "Du Passé Faisons Table Rase".

5. L'exception à cette règle est discutée dans le chapitre 15.

listes. Regardez voici une trace de ce dernier exemple :

```
? (EQUAL '(LE NEZ (DE CLEOPATRE)) '(LE NEZ (DE CLEOPATRE)))
  ---> EQUAL : ((LE NEZ (DE CLEOPATRE)) (LE NEZ (DE CLEOPATRE)))
    ---> EQUAL : ((NEZ (DE CLEOPATRE)) (NEZ (DE CLEOPATRE)))
      ---> EQUAL : (((DE CLEOPATRE)) ((DE CLEOPATRE)))
  <--------- EQUAL = NIL
= NIL
```

Le test (EQ (CAR ARG1)(CAR ARG2)) ramène NIL dès que le premier élément est une liste ! Pour réparer cette insuffisance, il suffit de tester non pas l'égalité EQ, mais l'égalité EQUAL des CAR's successifs. Ce qui nous donne la version modifiée que voici :

```
(DEFUN EQUAL (ARG1 ARG2)
    (IF (ATOM ARG1)(EQ ARG1 ARG2)
        (IF (ATOM ARG2) NIL
            (IF (EQUAL (CAR ARG1)(CAR ARG2))
                (EQUAL (CDR ARG1)(CDR ARG2))
                NIL))))
```

et voici une trace de cette nouvelle fonction sur le même exemple :

```
? (EQUAL '(LE NEZ (DE CLEOPATRE)) '(LE NEZ (DE CLEOPATRE)))
  ---> EQUAL : ((LE NEZ (DE CLEOPATRE)) (LE NEZ (DE CLEOPATRE)))
    ---> EQUAL : (LE LE)
    <--- EQUAL = T
    ---> EQUAL : ((NEZ (DE CLEOPATRE)) (NEZ (DE CLEOPATRE)))
      ---> EQUAL : (NEZ NEZ)
      <--- EQUAL = T
      ---> EQUAL : (((DE CLEOPATRE)) ((DE CLEOPATRE)))
        ---> EQUAL : ((DE CLEOPATRE) (DE CLEOPATRE))
          ---> EQUAL : (DE DE)
          <--- EQUAL = T
          ---> EQUAL : ((CLEOPATRE) (CLEOPATRE))
            ---> EQUAL : (CLEOPATRE CLEOPATRE)
            <--- EQUAL = T
            ---> EQUAL : (NIL NIL)
        <--------- EQUAL = T
        ---> EQUAL : (NIL NIL)
  <------------ EQUAL = T
= T
```

Remarquez que cette fonction contient deux appels récursifs : le résultat du premier appel est utilisé comme test du IF, et le deuxième appel nous livre le résultat. Cette fonction contient donc un appel récursif terminal (le deuxième appel récursif) et un qui ne l'est pas (le premier appel récursif).

Une dernière remarque sur cette fonction : très souvent il peut arriver que vous ayez à écrire

7. pour en savoir plus sur "le nez de Cléopâtre" regardez donc l'excellent article de Raymond Queneau sur *la littérature définitionnelle* dans *Oulipo, la littérature potentielle*, pp. 119-122, idées/Gallimard.

une fonction avec un ensemble de IFs imbriqués, comme ici, où vous aviez trois IFs les uns dans les autres. Pour de tels cas il existe une abréviation, à la fois plus lisible et plus puissante : c'est la fonction **COND**, une fonction de sélection généralisée. Elle est syntaxiquement définie comme suit :

(**COND** $clause_1$ $clause_2$ ... $clause_n$)

et chacune des *clauses* doit être de la forme :

(*test* {$action_1$ $action_2$ ... $action_n$})

L'évaluation de la fonction **COND** procède comme ceci :

le *test* de la première clause ($clause_1$) est évalué

[1] si son évaluation ramène la valeur **NIL**
alors s'il y a encore des clauses
on continue avec l'évaluation du *test* de la clause suivante et on continue en [1]
sinon on arrête l'évaluation du **COND** en ramenant **NIL** en valeur
sinon on évalue en séquence les act_1 à act_n de la clause courante et on sort du **COND** en ramenant en valeur la valeur de l'évaluation de act_n. (s'il n'y a pas d'actions à évaluer on sort du **COND** en ramenant en valeur la valeur de l'évaluation de *test*)

[2] si aucun des *tests* n'est satisfait, la valeur du **COND** est **NIL**.

Ceci signifie que l'expression :

(**IF** $test_1$ act_1
(**IF** $test_2$ act_2
(**IF** $test_3$ act_3
...
(**IF** $test_n$ act_{n1} act_{n2} .. act_{nm}) ...)))

est équivalente à l'expression :

(**COND**
($test_1$ act_1)
($test_2$ act_2)
($test_3$ act_3)
...
($test_n$ act_{n1})
(**T** act_{n2} ... act_{nm}))

La structure du **COND** est beaucoup plus lisible que celle des **IF**'s imbriqués. Nous pouvons ainsi écrire la fonction **EQUAL** dans une forme plus élégante (mais faisant exactement la même chose) :

```
(DEFUN EQUAL (ARG1 ARG2)(COND
    ((ATOM ARG1)(EQ ARG1 ARG2))
    ((ATOM ARG2) NIL)
    ((EQUAL (CAR ARG1)(CAR ARG2))
        (EQUAL (CDR ARG1)(CDR ARG2))) ))
```

6.4 LA FONCTION DELETE

Etudions en détail un dernier exemple de fonction récursive. Cette fois le problème sera d'écrire une fonction qui enlève toutes les occurrences d'un élément donné d'une liste. Cette fonction, que nous nommerons **DELETE**, aura donc deux arguments : un premier argument donnant l'élément à éliminer, et un deuxième argument donnant la liste à l'intérieur de laquelle on veut éliminer cet élément. Sa construction sera tout à fait *immédiate*.

Rappelons que dans toutes les fonctions à structure répétitive, il existe *toujours* au moins un test d'arrêt et au moins un appel récursif à l'intérieur duquel au moins un des arguments doit être modifié d'une manière telle que les valeurs successives de cet argument convergent vers la satisfaction du test d'arrêt.

Pour pouvoir *enlever* d'une liste un élément donné, il faut naturellement d'abord *chercher* cet élément. Nous venons juste de voir un algorithme de recherche d'un élément dans la fonction **EQUAL**. Rien ne nous empêche de l'utiliser à nouveau. La seule différence ici est que nous ne nous n'arrêtons pas après avoir trouvé *une* occurrence de l'élément en question, mais que nous voulons en trouver toutes les occurrences. D'autre part, un algorithme uniquement de recherche n'est pas suffisant, puisque chaque fois que nous avons trouvé une occurrence nous voulons l'éliminer. De plus, nous voulons garder tous les autres éléments de la liste. De ces remarques s'ensuit pratiquement le programme : nous allons comparer chaque élément de la liste avec l'élément particulier; si nous avons affaire à un élément différent, nous le gardons et continuons notre parcours dans le reste de la liste, sinon nous l'enlevons (en fait, nous ne le gardons pas) et nous continuons également à chercher (d'autres occurrences) dans le reste de la liste. Naturellement nous arrêtons quand la liste est vide.

Cet algorithme se traduit aisément en LISP :

```
(DEFUN DELETE (ELE LISTE)(COND
    ((NULL LISTE) NIL)
    ((EQUAL ELE (CAR LISTE))(DELETE ELE (CDR LISTE)))
    (T (CONS (CAR LISTE)(DELETE ELE (CDR LISTE)))) ))
```

Si l'élément **ELE** ne fait pas partie de la liste **LISTE**, cette fonction livre une copie de **LISTE**. Notez également l'utilisation de la fonction **EQUAL** comme prédicat.

Voici quelques exemples d'appel :

```
(DELETE 'A '(A A H))              →  (H)
(DELETE '(A) '(A A H))            →  (A A H)
(DELETE '(A) '(A (A) H))          →  (A H)
(DELETE '(DO RE) '(FA SOL (DO RE)))  →  (FA SOL)
```

et voici une trace de l'exécution d'un appel de la fonction **DELETE**. Regardez à nouveau le mélange entre appels récursifs terminaux et appels récursifs normaux :

```
? (DELETE 'A '(A A H A H))
  ---> (DELETE A (A A H A H))
    ---> (DELETE A (A H A H))
      ---> (DELETE A (H A H))
        ---> (DELETE A (A H))
          ---> (DELETE A (H))
            ---> (DELETE A NIL)
              <--- DELETE NIL
          <------ DELETE (H)
    <-------- DELETE (H H)
= (H H)
```

6.5 EXERCICES

1. Si l'on appelle **DELETE** comme suit :

   (DELETE 'A '(A (B A (C A) A) A))

   le résultat sera **((B A (C A) A))**. Changez la fonction de façon qu'elle livre le résultat **((B (C)))**, c'est-à-dire : modifiez la fonction pour qu'elle élimine *toute* occurrence d'un élément donné, indépendamment de la profondeur à laquelle cet élément se situe à l'intérieur de la liste.

2. Ecrivez une fonction qui double chacun des éléments de la liste donnée en argument. Exemples d'appels possibles :

(DOUBLE '(A B C))	→	**(A A B B C C)**
(DOUBLE '(DO (RE MI) FA))	→	**(DO DO (RE MI)(RE MI) FA FA)**
(DOUBLE '(JE BE GAYE))	→	**(JE JE BE BE GAYE GAYE)**

3. Modifiez la fonction **DOUBLE** de l'exercice 3 de manière à doubler tous les atomes d'une liste, indépendamment de la profondeur à laquelle ils se trouvent.

4. Ecrivez une fonction prédicat, à deux arguments, qui ramène *vrai* si le premier argument a une occurrence à l'intérieur de la liste deuxième argument, et qui ramène **NIL** si le premier argument n'a aucune occurrence dans la liste deuxième argument. Ci-dessous quelques exemples d'applications de cette fonction :

(MEMQ 'B '(A B C D))	→	**(B C D)**
(MEMQ 'Z '(A B C D))	→	**()**
(MEMQ 3 '(1 2 3 4))	→	**(3 4)**

5. Modifiez la fonction précédente de manière telle que l'élément cherché puisse se trouver à une profondeur quelconque et puisse être d'un type arbitraire (c'est-à-dire : une liste ou un atome).

6. Ecrivez une fonction qui groupe les éléments de deux listes deux à deux. Voici quelques exemples montrant ce que la fonction doit faire :

```
(GR '(A B C) '(1 2 3))          →   ((A 1)(B 2)(C 3))
(GR '(M N O) '(13 14 15 16))    →   ((M 13)(N 14)(O 15) 16)
(GR '(M N O P) '(13 14 15))     →   ((M 13)(N 14)(O 15) P)
```

7. Que fait la fonction suivante :

```
(DEFUN FOO (X)
    (IF (NULL X) NIL
        (APPEND (FOOBAR X X) (FOO (CDR X)))))
```

avec la fonction **FOOBAR** que voici :

```
(DEFUN FOOBAR (X Y)
   (IF (NULL Y) ()
      (CONS X (FOOBAR X (CDR Y)))))
```

8. Que fait la fonction suivante :

```
(DEFUN F (X Y)
   (IF X (CONS (CAR X)(F Y (CDR X))) Y))
```

9. Et que fait la fonction suivante :

```
(DEFUN BAR (X)
   (IF (NULL (CDR X)) X
      (CONS (CAR (BAR (CDR X)))
         (BAR (CONS (CAR X)
                    (BAR (CDR (BAR (CDR X)))))))))
```

7. L'ARITHMETIQUE

7.1 LES FONCTIONS ARITHMETIQUES DE BASE

Bien que LISP soit principalement un langage de programmation symbolique (non numérique) il donne des possibilités de programmer des problèmes numériques. Dans ce chapitre nous nous limiterons au calcul avec des nombres entiers compris entre -2^{31} et $2^{31} - 1$.[1] Regardons d'abord les fonctions arithmétiques standard :

$$(1+\ n) \rightarrow n+1$$

La fonction **1+** additionne donc la valeur **1** à son argument. C'est une fonction d'*incrémentation*.

$$(1-\ n) \rightarrow n-1$$

La fonction **1–** soustrait la valeur **1** à son argument. C'est une fonction de *décrémentation*.

Attention 1

Remarquons que **1+** et **1–** sont des *noms* de fonction qui s'écrivent comme toutes les fonctions LISP en préfixe, c.à.d.: avant l'argument. **1+** ou **1–** sont donc des noms comme **CAR** ou **CDR**, et le signe '+' ou le signe '-' doit être écrit de manière telle qu'il suive directement le chiffre '1' (sans espace entre '1' et '+' ou '–' !).

Attention 2

Notons également que ni la fonction **1+**, ni la fonction **1–**, ni les autres fonctions arithmétiques de base ne modifient les valeurs de leurs arguments : tout ce qu'elles font est de calculer une valeur !

Voici les fonctions d'addition (+), de soustraction (-), de multiplication (*) et de division (/) :[2]

1. Dans des LISPs classiques, la taille minimum et maximum des nombres dépend de la machine particulière sur laquelle votre LISP tourne (ici nous comptons avec une machine à mots de 32 bits). Néanmoins, CommonLISP change automatiquement la représentation des nombres si le résultat d'un calcul est à l'extérieur de l'intervalle de ces nombres *ordinaires*.

$$
\begin{array}{lcl}
(+\ n1\ n2) & \rightarrow & n1 + n2 \\
(-\ n1\ n2) & \rightarrow & n1 - n2 \\
(*\ n1\ n2) & \rightarrow & n1 * n2 \\
(/\ n1\ n2) & \rightarrow & \lfloor n1 / n2 \rfloor
\end{array}
$$

Toutes ces opérations arithmétiques sont génériques, c.à.d. les arguments peuvent être d'un type numérique quelconque : si tous les arguments sont des nombres entiers, le résultat est également un nombre entier, sinon, le résultat s'adapte au type des arguments. L'exception à cette règle est la fonction de division / : dans quelques implémentations de LISP, cette division est une *division entière*, ce qui veut dire que le résultat d'une division de *n1* par *n2* est le plus grand entier x, tel que

$$(x * n2) \leq n1$$

ce qui se note normalement : $\lfloor n1 / n2 \rfloor$.

En Le_LISP, par contre, le résultat d'une division, par la fonction /, d'un nombre entier n par un entier m produit une interruption si n n'est pas un multiple de m. Pour avoir la division entière telle que nous venons de la décrire, donc avec deux arguments entiers, Le_LISP offre la fonction **QUO**.

En CommonLISP, le résultat d'une division entière est soit un entier, soit une fraction. Ceci préserve la correction du calcul dans des expressions où le résultat d'une division entière est utilisé pour d'autres computations.

Il existe également une fonction pour calculer le *reste de la division entière*, nommé **REM**.

$$(\textbf{REM}\ n1\ n2) \rightarrow n1 - ((\lfloor n1 / n2 \rfloor) * n2)$$

Voici quelques exemples d'utilisation de ces fonctions, ainsi que leurs résultats en Le_LISP et en CommonLISP :

		Le_LISP	CommonLISP
(1+ 1024)	→	**1025**	**1025**
(1- 0)	→	**-1**	**-1**
(1- -345)	→	**-346**	**-346**
(+ 17 9)	→	**26**	**26**
(+ 1 -1)	→	**0**	**0**
(- 10 24)	→	**-14**	**-14**
(* 6 4)	→	**24**	**24**
(/ 1024 2)	→	**512**	**512**
(/ 17 7)	→	**?**	**17/7**[4]
(REM 1024 2)	→	**0**	**0**
(REM 17 7)	→	**3**	**3**

2. En général, les fonctions +, - et * admettent un nombre quelconque d'arguments. Avant de commencer à écrire de grands programmes sur une machine, testez donc d'abord l'arité de ces quelques fonctions arithmétiques.

```
(+ (* 3 3)(* 4 4))          →   25      25
(- 111 (* 4 (/ 111 4)))     →   3       0
(/ (+ 10 90)(- (* 5 6) 10)) →   5       5
```

Munis de ces quelques fonctions standard nous pouvons écrire déjà un grand nombre de fonctions utiles. Voici par exemple une fonction qui calcule le *carré* de son argument :

```
(DEFUN CARRE (N) (* N N))
```

et voici encore quelques fonctions arithmétiques simples et fort utiles : d'abord deux fonctions qui calculent le *carré* du *carré* de son argument :

```
(DEFUN CARRE-DU-CARRE (N) (CARRE (CARRE N)))
```

ou, plus simplement :

```
(DEFUN CARRE-DU-CARRE (N) (* N (* N (* N N))))
```

Voici une fonction qui calcule l'expression (A + B)/(A − B) :

```
(DEFUN SOMME-DIFFER (N M)(/ (+ N M)(- N M)))
```

Que fait donc la fonction suivante ?

```
(DEFUN FOO (L1 L2)
    (+ (* (CAR L1)(CAR L2))(* (CADR L1)(CADR L2))))
```

7.2 LES PREDICATS NUMERIQUES

Afin de pouvoir écrire des fonctions arithmétiques plus intéressantes, nous devons préalablement connaître quelques prédicats numériques. Voici les plus importants :

D'abord un prédicat qui teste si son argument est égal à 0

```
                    →  T si n = 0
(ZEROP n)
                    →  NIL si n ≠ 0
```

Le prédicat = se lit *numériquement égal* et teste l'égalité de deux nombres

4. Pour avoir le résultat entier d'une division entière en CommonLISP, vous devez utiliser la fonction **FLOOR** sur la fraction résultat. Par exemple : **(FLOOR (/ 17 7)) → 2**.

(= *arg1 arg2*) → **T** si *arg1* = *arg2*
→ **NIL** s'ils sont différents

Quelques autres prédicats numériques sont :

(> *n1 n2*) → **T** si *n1* est supérieur à *n2*
→ **NIL** si *n1* est inférieur ou égal à *n2*

(< *n1 n2*) → **T** si *n1* est inférieur à *n2*
→ **NIL** si *n1* est supérieur ou égal à *n2*

(>= *n1 n2*) → **T** si *n1* est supérieur ou égal à *n2*
→ **NIL** si *n1* est inférieur à *n2*

(<= *n1 n2*) → **T** si *n1* est inférieur ou égal à *n2*
→ **NIL** si *n1* est supérieur à *n2*

Dans la foulée, construisons deux autres prédicats, l'un, appelé **EVENP**, qui teste si son argument est un nombre pair, et l'autre, appelé **ODDP**, qui teste si son argument est un nombre impair. Ils seraient donc définis comme suit :

(EVENP *n*) → **T** si *n* est pair
→ **NIL** si *n* est impair

(ODDP *n*) → **T** si *n* est impair
→ **NIL** si *n* est pair

L'écriture de ces deux fonctions ne devrait pas poser de problèmes :

```
(DEFUN EVENP (N) (ZEROP (REM N 2)))

(DEFUN ODDP (N) (= (REM N 2) 1))
```

7.3 LA PROGRAMMATION NUMERIQUE

Pour nous habituer un peu aux algorithmes numériques nous allons, dans le reste de ce chapitre, donner quelques-uns des algorithmes les plus connus. Commençons par l'omniprésente fonction **FACTORIELLE**. Pour ceux qui ne savent pas ce qu'est la factorielle d'un nombre *n*, rappelons que c'est le produit de tous les nombres entiers positifs inférieurs ou égaux à *n*. Ainsi on a les deux définitions suivantes :

factorielle (n) = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1
factorielle (0) = 1

Dans les manuels de mathématiques, vous trouvez très souvent la définition récurrente que voici :

factorielle (0) = 1
factorielle (n) = n * factorielle (n - 1)

Cette définition nous donne - tel quel - l'algorithme récurrent à construire. Traduisons-la en LISP sous forme récursive :

```
(DEFUN FACTORIELLE (N)
    (IF (ZEROP N) 1
        (* N (FACTORIELLE (1- N)))))
```

C'est, à quelques transpositions textuelles près, la même chose que la définition mathématique. Juste pour le plaisir voici quelques appels et leurs résultats :

```
(FACTORIELLE 0)  →  1
(FACTORIELLE 1)  →  1
(FACTORIELLE 2)  →  2
(FACTORIELLE 3)  →  6
(FACTORIELLE 4)  →  24
(FACTORIELLE 5)  →  120
```

Cette fonction croît très vite.[5]

Dans son temps, Euclide avait développé un algorithme pour trouver le *plus grand commun diviseur*, abrégé PGCD.[6] Son algorithme est le suivant :

5. D'ailleurs, saviez vous qu'il n'existe qu'un seul nombre pour lequel l'équation :

   factorielle(n) = factorielle(a) * factorielle(b)

   a une solution non triviale ? C'est

   factorielle(10) = factorielle(7) * factorielle(6)

6. Vous trouverez de nombreux renseignements sur l'algorithme d'Euclide et d'autres algorithmes numériques dans le deuxième volume de l'oeuvre monumentale de D. E. Knuth. Ce livre est une telle source d'informations que sa lecture est devenue quasiment obligatoire. Voici les références : D. E. Knuth, *The Art of Computer Programming*, le premier volume s'intitule *Fundamental Algorithms*, le deuxième *Semi-numerical Algorithms* et le troisième *Searching and Sorting*. Le tout est publié chez Addison-Wesley Publ. Company.

pour calculer le PGCD de *m* et *n* il faut
si *m* est supérieur à *n* calculer le PGCD de *n* et *m*
sinon, si le reste de la division de *n* par *m* est égal à 0
si *m* = 1 alors *m* et *n* sont premiers entre eux
sinon *m* est le PGCD de *m* et *n*
sinon on a la relation :
PGCD (m, n) = PGCD (reste ⌊(n / m)⌋, m)

Encore une fois : les définitions récurrentes sont très courantes en mathématiques. Voici la traduction de l'algorithme d'Euclide en fonction LISP récursive :[7]

```
(DEFUN PGCD (M N)
    (LET ((X (PGCD1 M N)))
        (IF X X (LIST M 'ET N 'SONT 'PREMIERS 'ENTRE 'EUX))))

(DEFUN PGCD1 (M N)(COND
    ((> M N) (PGCD1 N M))
    ((= (REM N M) 0)
        (IF (= M 1) () M))
    (T (PGCD1 (REM N M) M)) ))
```

Evidemment, vous ne connaissez pas la fonction **LET**. **LET** est utilisée ici pour garder quelque part la valeur de l'appel **(PGCD1 M N)**, afin d'y avoir accès dans la suite. Il aurait été possible d'écrire la fonction **PGCD** comme suit :

```
(DEFUN PGCD (M N)
    (IF (PGCD1 M N) (PGCD1 M N)
        (LIST M 'ET N 'SONT 'PREMIERS 'ENTRE 'EUX)))
```

Mais avec cette écriture l'appel de **(PGCD1 M N)** aurait été calculé *deux* fois : une fois pour le prédicat du **IF**, pour savoir s'il existe un PGCD de **M** et **N**, et une fois pour réellement connaître la valeur de l'appel. Ce qui – le moins qu'on puisse dire – ne serait pas très élégant. On avait besoin d'une méthode pour *lier* cette valeur intermédiaire à une variable. C'est justement l'une des utilisations courantes de la fonction **LET**.

Voici comment elle fonctionne : **LET** est une manière de définir une fonction *sans nom*. Voici sa définition syntaxique :

(LET (*élé*$_1$ *élé*$_2$ ... *élé*$_n$) *corps-de-fonction*)

avec chacun des *élé*$_i$ défini comme :

(*variable valeur*)

La sémantique – c.à.d.: le sens – de la fonction **LET** est la suivante :

7. Cette définition en LISP ne correspond pas tout à fait à celle donnée plus haut. Nous l'avons rendue un peu plus parlante : au lieu de ramener le nombre 1 si les deux arguments sont premiers entre eux, nous ramenons une petite phrase disant exactement cela. Bien entendu, si vous voulez vraiment utiliser le PGCD dans un calcul, vous devrez faire les changements adéquats.

A l'entrée du **LET** les différentes variables sont liées aux valeurs données dans les couples *variables-valeurs* de la liste de variables du **LET**. Ensuite est evalué le *corps-de-fonction* du **LET** avec ces liaisons. La valeur d'un **LET** est la valeur de la dernière expression évaluée (comme pendant l'évaluation de fonctions utilisateurs normales). C'est donc à la fois la définition d'une fonction (sans nom toutefois) et son appel !

Revenons alors à la fonction **PGCD** : à l'entrée du **LET** la variable **X** sera liée à la valeur de l'appel de la fonction auxiliaire **PGCD1**. Cette valeur peut être un nombre, si les arguments ont un PGCD, ou **NIL** si les arguments sont premiers entre eux. Selon cette valeur de **X**, on ramènera alors en valeur soit le PGCD soit une liste indiquant que les deux arguments sont premiers entre eux. Voici quelques appels et leurs valeurs :

```
? (PGCD 256 1024)
= 256

? (PGCD 3456 1868)
= 4

? (PGCD 567 445)
= (567 ET 445 SONT PREMIERS ENTRE EUX)

? (PGCD 565 445)
= 5

? (PGCD 729 756)
= 27

? (PGCD 35460 18389)
= 9
```

Ci-dessous est une autre fonction numérique : la fonction **RACINE-ENTIER**. C'est une fonction qui trouve la racine entière d'un nombre. Rappelons que la racine entière x d'un nombre n est définie comme le plus grand entier x tel que

$$(x * x) \leq n$$

(le signe '$\leq$' veut dire *est inférieur ou égal à*). Voici la définition CommonLISP de la fonction **RACINE-ENTIER** :

```
(DEFUN RACINE-ENTIER (N)
    (LABELS ((AUXILIAIRE (P K N)
                (IF (< N 0) K
                    (AUXILIAIRE (+ P 2) (1+ K) (- N (+ 2 P))))))
        (AUXILIAIRE 1 0 N)))
```

La fonction **LABELS** permet de définir des fonctions *locales*. De telles fonctions locales ne sont

connues qu'à l'intérieur de l'appel de la fonction **LABELS** correspondant. Voici la définition syntactique de **LABELS** :

(**LABELS** (*(nom1 liste-de-variables corps-de-fonction)*
(nom2 liste-de-variables corps-de-fonction)
....
(nom-n liste-de-variables corps-de-fonction))
corps-du-LABELS)

Ceci veut dire : le premier argument de la fonction **LABELS** est une liste de définitions de fonctions. Chaque définition de fonction est de la forme :

(*nom liste-de-variables corps-de-fonction*)

où *nom* est le nom de la fonction, *liste-de-variables* est la liste des variables de la fonction et *corps-de-fonction* est son corps, c'est-à-dire : une séquence d'expressions à évaluer.

Le *corps-du-LABELS* est une séquence d'expressions à évaluer. Les diverses fonctions *nom1, nom2, ..., nom-n* sont *locales* au *corps-du-LABELS* et ne peuvent être appelées que de l'intérieur de celui-ci (ou de l'intérieur de la fonction même, dans le cas de définitions récursives). Dans notre exemple, la fonction **AUXILIAIRE** n'est connue qu'à l'intérieur du **LABELS** de la fonction **RACINE-ENTIER**.

En Le_LISP ou d'autres LISPs de cette famille, les fonctions locales sont construites avec une forme spéciale de **LET**. La fonction **LET** est, comme nous l'avons déjà remarqué, à la fois la définition d'une fonction *et* son appel.

Donc, en Le_LISP, la forme spéciale du **LET** est la fonction **LETN**, pour **LET** Nommé, qui prend comme premier argument supplémentaire le nom de la fonction définie par ce **LETN**.

La définition syntaxique de cette fonction est comme suit :

(**LETN** *nom* ($élé_1$ $élé_2$ ... $élé_n$) *corps-de-fonction*)

Cette forme a le même comportement que la fonction **LET** *sauf* que le symbole *nom* est, à l'intérieur de **LETN**, lié à cette expression même. On peut donc récursivement appeler ce **LET** par un appel de la fonction *nom*.

En Le_LISP, vous utilisez donc deux formes de **LET** : la première, **LET**, si vous voulez juste temporairement lier des valeurs *sans* faire des appels récursifs, la deuxième, **LETN**, si vous voulez faire des appels récursifs de la fonction. En Le_LISP, la fonction **RACINE-ENTIER** s'écrit donc de la manière suivante :

```
(DE RACINE-ENTIER (N)
    (LETN AUXILIAIRE ((P 1)(K 0)(N (1- N)))
        (IF (< N 0) K
            (AUXILIAIRE (+ P 2)(+ K 1)(- N (+ 2 P))))))
```

La fonction **AUXILIAIRE** réfère donc à la fonction construite avec la fonction **LETN**. Cette fonction **AUXILIAIRE** n'est connue qu'à l'intérieur du corps du **LET** nommé (donc du **LETN**) et n'est pas connue à l'extérieur.

Quel est l'algorithme sous-jacent dans **RACINE-ENTIER** ? Pourquoi cette fonction livre-t-elle la racine entière ? Pour montrer que ça marche réellement, voici quelques exemples :

```
(RACINE-ENTIER 9)      →   3
(RACINE-ENTIER 10)     →   3
(RACINE-ENTIER 8)      →   2
(RACINE-ENTIER 49)     →   7
(RACINE-ENTIER 50)     →   7
(RACINE-ENTIER 12059)  →  109
```

Voici encore quelques exemples de fonctions numériques. D'abord la fonction Le_LISP **DEC-BIN** qui traduit un nombre décimal en une liste de zéros et de uns (donc en une liste représentant un nombre binaire) :

```
(DE DEC-BIN (N)
   (IF (> N 0)
       (APPEND (DEC-BIN (QUO N 2))(APPEND (REM N 2) NIL))
       NIL))
```

Pour cette fonction, la définition de la fonction **APPEND** (cf. chapitre 6.2) a été légèrement modifiée. Quelle est la nature de cette modification ? Pourquoi est-elle nécessaire ?

Voici quelques exemples d'appels :

```
? (DEC-BIN 5)
= (1 0 1)

? (DEC-BIN 1023)
= (1 1 1 1 1 1 1 1 1 1)

? (DEC-BIN 1024)
= (1 0 0 0 0 0 0 0 0 0 0)

? (DEC-BIN 555)
= (1 0 0 0 1 0 1 0 1 1)
```

Si vous n'avez pas encore trouvé la modification nécessaire de la fonction **APPEND**, pensez au fait qu'elle doit concaténer deux *listes*, et qu'ici le premier argument peut être un atome, comme dans l'appel **(APPEND (REM N 2) NIL)**.

Voici donc la nouvelle fonction **APPEND** :

```
(DEFUN APPEND (ELEMENT LISTE)(COND
    ((NULL ELEMENT) LISTE)
    ((ATOM ELEMENT) (CONS ELEMENT LISTE))
    (T (CONS (CAR ELEMENT) (APPEND (CDR ELEMENT) LISTE)))))
```

Afin de pouvoir écrire cette fonction **BIN-DEC** en CommonLISP, nous devons nous rappeler que la fonction CommonLISP de division '/' ne tronque pas son résultat à la partie entière, mais c'est exactement pourquoi la fonction utilisant Le_LISP's **QUO** donne le bon résultat.

Nous devons ici utiliser la fonction **FLOOR** de CommonLISP. **FLOOR** est une fonction très intéressante : c'est une fonction qui retourne *deux* valeurs. Voici un exemple :

(FLOOR 10 3) → 3 1

La première valeur est le plus grand entier qui est inférieur ou égal au quotient (le même nombre que le résultat de la fonction **QUO** de Le_LISP), la deuxième valeur est la différence entre le premier argument, ici c'est le nombre 10, et le résultat de la multiplication de la première valeur et du deuxième argument, bref : c'est le reste de la division.

Ci-dessous, quelques exemples supplémentaires de l'utilisation de la fonction **FLOOR** :

```
(FLOOR 17 7)          →   2   3
(FLOOR (/ 17 7) 1)    →   2   3/7
(FLOOR (/ 17 7))      →   2   3/7
(FLOOR 1023 17)       →  60   3
(FLOOR 483 23)        →  21   0
```

Notez que le deuxième argument, si omis, équivaut à **1**.

Des fonctions à *valeurs multiples* – des fonctions retournant plus qu'une valeur – sont une chose très spéciale de CommonLISP. Ici nous n'entrerons pas plus dans leurs détails et nous notons juste que, si de telles fonctions sont utilisées comme des arguments à d'autres fonctions qui n'attendent qu'une seule valeur, seule la première valeur sera utilisée. Par exemple :

(1+ (FLOOR 483 23)) → 22

La deuxième valeur est alors perdue.

Après cette longue digression, voici finalement la fonction **DEC-BIN** en CommonLISP :

```
(DEFUN DEC-BIN (N)
   (IF (> N 0)
       (APPEND (DEC-BIN (FLOOR N 2))(APPEND (REM N 2) NIL))
       NIL))
```

Naturellement, si nous avons une fonction de traduction de nombres décimaux en nombres binaires, il nous faut aussi la fonction inverse : la fonction **BIN-DEC** traduit des nombres

binaires en nombres décimaux.

```
(DEFUN BIN-DEC (N)
    (LABELS ((AUXILIAIRE (N RES)
                (IF N (AUXILIAIRE (CDR N)
                            (+ (* RES 2)(CAR N)))
                    RES)))
            (AUXILIAIRE N 0)))
```

quelques exemples d'utilisation :[8]

```
? (BIN-DEC '(1 0 1))
= 5

? (BIN-DEC '(1 0 0 0 0 0 0 0 0 0 0))
= 1024

? (BIN-DEC '(1 0 1 0 1 0 1 0 1 0 1))
= 1365

? (BIN-DEC '(1 1 1))
= 7
```

Etudiez ces fonctions attentivement ! Faites les tourner à la main, faites les tourner sur la machine.

7.4 EXERCICES

1. Ecrivez une fonction, nommée **NTH**, à deux arguments : un nombre *n* et une liste *l*, et qui livre à l'appel le *n*-ième élément de la liste *l*.

   Voici quelques appels possibles de cette fonction :

   ```
   (NTH 2 '(DO RE MI FA))                → RE
   (NTH 3 '(DO RE MI FA))                → MI
   (NTH 1 '((TIENS TIENS) AHA AHA))      → (TIENS TIENS)
   ```

2. Ecrivez une fonction de transformation de nombres décimaux en nombres octaux et hexadécimaux.

---

8. Rappelons que, d'après notre algorithme de traduction, le programme **BIN-DEC** s'écrit en Le_LISP comme suit:

```
(DE BIN-DEC (N)
    (LETN AUXILIAIRE ((N N)(RES 0))
        (IF N (AUXILIAIRE (CDR N)
                    (+ (* RES 2)(CAR N)))
            RES)))))
```

3. Ecrivez une fonction qui traduit des nombres octaux en nombres décimaux, et une autre pour traduire des nombres hexadécimaux en nombres décimaux.

4. En 1220, un grand problème mathématique dans la région de Pise était le suivant : si un couple de lapins a chaque mois deux enfants, un mâle et une femelle, et si après un mois les nouveaux couples peuvent également produire deux petits lapins de sexe opposé, combien de lapins existent alors après n mois ? Monsieur Leonardo de Pise (communément surnommé Fibonacci) trouva la règle récurrente suivante :

$$\begin{aligned} &\text{fibonacci}(0) = 1 \\ &\text{fibonacci}(1) = 1 \\ &\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \end{aligned}$$

Ecrivez une fonction qui calcule les nombres de Fibonacci pour tout argument n positif.

5. Que fait la fonction suivante :

```
(DEFUN QUOI (N)
    (LABELS ((SELF (RES N)
        (IF (> N 0)
            (SELF (+ (* RES 10)(REM N 10)) (FLOOR N 10))
            RES)))
    (SELF 0 N)))
```

6. La fonction **ACKERMANN** est définie comme suit :

$$\begin{aligned} &\text{ack}(0, n) = n + 1 \\ &\text{ack}(m, 0) = \text{ack}(m-1, 1) \\ &\text{ack}(m, n) = \text{ack}(m-1, \text{ack}(m, n-1)) \end{aligned}$$

Ecrivez cette fonction en LISP. Avez-vous une idée de ce qu'elle fait ?

7. Ecrivez la fonction **LENGTH** qui calcule le nombre d'éléments d'une liste. Voici quelques exemples d'appel :

```
(LENGTH '(PEU A PEU))              →  3
(LENGTH '(((ART) ADJ) NOM VERBE))  →  3
(LENGTH '((DO RE MI)))             →  1
(LENGTH ())                        →  0
```

8. Ecrivez une fonction qui calcule le nombre d'atomes d'une liste. Exemples :

```
(NBATOM '(PEU A PEU))              →  3
(NBATOM '(((ART) ADJ) NOM VERBE))  →  4
(NBATOM '((DO RE MI)))             →  3
(NBATOM '())                       →  0
```

8. LES P-LISTES

Jusqu'à maintenant nous avons utilisé les *symboles*, c'est-à-dire les *atomes littéraux*, soit comme des *noms*, soit comme des *paramètres* de fonctions utilisateurs. Dans ce deuxième cas, les symboles peuvent avoir des *valeurs* : les valeurs auxquelles ils sont liés à l'appel des fonctions dont ils constituent des paramètres. On appelle la valeur de liaison la *C-valeur* ou la *cell-value*. Le langage LISP a également prévu des mécanismes de bases de données associatives grâce à la *P-liste* ou *liste de propriétés*. C'est ce que nous étudierons dans ce chapitre.

8.1 LES SYMBOLES

Regardons d'abord la représentation interne des symboles. Un symbole se distingue d'un autre principalement par trois caractéristiques : son *nom* (sa représentation externe), sa *valeur*, (la valeur à laquelle il est lié), et sa *P-liste*. La valeur d'un symbole est donnée par la fonction **SYMBOL-VALUE**. La P-liste d'un symbole est donnée par la fonction **SYMBOL-PLIST**. Ainsi, l'appel :

(SYMBOL-VALUE 'XYZZY)

calcule la C-valeur du symbole **XYZZY**,[1] et l'appel :

(SYMBOL-PLIST 'XYZZY)

ramène la P-liste du même symbole.[2] Notons qu'en Le_LISP, la fonction **PLIST** peut optionnellement avoir un deuxième argument. Dans ce cas, la liste deuxième argument devient la nouvelle P-liste du symbole premier argument.

Si le symbole **XYZZY** est lié à la valeur numérique **100**, les valeurs de l'expression LISP :

(1+ (SYMBOL-VALUE 'XYZZY))

et de l'expression

(1+ XYZZY)

1. **XYZZY** est un mot magique bien connu des joueurs d'*adventure*. Si vous ne le connaissez pas encore, essayez le, admirez l'effet instantané. Essayez aussi le mot magique **PLUGH**.

2. En Le_LISP la fonction CommonLISP **SYMBOL-VALUE** s'appelle **SYMEVAL**, et la fonction CommonLISP **SYMBOL-PLIST** devient **PLIST** en Le_LISP.

sont identiques. Dans les deux cas la valeur sera **101**.

Voici à présent comment un symbole peut être représenté de façon interne :

C-valeur
sa valeur
P-liste
sa liste de propriétés
P-name
le *nom* de l'atome

Initialement, la C-valeur d'un symbole est la valeur **INDEFINI**, indiquant que le symbole n'est pas lié à une valeur et que toute demande d'accès à sa valeur provoquera une *erreur* : **atome non défini**. La P-liste est initialisée à la valeur **NIL**. Naturellement, le *P-name* - ou *print-name* - sera la suite de caractères donnant le nom de l'atome. Voici la représentation interne du symbole **XYZZY** vu ci-dessus.

100
la valeur de l'atome est **100**
NIL
sa P-liste est égale à **NIL**
XYZZY
et son nom imprimable est **XYZZY**

8.2 L'ACCES AUX P-LISTES

Très souvent il arrive qu'un symbole devra avoir plus d'une seule valeur. Par exemple, pour implémenter une base de données de relations familiales, il peut être nécessaire de représenter quelque part le fait que **PIERRE** a un père, nommé **JEAN**, une mère, nommée **JULIE**, et un fils de nom **GERARD**. Cela veut dire que la valeur de la propriété *père* de **PIERRE** est **JEAN**, que la valeur de la propriété *mère* de **PIERRE** est **JULIE** et que la valeur de la propriété *fils* de **PIERRE** est **GERARD**. Evidemment, les valeurs associées à **PIERRE** sont multiples et dépendent de la caractéristique particulière qu'on interroge. Pour implémenter de telles structures, LISP met trois fonctions, agissant sur la P-liste, à votre service : les fonctions **PUTPROP**, **GET** et **REMPROP**.

Regardons d'abord un exemple d'utilisation de ces fonctions, ensuite leurs définitions. Voici à

présent la suite d'instructions LISP pour implémenter les relations familiales de **PIERRE** :

(PUTPROP 'PIERRE 'PERE 'JEAN)

(PUTPROP 'PIERRE 'MERE 'JULIE)

(PUTPROP 'PIERRE 'FILS 'GERARD)

Ces instructions ont comme effet de mettre sur la P-liste du symbole **PIERRE** sous l'indicateur **PERE** la valeur de l'expression **'JEAN** (i.e.: l'atome **JEAN**), sous l'indicateur **MERE** l'atome **JULIE** et sous l'indicateur **FILS** l'atome **GERARD**.

La P-liste d'un symbole est donc une suite d'indicateurs et de valeurs associées à ces indicateurs :

($indicateur_1$ $valeur_1$ $indicateur_2$ $valeur_2$ . . . $indicateur_n$ $valeur_n$)

Ainsi, après les quelques appels de la fonction **PUTPROP** ci-dessus la P-liste du symbole **PIERRE** aura la forme suivante :

(PERE JEAN MERE JULIE FILS GERARD)

Les P-listes peuvent être imaginées comme des tableaux de correspondance qu'on associe aux symboles. Ainsi la P-liste donnée ci-dessus, peut être représentée comme :

PIERRE

indicateur	MERE	FILS	PERE
valeur	JULIE	GERARD	JEAN

Pour connaître la valeur associée à un indicateur on utilise la fonction **GET**. Ainsi pour connaître le père de Pierre, il suffit de demander :

(GET 'PIERRE 'PERE)

ce qui vous ramène en valeur l'atome **JEAN**.

Voici la définition de la fonction **PUTPROP** :

(PUTPROP *symbole indicateur valeur*) → *symbole*

ce qui a comme effet :

> **PUTPROP** met sur la P-liste du symbole *symbole*, sous l'indicateur *indicateur*, la valeur *valeur*. Si l'indicateur *indicateur* existe déjà, la valeur associée préalablement est perdue. *Tous* les arguments sont évalués (c'est pourquoi nous les avons **quotés**). La valeur ramenée par la fonction **PUTPROP** est la valeur de son premier argument, donc de *symbole*.

En **MACLISP**, le dialecte LISP tournant - entre autres - sur la machine Multics, la fonction **PUTPROP** ramène *valeur* en valeur. En Le_LISP les arguments de **PUTPROP** sont dans l'ordre :

(**PUTPROP** *symbole valeur indicateur*).

Notez également que dans quelques versions de LISP, la fonction **PUTPROP** admet un nombre quelconque d'arguments et est définie comme suit :

(**PUTPROP** *symbole* $indicateur_1$ $valeur_1$ . . . $indicateur_n$ $valeur_n$) → *symbole*

Voici alors la définition de la fonction **GET** :

(**GET** *symbole indicateur*) → *valeur*

> **GET** ramène la valeur associée à l'indicateur *indicateur* sur la P-liste du symbole *symbole*. Si l'indicateur ne se trouve pas sur la P-liste du *symbole*, **GET** ramène la valeur **NIL**. *Il n'y a donc pas de possibilité de distinguer la valeur* **NIL** *associée à un indicateur de l'absence de cet indicateur.*

C'est la raison pour laquelle la fonction CommonLISP **GET** admet un troisième argument optionnel. Ce troisième argument indique la valeur que **GET** doit ramener dans le cas où la propriété cherchée n'existe pas. Par exemple, l'appel :

(GET 'PETER 'AGE 'INCONNU)

ramène la valeur associée à la propriété **AGE** sur la P-liste du symbole **PETER** *si* cette propriété existe, sinon, au lieu de ramener la valeur **NIL**, cet appel de **GET** ramène l'atome **INCONNU**. Ainsi, le troisième argument permet de distinguer aisément entre l'absence d'une propriété et la valeur **NIL** associée à cette propriété.

En fait, nous avons un peu triché : la fonction **PUTPROP** existe dans tous les dialectes de LISP *excepté* en CommonLISP. Là, afin de pouvoir utiliser cette fonction nous devons d'abord la définir comme suit :

```
(DEFUN PUTPROP (SYMBOLE INDICATEUR VALEUR)
       (SETF (GET SYMBOLE INDICATEUR) VALEUR)
       SYMBOLE)
```

La fonction **SETF** est une fonction d'affectation très générale que nous examinerons par la suite plus en détail. Pour l'instant, notons qu'en CommonLISP une valeur de propriété est donnée par :

(**SETF** (**GET** *symbole indicateur*) *nouvelle-valeur*)

Donc, sans la fonction **PUTPROP** que nous venons de définir, la construction de la P-liste du symbole **PIERRE** aurait pu être faite par les trois appels ci-dessous :

```
(SETF (GET 'PIERRE 'PERE) 'JEAN)
(SETF (GET 'PIERRE 'MERE) 'JULIE)
(SETF (GET 'PIERRE 'FILS) 'GERARD)
```

La fonction **REMPROP** enlève une propriété de la P-liste. Ainsi, après l'appel :

```
(REMPROP 'PIERRE 'FILS)
```

le couple **FILS - GERARD** sera enlevé de la P-liste de l'atome **PIERRE**, et l'instruction suivante :

```
(GET 'PIERRE 'FILS)
```

ramènera la valeur **NIL**, indiquant l'absence de l'indicateur **FILS**.

(**REMPROP** *symbole indicateur*) → *symbole*

REMPROP enlève de la P-liste du symbole *symbole*, l'indicateur *indicateur*, ainsi que la valeur associée. **REMPROP** ramène le symbole donné en premier argument.

8.3 EXERCICES

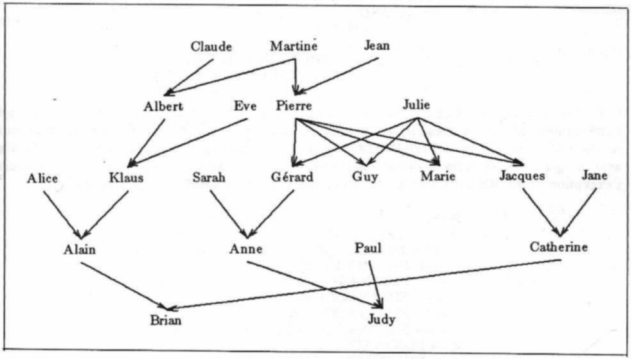
1. Traduisez en LISP, à l'aide des fonctions **PUTPROP**, **GET** et **REMPROP**, les phrases suivantes :

   a. Pierre est le père de Guy.
   b. Pierre a aussi un enfant nommé Marie.
   c. Pierre a également un enfant nommé Jacques.
   d. Pierre est de sexe masculin.
   e. Marie est de sexe féminin.
   f. Guy et Jacques sont de sexe masculin.
   g. Judy est de même sexe que Marie.
   h. Judy a 22 ans.
   i. Anne a 40 ans.
   j. Sarah a le même âge que la somme de l'âge de Judy et Anne.

2. (projet) Supposez que chaque livre d'une bibliothèque soit représenté par un atome et que la bibliothèque entière soit représentée par une liste contenant ces atomes. Supposez également que chaque livre a une propriété nommée *titre*, une nommée *auteur* et une nommée *mots-clefs*. Construisez alors une base de données bibliographiques, avec des fonctions d'interrogation de cette base de données respectives à ces trois propriétés. Naturellement, si pour une requête plusieurs réponses existent, la fonction d'interrogation doit ramener une liste de toutes les réponses possibles. Ainsi, si vous demandez le/les livres de Karl

Kraus, par exemple, et que vous avez dans la bibliothèque les deux titres *Die letzten Tage der Menschheit* et *Von Pest und Presse*, la fonction doit ramener la liste **((DIE LETZTEN TAGE DER MENSCHHEIT)(VON PEST UND PRESSE))**. (Pour avoir une idée de tous les problèmes associés à de telles bases de données, regardez donc un livre sur la *documentation automatique*).

3. Ci-dessous, un petit arbre généalogique, dont les flèches vont des parents vers leurs enfants. Représentez les relations familiales simples (père et mère) et le sexe sur les P-listes des atomes des noms des différentes personnes. Ensuite, écrivez un programme qui y trouve des relations non directement représentées, telles que les relations frères, sœurs, fils, filles, oncles, tantes, cousins et cousines, grands-pères et grands-mères.



Définissez également une fonction **ANCETRES**, qui vous ramène la liste de tous les ancêtres connus d'une personne.

9. LES MEMO-FONCTIONS

Le chapitre 7 s'est terminé par un exercice d'écriture de la fonction **FIBONACCI**. Suivant la définition qui y était donnée, la forme la plus simple de cette fonction est :

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

Cette fonction est un très bel exemple d'une fonction récursive, mais malheureusement, elle est extrêmement inefficace. Examinons la : afin de calculer la valeur de **FIBONACCI** d'un nombre *n*, elle calcule la valeur de **FIBONACCI** de *n- 1*, ensuite de *n- 2* etc, jusqu'à ce qu'elle s'arrête avec *n* égal à 1. Les mêmes valeurs sont ensuite recalculées pour le deuxième appel récursif (à l'exception de **FIBONACCI** de *n- 1*). Voici une trace d'un appel de cette fonction :

```
? (FIBONACCI 4)
  ---> FIBONACCI : (4)
    ---> FIBONACCI : (3)
      ---> FIBONACCI : (2)
        ---> FIBONACCI : (1)
        <--- FIBONACCI = 1
        ---> FIBONACCI : (0)
        <--- FIBONACCI = 1
      <--- FIBONACCI = 2
      ---> FIBONACCI : (1)
      <--- FIBONACCI = 1
    <--- FIBONACCI = 3      ; fin du premier appel récursif ;
    ---> FIBONACCI : (2)
      ---> FIBONACCI : (1)
      <--- FIBONACCI = 1
      ---> FIBONACCI : (0)
      <--- FIBONACCI = 1
    <--- FIBONACCI = 2      ; fin du deuzième appel récursif ;
  <--- FIBONACCI = 5
= 5
```

Intuitivement, il semble ahurissant que la machine recalcule des valeurs qu'elle a déjà préalablement calculées. Les *mémo-fonctions* font une utilisation intensive des P-listes pour y garder les valeurs déjà calculées des appels, donnant ainsi une possibilité de *souvenir* ou *mémorisation* à la fonction. Regardez, voici la définition de la fonction **FIBONACCI** comme mémo-fonction :

```
(DEFUN FIB (N)
   (COND
      ((= N 0) 1)
      ((= N 1) 1)
      ((GET 'FIB N))      ; c'est ICI qu'on économise ! ;
      (T (PUTPROP 'FIB N
            (+ (FIB (1- N))(FIB (- N 2))))
         (GET 'FIB N))))
```

Chaque fois, avant de calculer récursivement la valeur d'un appel, cette fonction regarde d'abord sur sa P-liste s'il n'y aurait pas déjà une valeur pour cet appel, si oui, **FIB** ramène cette valeur. Sinon, **FIB** calcule la valeur, la met sur la P-liste (pour des utilisations ultérieures) et la ramène également. Comparez la trace du calcul de **FIB** de 4 avec la trace obtenue pour la fonction **FIBONACCI** :

```
? (FIB 4)
  ---> FIB : (4)
    ---> FIB : (3)
      ---> FIB : (2)
        ---> FIB : (1)
        <--- FIB = 1
        ---> FIB : (0)
        <--- FIB = 1
      <--- FIB = 2
      ---> FIB : (1)
      <--- FIB = 1
    <--- FIB = 3                ; fin du premier appel récursif ;
    ---> FIB : (2)
    <--- FIB = 2                ; fin du deuzième appel récursif ;
  <--- FIB = 5
= 5
```

Regardez maintenant la trace de l'appel de **FIB** de 5 suivant :

```
? (FIB 5)
  ---> FIB : (5)
    ---> FIB : (4)          ; !! premier appel récursif ;
    <--- FIB = 5
    ---> FIB : (3)          ; !! deuzième appel récursif ;
    <--- FIB = 3
  <--- FIB = 8
= 8
```

Dès que la fonction rencontre l'appel récursif de **(FIB 4)**, elle peut immédiatement ramener la valeur 5, grâce aux souvenirs mémorisés sur la P-liste de **FIB**. Voici sa P-liste après ces appels :

(5 8 4 5 3 3 2 2)

où vous trouvez sous l'indicateur 5 la valeur 8, sous l'indicateur 4 la valeur 5, sous l'indicateur 3 la valeur 3 et sous l'indicateur 2 la valeur 2. Ainsi, pour chaque appel de fonction qui a été déjà calculé une fois, le résultat est immédiatement trouvé en consultant la mémoire de la P-

liste, et aucun calcul n'est effectué plusieurs fois.

Pensez à cette technique, si vous avez une fonction *très* récursive que vous utilisez beaucoup à l'intérieur de votre programme.

Il est encore possible d'améliorer cette fonction **FIB**, en commençant par mettre sur la P-liste de **FIB** les valeurs pour 0 et 1, ce qui permettra d'enlever les deux tests :

```
((= N 0) 1)
((= N 1) 1)
```

et accélérera le calcul.

Ci-dessous la fonction **FIB** ainsi modifiée, d'abord pour Le_LISP :

```
(DE FIB (N)
  (IF (<= N 1) 1     ; pour éviter des problèmes ! ;
      (UNLESS (GET 'FIB 0)
              (PUTPROP 'FIB 0 1)
              (PUTPROP 'FIB 1 1))
      (LETN SELF ((N N)) (COND
            ((GET 'FIB N))
            (T (PUTPROP 'FIB N
                  (+ (SELF (1- N)) (SELF (- N 2))))
               (GET 'FIB N))))))
```

et ensuite pour CommonLISP :

```
(DEFUN FIB (N)
  (LABELS ((SELF (N) (COND
                  ((GET 'FIB N))
                  (T (SETF (GET 'FIB N)
                           (+ (SELF (1- N)) (SELF (- N 2))))))))

    (COND
      ((<= N 1) 1)     ; pour éviter des problèmes ! ;
      (T (UNLESS (GET 'FIB 0)
           (PUTPROP 'FIB 0 1)
           (PUTPROP 'FIB 1 1))
         (SELF N)))))
```

Dans cette version nous avons utilisé une nouvelle fonction : **UNLESS**. De même que la fonction **IF**, c'est une fonction de sélection. Sa définition est :

→ évalue $action_1$ à $action_n$ et ramène $action_n$ en valeur si *test* = NIL

(UNLESS *test* $action_1$ $action_2$ . . . $action_n$**)**

→ NIL si *test* ≠ NIL

L'appel :

(**UNLESS** *test action*$_1$ *action*$_2$ . . . *action*$_n$)

est donc identique à l'appel de la fonction **IF** suivant :

(**IF** *test* () action$_1$ action$_2$ . . . action$_n$)

et n'est qu'une simplification de celui-ci.

Nous avons également une fonction **WHEN** qui est l'inverse de **UNLESS** :

(**WHEN** *test action*$_1$ *action*$_2$ . . . *action*$_n$)
- → évalue *action*$_1$ à *action*$_n$ et ramène *action*$_n$ en valeur si *test* ≠ **NIL**
- → **NIL** si *test* = **NIL**

L'appel :

(**WHEN** *test action*$_1$ *action*$_2$ . . . *action*$_n$)

peut donc être écrit comme :

(**UNLESS** (**NULL** *test*) *action*$_1$ *action*$_2$ . . . *action*$_n$)

ou encore comme :

(**IF** (**NULL** *test*) () action$_1$ action$_2$ . . . action$_n$)

Pour écrire vos sélections, vous avez donc le choix entre les fonctions **IF**, **COND**, **UNLESS** et **WHEN**.

9.1 EXERCICES

1. Ecrivez la fonction **FACTORIELLE** comme mémo-fonction. Comparez les temps d'exécution de **FACTORIELLE** avec utilisation de la P-liste avec ceux de la fonction sans utilisation de la P-liste.

2. Regardez le programme suivant :

```
(DEFUN FOO (X) (COND
    ((< X 0) (- 0 X))
    (T (COND ((ZEROP X) 0)
             (T (* X -1))))))
```

   a. Simplifiez ce programme, en ne prenant en considération que la structure syntaxique de ce programme, de manière à n'utiliser qu'un seul **COND**.

b. Connaissant le sens des différentes fonctions arithmétiques, simplifiez le programme de manière telle qu'il n'y ait plus de **COND** du tout.

3. Ecrivez un petit simplificateur algébrique. Ce simplificateur doit savoir simplifier des expressions arithmétiques LISP (donc des expressions algébriques bien parenthésées) et connaître – au minimum – les simplifications suivantes :

(+ e 0)	→	e
(+ 0 e)	→	e
(* e 1)	→	e
(* 1 e)	→	e
(* e 0)	→	0
(* 0 e)	→	0
(+ *nombre1 nombre2*)	→	*nombre1* + *nombre2*
(* *nombre1 nombre2*)	→	*nombre1* * *nombre2*

Ainsi, le programme devra, par exemple, simplifier l'expression :

(+ (* x 0) (* 10 (+ y 0)))

en :

(* 10 y)

10. LES ENTREES / SORTIES (PREMIERE PARTIE)

Nous n'avons envisagé qu'une seule manière de communiquer avec notre machine : soit en évaluant l'appel d'une fonction, soit en évaluant une variable. Les seules valeurs que nous puissions donner à des fonctions sont celles données à l'appel, et les seules valeurs apparaissant sur l'écran de votre terminal, sont celles résultant d'une évaluation demandée au niveau de l'interaction avec la machine, c'est-à-dire : au *top-level*. En fait, on aimerait également pouvoir imprimer des résultats intermédiaires, ou d'autres informations, voire même écrire des programmes permettant un dialogue avec la machine. Les fonctions permettant d'imprimer ou de lire au milieu de l'évaluation (ou de l'exécution) d'un programme sont appelées : les fonctions d'*entrées/sorties*. C'est de la définition et de l'utilisation de ces fonctions que traite ce chapitre.

10.1 LA FONCTION GENERALE D'IMPRESSION : PRINT

PRINT est la fonction LISP utilisée pour imprimer des valeurs. Cette fonction est la troisième que nous rencontrons, qui est utilisée non seulement pour la valeur qu'elle ramène après évaluation de son argument mais aussi à cause de l'*effet-de-bord* qu'elle provoque.[1] Bien entendu, cet effet de bord est l'impression de la valeur des arguments.

Voici la définition de la fonction **PRINT** :

(**PRINT** *arg*) → *arg*

PLUS : impression du valeur de l'argument arg

PRINT est donc une fonction à un argument. Cet argument sera évalué à l'appel *et* imprimé sur le périphérique standard de sortie (écran de visualisation, papier du télétype, fichier disque ou bande magnétique, suivant le contexte dans lequel vous vous trouvez).[2] La valeur de l'argument sera ensuite ramenée en valeur, de la même manière que toutes les autres fonctions ramènent une valeur utilisable dans la suite du calcul.

Voici deux exemples simples d'utilisation de cette fonction :

1. Les deux autres fonctions à *effets de bord* que nous connaissons sont la fonction **SETF** et la fonction **REMPROP**, qui, d'une part ramènent une valeur, d'autre part provoquent l'effet de bord d'une modification de la P-liste.

2. Nous verrons plus tard comment on peut changer de périphérique standard.

```
? (PRINT 4)
  4                                          ; l'impression ;
= 4                                          ; la valeur ramenée ;
? (PRINT (CONS 'VOILA '(CA MARCHE)))
  (VOILA CA MARCHE)                          ; l'impression ;
= (VOILA CA MARCHE)                          ; la valeur ramenée ;
```

A la fin de l'impression, **PRINT** effectue automatiquement un saut à la ligne suivante.

En Le_LISP et d'autres dialectes de LISP, **PRINT** accepte un nombre arbitraire d'arguments. Chacun d'eux sera évalué et imprimé. La valeur ramenée est la valeur du dernier argument. De plus, l'impression des arguments n'est pas séparée par des espaces en Le_LISP. Ainsi :

```
(PRINT 1 2 3 4)
```

imprime :

```
1234
```

mais ramène toutefois **4** en valeur.

Afin d'obtenir quand même des espaces, il faut les donner comme chaînes de caractères explicites. Par exemple, pour obtenir l'impression séparée des divers arguments, il faut donner l'expression suivante :

```
? (PRINT 1 " " 2 " " 3 " " 4)
  1 2 3 4
= 4
```

Les guillemets entourent des chaînes de caractères. Ici les trois chaînes de caractères sont réduites au caractère espace. Dans la suite de ce chapitre nous verrons les chaînes plus en détail.

Voici à présent une petite fonction qui imprime tous les éléments d'une liste :

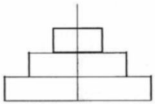
```
(DEFUN IMPRIME-TOUS (L)
   (IF (NULL L) '(VOILA Y EN A PUS)
      (PRINT (CAR L))
      (IMPRIME-TOUS (CDR L))))
```

et voici deux appels de cette fonction :

```
? (IMPRIME-TOUS '(DO RE MI FA SOL))
  DO
  RE
  MI
  FA
  SOL
= (VOILA Y EN A PUS)

? (IMPRIME-TOUS '(UN DEUX TROIS))
  UN
  DEUX
  TROIS
= (VOILA Y EN A PUS)
```

Après chaque impression la machine effectue un *retour-chariot* et un *saut de ligne* (ce qui veut dire en français : elle remet le curseur au début de la ligne suivante).

Pour donner un exemple moins trivial d'utilisation de la fonction **PRINT**, voici une solution au problème des *tours de Hanoi*.[3] Le problème est le suivant : vous avez une aiguille sur laquelle sont empilés des disques de diamètres décroissants, comme ceci :



A proximité vous avez deux autres aiguilles. Il faut alors transporter les disques de la première aiguille vers la deuxième, en se servant éventuellement de la troisième aiguille comme lieu de stockage temporaire et en prenant en compte, toutefois, les restrictions suivantes :

1. on ne peut déplacer qu'un disque à la fois

2. à aucun instant, un disque ne peut se trouver sur un disque de diamètre inférieur.

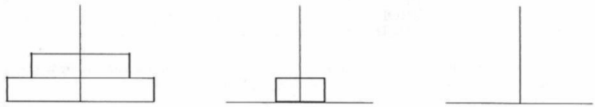
Regardez, voici, graphiquement, la solution pour 3 disques :

3. D'après un mythe ancien, quelques moines dans un temple de l'orient font écouler le temps en transférant 64 disques d'or d'une aiguille vers une autre, en suivant les règles données. L'existence de l'univers tel que nous le connaissons sera achevée, dès l'instant où ils auront fini cette tâche.

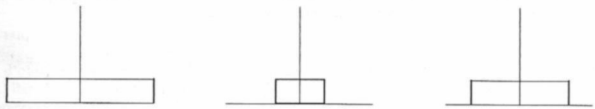
   D'ailleurs, quel sera l'âge total de l'univers si le déplacement d'un disque prend une seconde ?

1. état initial

2. premier déplacement:

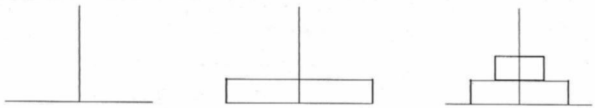


3. deuxième déplacement:

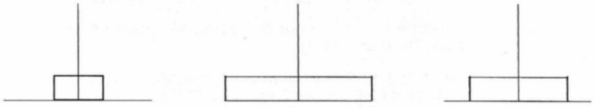


4. troisième déplacement :

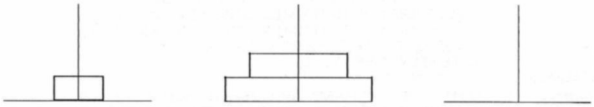
5. quatrième déplacement :



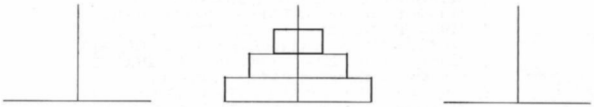
6. cinquième déplacement :



7. sixième déplacement :



8. état final :



Dans la quatrième image on a, sur l'aiguille servant d'intermédiaire, une configuration similaire à celle de l'état initial. La différence entre les deux états est que le disque le plus grand ne fait pas partie de la configuration sur l'aiguille intermédiaire. C'est l'analyse de cette observation qui nous livre la solution au problème. Observez, les trois premiers déplacements n'ont fait rien d'autre que de reconstruire, sur l'aiguille *intermédiaire*, une *tour de Hanoi* comportant un disque de moins que la tour originale. L'aiguille *arrivée* étant libre à cet instant, on peut y placer maintenant le disque de diamètre maximal qui se trouve tout seul sur l'aiguille de *départ*. Il suffit de recommencer les opérations avec, toutefois, une permutation du *rôle* des aiguilles : l'aiguille qui servait au début d'intermédiaire est évidemment devenue l'aiguille de départ (puisque tous les disques à déplacer s'y trouvent). Par contre, l'aiguille originale de départ peut servir maintenant d'aiguille intermédiaire.

La généralisation de cette observation nous livre l'algorithme suivant :

```
(DEFUN HANOI (N DEPART ARRIVEE INTERMEDIAIRE)
   (IF (= N 0) '(ENFIN, C EST FINI)
      (HANOI (1- N) DEPART INTERMEDIAIRE ARRIVEE)
      (PRINT (LIST 'DISQUE N 'DE DEPART 'VERS ARRIVEE))
      (HANOI (1- N) INTERMEDIAIRE ARRIVEE DEPART)))
```

et voici l'exemple de quelques appels :

```
? (HANOI 2 'DEPART 'ARRIVEE 'INTERMEDIAIRE)
  (DISQUE 1 DE DEPART VERS INTERMEDIAIRE)
  (DISQUE 2 DE DEPART VERS ARRIVEE)
  (DISQUE 1 DE INTERMEDIAIRE VERS ARRIVEE)
= (ENFIN, C EST FINI)

? (HANOI 3 'DEPART 'ARRIVEE 'INTERMEDIAIRE)
  (DISQUE 1 DE DEPART VERS ARRIVEE)
  (DISQUE 2 DE DEPART VERS INTERMEDIAIRE)
  (DISQUE 1 DE ARRIVEE VERS INTERMEDIAIRE)
  (DISQUE 3 DE DEPART VERS ARRIVEE)
  (DISQUE 1 DE INTERMEDIAIRE VERS DEPART)
  (DISQUE 2 DE INTERMEDIAIRE VERS ARRIVEE)
  (DISQUE 1 DE DEPART VERS ARRIVEE)
= (ENFIN, C EST FINI)

? (HANOI 4 'DEPART 'ARRIVEE 'INTERMEDIAIRE)
  (DISQUE 1 DE DEPART VERS INTERMEDIAIRE)
  (DISQUE 2 DE DEPART VERS ARRIVEE)
  (DISQUE 1 DE INTERMEDIAIRE VERS ARRIVEE)
  (DISQUE 3 DE DEPART VERS INTERMEDIAIRE)
  (DISQUE 1 DE ARRIVEE VERS DEPART)
  (DISQUE 2 DE ARRIVEE VERS INTERMEDIAIRE)
  (DISQUE 1 DE DEPART VERS INTERMEDIAIRE)
  (DISQUE 4 DE DEPART VERS ARRIVEE)
  (DISQUE 1 DE INTERMEDIAIRE VERS ARRIVEE)
  (DISQUE 2 DE INTERMEDIAIRE VERS DEPART)
  (DISQUE 1 DE ARRIVEE VERS DEPART)
  (DISQUE 3 DE INTERMEDIAIRE VERS ARRIVEE)
  (DISQUE 1 DE DEPART VERS INTERMEDIAIRE)
  (DISQUE 2 DE DEPART VERS ARRIVEE)
  (DISQUE 1 DE INTERMEDIAIRE VERS ARRIVEE)
= (ENFIN, C EST FINI)
```

Analysez bien cette fonction et construisez-en l'arbre d'appel : elle est intéressante, pas seulement à cause de l'utilisation de la fonction **PRINT**.

10.2 LES CHAINES DE CARACTERES

Jusqu'à maintenant nous ne connaissons que trois types d'objets LISP : les *listes*, les *nombres* et les *atomes*. Afin de pouvoir rendre les sorties les plus agréables possible (pour l'oeil), nous avons éventuellement besoin d'imprimer des caractères spéciaux tels que des *espaces*, des *points* ou tout autre caractère jusqu'à maintenant inaccessible à cause de son rôle de *séparateur* qu'il possède normalement en LISP. Les *chaînes-de-caractères* sont des objets LISP permettant l'accès à ces caractères et aucun contrôle n'y est effectué quant au rôle syntaxique que pourraient avoir normalement des caractères pouvant faire partie d'une *chaîne* (abréviation pour *chaîne de caractère*).

Syntaxiquement, toute chaîne de caractères s'écrit entre des guillemets (") :

chaîne de caractère ::= "*suite-de-caractères*"

où une *suite-de-caractères* peut être constituée de n'importe quels caractères.

Voici quelques exemples de chaînes de caractères bizarres mais tout à fait correctes :

```
"aBcDeFgHiJk"
"((((((((((("
".<>.<>."
"! _ %&'()_@ "
```

Les chaînes de caractères sont des objets LISP de type *atome*. Le prédicat **ATOM** appliqué à une chaîne de caractères, ou à une variable ayant comme valeur une chaîne de caractères, ramène donc *vrai*, i.e.: **T**, comme dans l'exemple suivant :

(ATOM "aBc.cBa") → **T**

Afin de néanmoins pouvoir distinguer entre des chaînes de caractères et d'autres atomes, LISP fournit le prédicat **STRINGP** défini comme suit :

(STRINGP *e*)

→ **T** en CommonLISP, la chaîne *e* en Le_LISP si l'argument *e* est une chaîne de caractères

→ **()** dans tous les autres cas

Voici quelques exemples d'applications de ce prédicat : d'abord en CommonLISP :

```
(STRINGP "LISP is beautiful")                  →  T
(STRINGP 'CUBE1)                               →  NIL
(STRINGP (CAR '("aie" est-ce vrai? )))         →  T
(STRINGP '("tiens" "voila" "du boudin"))       →  NIL
```

ensuite en Le_LISP :

```
(STRINGP "LISP is beautiful")                  →  "LISP is beautiful"
(STRINGP 'CUBE1)                               →  ()
(STRINGP (CAR '("aie" est-ce vrai? )))         →  "aie"
(STRINGP '("tiens" "voila" "du boudin"))       →  ()
```

NOTE

Remarquez qu'il ne faut pas *quoter* les chaînes de caractères. Elles sont des *constantes* comme l'atome **T** et l'atome **NIL** et peuvent être considérées comme des atomes qui ont eux-mêmes comme valeur.

Voici quelques exemples d'utilisation de la fonction **PRINT** avec des chaînes de caractères :

```
? (PRINT 3)                    ; normalement ;
  3
= 3

? (PRINT "  ")                 ; avec des espaces ;
  "  "
= "  "

? (PRINT ".....")              ; voyez vous l'utilité ? ;
  "....."
= "....."
```

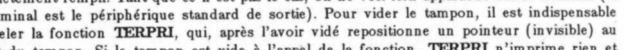
Pour l'instant rappelons-nous juste que chaque fois que nous aurons besoin d'imprimer des caractères spéciaux, nous utiliserons des chaînes de caractères.

10.3 LES FONCTIONS PRIN1, TERPRI ET PRINC

Les impressions ou, plus généralement, les sorties s'effectuent normalement en deux étapes : d'abord on copie ce qu'on veut imprimer dans une zone spéciale de la mémoire de l'ordinateur, zone généralement appelée *buffer* (ou *tampon*), ensuite on envoie cette zone mémoire, ce buffer, vers le périphérique de sortie : on *vide le tampon.*

La fonction **PRINT** combine les deux étapes : elle remplit le tampon de sortie et ensuite elle l'envoie vers le périphérique. Mais il est également possible de séparer les étapes grâce aux fonctions **PRIN1**, en CommonLISP, ou la fonction **PRIN**, en Le_LISP, et **TERPRI**. **PRIN1** est une fonction qui remplit le tampon, on dit qu'elle *édite* le tampon. Cette fonction procède à l'impression effective seulement si, au fur et à mesure du remplissage, le tampon a été complètement rempli. Tant que ce n'est pas le cas, on ne voit rien apparaître sur le terminal (si le terminal est le périphérique standard de sortie). Pour vider le tampon, il est indispensable d'appeler la fonction **TERPRI**, qui, après l'avoir vidé repositionne un pointeur (invisible) au début du tampon. Si le tampon est vide à l'appel de la fonction, **TERPRI** n'imprime rien et passe à la ligne.

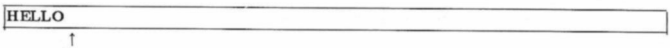
Afin d'être un peu plus explicite, supposons que nous ayons le tampon suivant :



avec un pointeur vers le début du tampon, représenté ici par la petite flèche en dessous du rectangle représentant le tampon. Si l'on évalue l'appel

```
(PRIN1 'HELLO)    ; en CommonLISP
(PRIN 'HELLO)     ; en Le_LISP
```

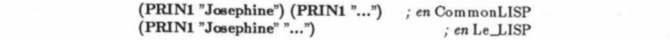
l'effet sera le suivant :



La suite des caractères constituant l'atome **HELLO** a été introduite en tête de tampon et le pointeur a avancé pour pointer maintenant sur la première position vide. L'appel :

(PRIN1 "Josephine") (PRIN1 "...") *; en* CommonLISP
(PRIN1 "Josephine" "...") *; en* Le_LISP

modifiera l'état du tampon comme suit :



Aucune impression effective n'a encore eu lieu. **PRIN1** remplit son tampon de sortie, c'est tout. Pour l'imprimer effectivement, il est nécessaire d'appeler explicitement la fonction **TERPRI** :

(TERPRI)

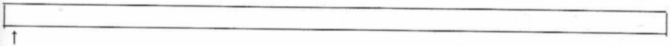
ce qui entraînera, en Le_LISP, l'apparition de la ligne suivante :

HELLO Josephine ...

et en CommonLISP la ligne

HELLO "Josephine" "..."

et le tampon retrouvant son état initial :



c'est-à-dire : le tampon est vidé et son pointeur pointe vers la première position.

Chaque fois que l'on utilise la fonction **PRIN1**, il faut faire suivre tous les appels de cette fonction par au moins un appel de la fonction **TERPRI**.

Formellement, **PRIN1** est défini comme suit :

(PRIN1 *arg*) → *arg*

PLUS : remplissage du tampon de sortie avec la valeur de l'argument.

PRIN1 évalue son argument, l'introduit dans le tampon de sortie, et ramène sa valeur en valeur. **PRIN1** ne procède à l'impression effective que si le tampon est complètement rempli, et qu'aucun autre élément ne peut être introduit.

(TERPRI) → NIL

En Le_LISP, **TERPRI** possède un argument optionnel qui, s'il existe, doit être un nombre. **TERPRI** imprime le tampon de sortie. Si le tampon était vide l'opération est identique à un saut de ligne. Si un argument *n* est présent, la fonction **TERPRI** vide le tampon (c'est-à-dire : imprime le contenu du tampon) et effectue *n- 1 sauts de lignes* supplémentaires. **TERPRI** ramène en valeur la valeur de son argument, s'il est présent, sinon **NIL**.

En CommonLISP, les fonctions **PRINT** et **PRIN1** impriment les objets de manière telle que LISP peut les relire; par exemple, des chaînes de caractères sont imprimées avec les guillemets les entourant.

La fonction Le_LISP **PRINCH**, acronyme pour *PRIN*t *CH*aracter, possède deux arguments : un caractère à imprimer et un nombre indiquant combien de fois on veut imprimer ce caractère. Les *éditions* successives de ce caractère ne sont pas séparées par des espaces, mais 'collées' les unes aux autres. La valeur ramenée par un appel de la fonction **PRINCH** est le caractère donné en premier argument. Voici quelques appels exemples de cette fonction :

```
? (PRINCH "=" 5)
=====
```
; imprime 5 fois le caractère "=" et le ramène en valeur. Ici, la valeur est imprimée sur la même ligne que les 5 impressions du caractère, puisque la fonction **PRINCH** *ne procède pas à l'impression, elle remplit juste le tampon. L'impression est effectuée par le top-level de LISP ;*

```
? (PRINCH "." 10)
.....=.
```
; le dernier point est celui qui est ramené en valeur ;

```
? (PRINCH 1)
1=1
```
; s'il n'a pas de deuxième argument, la valeur 1 est supposée par défaut ;

CommonLISP ne possède pas cette fonction **PRINCH**. Mais, une des caractéristiques les plus puissantes de LISP, dans n'importe quel dialecte, est la possibilité de pouvoir définir ses propres fonctions qui peuvent être utilisées ensuite comme les fonctions standard. Définissons alors la fonction **PRINCH** :

```
(DEFUN PRINCH (STRING N) (COND
      ((ZEROP N) STRING)
      (T (PRINC STRING)(PRINCH STRING (1- N))) ))
```

Cette fonction utilise la fonction standard **PRINC**. **PRINC** est identique à la fonction **PRIN1**, sauf que les chaînes de caractères sont imprimées *sans* les guillemets les entourant. Ci-dessous deux exemples d'utilisation de **PRINC** :

```
? (PRINC ".")
.
=."
? (PRINC "Hello World")
 Hello World
="Hello World"
```

Afin de donner un exemple d'utilisation de ces fonctions, voici maintenant la définition d'une fonction de *formatage* – comme en FORTRAN. Cette fonction a deux arguments, une liste d'éléments à imprimer et une liste de colonnes donnant les tabulations auxquelles les éléments successifs doivent être imprimés.[4]

```
(DEFUN FORMATAGE (ELEMENTS COLONNES)
  (TERPRI)        ; pour une ligne vide ;
  (FORMAT-AUX 0 ELEMENTS COLONNES))

(DEFUN PRINT-FIRST-ELEMENT (N ELE COL) (COND
   ((= N COL) (PRINC ELE) N)
   (T (PRINT-FIRST-ELEMENT (1+ N) ELE COL))))

(DEFUN FORMAT-AUX (N ELEMENTS COLONNES) (COND
      ((NULL ELEMENTS) ())          ; c'est fini ;
      ((NULL COLONNES) ())          ; c'est fini ;
      (T (FORMAT-AUX
            (+ (PRINT-FIRST-ELEMENT
                  N
                  (CAR ELEMENTS)
                  (CAR COLONNES))
               ; pour la place de l'élément ;
               (PLENGTH (CAR ELEMENTS)))
            (CDR ELEMENTS)
            (CDR COLONNES)))))
```

La construction :

```
(COND
   ((NULL ELEMENTS) ())
   ((NULL COLONNES) ())
    . . .
```

précise que le calcul sera terminé dès que la liste **ELEMENTS** ou la liste **COLONNES** est vide. Pour tester si *au moins un* test d'un ensemble de questions est vrai, LISP fournit la fonction **OR** (anglais pour *ou*) qui est définie comme suit :

4. La fonction **PLENGTH** qui calcule le nombre de caractères d'un nom d'atome sera vue au paragraphe suivant.

(OR $test_1$ $test_2$ ... $test_n$)
→ le résultat du premier *test* qui ne s'évalue pas à NIL
→ NIL si tous les $test_i$ s'évalue à NIL

Nous pouvons donc réécrire ce test comme :

```
(COND
   ((OR (NULL ELEMENTS) (NULL COLONNES)) ())
   . . .
```

La fonction **OR** correspond donc tout à fait à un *ou logique*.

Nous avons également la fonction **AND** (anglais pour *et*) qui correspond à un *et logique*, qui teste donc la vérification de *tous* les tests d'un ensemble de tests. Voici sa définition :

(AND $test_1$ $test_2$ . . . $test_n$)
→ $test_n$ si aucun des $test_i$ ne s'évalue à NIL
→ NIL si au moins un des $test_i$ s'évalue à NIL

Mais, revenons à notre programme de formatage.

La fonction auxiliaire **F1** prendra comme arguments une liste contenant des sous-listes, chacune donnant les éléments à imprimer dans le format indiqué par le deuxième argument. La voici :

```
(DEFUN F1 (LISTE-D-ELEMENTS COLONNES) (COND
   ((NULL LISTE-D-ELEMENTS) (TERPRI))
   (T (FORMATAGE (CAR LISTE-D-ELEMENTS) COLONNES)
         (F1 (CDR LISTE-D-ELEMENTS) COLONNES)))
```

et voici, finalement, quelques exemples des impressions que cette fonction produit :

```
(F1 '((ANGLAIS FRANCAIS ALLEMAND)
      (------------------- ------------------- --------------------)
      (COMPUTER ORDINATEUR COMPUTER)
      (STACK PILE KELLER)
      (MEMORY MEMOIRE SPEICHER)
      (DISPLAY ECRAN BILDSCHIRM)
      (KEYBOARD CLAVIER TASTATUR))
  '(5 25 45))

→

ANGLAIS             FRANCAIS            ALLEMAND
------------------- ------------------- --------------------
COMPUTER            ORDINATEUR          COMPUTER
STACK               PILE                KELLER
MEMORY              MEMOIRE             SPEICHER
DISPLAY             ECRAN               BILDSCHIRM
KEYBOARD            CLAVIER             TASTATUR

= NIL
```

Encore un exemple : un tableau de prix mensuels de deux réseaux français de télécommunications (mais avant de vous abonner à l'un d'eux, renseignez-vous sur les prix, ceux donnés ci-dessous risquent d'être périmés) :

```
(F1 '((COUTS TELEPHONE LIAISON CADUCEE TRANSPAC)
    (RACCORDEMENT 4800F 3200F 3200F 4000F)
    (ABONNEMENT 189F 4003F 3130F 1810F)
    (UTILISATION 13536F 0F 4060F 117F))
  '(4 20 30 40 50))

→

COUTS          TELEPHONE  LIAISON  CADUCEE  TRANSPAC
RACCORDEMENT   4800F      3200F    3200F    4000F
ABONNEMENT     189F       4003F    3130F    1810F
UTILISATION    13536F     0F       4060F    117F

= NIL
```

Il n'est actuellement pas nécessaire d'écrire les fonctions de formatage d'une manière aussi compliquée : nous pouvons faire cette chose beaucoup plus simplement si nous utilisons la fonction standard **FORMAT**. Ci-dessous, une nouvelle version de **FORMATAGE** :

```
(DEFUN FORMATAGE (ELEMENTS COLONNES) (COND
      ((NULL ELEMENTS) (TERPRI))  ; c'est fini ;
      (T (FORMAT T "~VT~A" (CAR COLONNES)(CAR ELEMENTS))
         (FORMATAGE (CDR ELEMENTS)(CDR COLONNES)))))
```

La fonction **FORMAT** est conçue spécialement pour aider le programmeur à pouvoir aisément produire des jolies sorties – sans pour autant écrire des programmes compliqués.[5] Actuellement,

chaque appel de **FORMAT** est un petit programme, et le langage de programmation utilisé est composé des signes entre les guillemets. Voici une définition syntactique de la fonction **FORMAT** :

(FORMAT T *une-chaîne-de-contrôle une-séquence-d'arguments***)**

Le premier argument, **T**, indique sur quel périphérique l'impression doit avoir lieu. Nous revenons vers cela plus loin dans ce livre. Le deuxième argument, *une-chaîne-de-contrôle*, indique comment les arguments composant *une-séquence-d'arguments* doivent être imprimés.

La manière la plus simple est de l''appeler avec une chaîne de contrôle ne contenant aucun caractère '~' (tilde). Dans ce cas, il ne peut y avoir des arguments correspondant à *une-séquence-d'arguments*, et la chaîne de contrôle est imprimée telle quelle. Voici l'instantané d'une interaction avec LISP utilisant **FORMAT** de cette manière simple :

```
? (FORMAT T "Hello World")
Hello World
=
? (FORMAT T "aha @ :[}]{")
aha @ :[}]{
=
?
```

Si l'on appelle **FORMAT** avec une chaîne de contrôle ne contenant pas de caractère tilde, cette fonction se comporte presque exactement comme la fonction **PRINC** avec cette même chaîne en argument. L'unique différence est que **PRINC** ramène la chaîne argument en valeur, **FORMAT** par contre ramène la valeur **NIL**.

Pour imprimer les valeurs de variables ou des objets LISP arbitraires, on doit inclure les caractères '~A' dans la chaîne de contrôle et faire suivre la chaîne de contrôle par des arguments correspondant à chaque '~A' de la chaîne de contrôle. Par exemple :

(FORMAT T "~A~A~A" 1 2 3)

imprime

123

et

(FORMAT T "~A ~A ~A" 1 2 3) *; notez l'espacement! ;*

imprime

1 2 3

5. Bien entendu, c'est une référence aux instructions *format* de Fortran. Ces instructions étaient aussi puissantes que celles de LISP. En Fortran, ces instructions étaient les seules instructions *interprétées*.

Voici une définition plus exacte de **FORMAT** : **FORMAT** prend la chaîne de contrôle, remplace toute occurrence de '~A' par la valeur de l'argument correspondant, laisse tous les autres caractères tels quels et, finalement, imprime la chaîne ainsi obtenue. Ci-dessous quelques exemples supplémentaires de l'utilisation de la fonction **FORMAT** :

```
? (FORMAT T "~A est différent de ~A" 1 2)
1 est différent de 2
=
? (FORMAT T "~2A~2A~2A" 1 2 3)
 1 2 3
=
? (FORMAT T "~2A~2A~2A" 1 '(A B C) 3)
 1(A B C) 3        ; notez l'espacement! ;
=
? (FORMAT T "~2,2,2A~2,2,2A~2,2,2A" 1 '(A B C) 3)
  1  (A B C)  3
=
```

L'argument de contrôle '~**A**' est la forme courte de '~*i,j,k,l*A'. Dans le deuxième et troisième exemple ci-dessus nous avons utilisé l'argument *i*, dans le quatrième exemple nous avons donné des valeurs à *i*, *j* et *k*. La signification de '~*i,j,k,l*A' dans la chaîne de contrôle est que l'argument doit être ajusté à droite avec au moins *k* copies du caractère *l* (*l* vaut par défaut le caractère espace), ensuite *j* caractères de padding sont insérés jusqu'à ce que la sortie soit au moins égale à *i* caractères (*i* vaut par défaut 0).

Si tout cela vous semble un peu obscur, ne vous découragez pas : il suffit de jouer un peu avec différentes valeurs pour *i*, *j*, et *k*, pour voir comment cela fonctionne.

Ci-dessous la liste des autres directives possibles dans la chaîne de contrôle :

~D l'argument correspondant doit être un nombre. Ce nombre sera imprimé en tant qu'entier décimal. '~*n*D' utilise une largeur de colonne de *n*, des espaces seront insérés à gauche du nombre si celui-ci nécessite moins que *n* colonnes pour ses chiffres et son signe.

~O c'est comme **~D**, mais le nombre sera imprimé en tant que nombre octal.

~X c'est encore comme **~D**, mais cette fois ce sera un nombre hexadécimal.

~S c'est comme **~A**, mais l'argument sera imprimé comme par **PRIN1** et non pas comme par **PRINC**.

Rappelons que **PRIN1** imprime de manière telle que LISP peut lire ensuite les choses imprimées, c'est-à-dire : les chaînes de caractères seront imprimées entourées par des guillemets.

~% imprime un caractère *retour charriot* (va à *la ligne*); ~*n*% imprime *n* caractères *newline*.

~~ imprime un caractère tilde; ~*n*~ imprime *n* caractères tilde.

~T le **T** représente la *tabulation*. '~*i*,*j*T' insère des espaces jusqu'à la colonne *i*, si cela est possible, sinon cela positionne à la colonne *i* + *kj*, où *k* est le plus petit entier pour lequel cette opération est possible.

Tout cela deviendra très clair dès que vous aurez expérimenté un peu la fonction **FORMAT** et ses diverses formes de directives.

Finalement, à la place d'un argument numérique pour une directive, on peut mettre le caractère **V**, qui prendra alors un argument de *une-séquence-d'arguments* en tant qu'argument pour la directive. C'est cela que nous avons fait dans la dernière version de la fonction **FORMATAGE**.

10.4 L'ACCES AUX NOMS DES ATOMES

En LISP, comme en physique, l'atome n'est indivisible qu'apparemment. Correspondant aux particules élémentaires constituant l'atome physique, les noms des atomes LISP sont constitués de caractères. En Le_LISP, ces caractères sont rendus accessibles par la fonction **EXPLODECH** et la fusion d'un atome à partir de particules élémentaires est effectuée à l'aide de la fonction **IMPLODECH**.

Allons pas à pas : d'abord la fonction **EXPLODECH** qui est définie comme suit :

(**EXPLODECH** *atome*) → une liste de caractères correspondant à la suite de caractères composant la forme externe, donc imprimable, de la valeur de *atome*.

La fonction **EXPLODECH** prend donc un argument, l'évalue, et, si la valeur est un atome, elle livre la liste de caractères constituant cet atome. Voici quelques exemples d'utilisation de cette fonction :

```
? (EXPLODECH 'ATOM)
= (A T O M)

? (EXPLODECH 123)
= (1 2 3)

? (EXPLODECH (CAR '(IL S EN VA)))
= (I L)

? (CDR (EXPLODECH 'UNTRESLONGATOM))))
= (N T R E S L O N G A T O M)

? (LENGTH (EXPLODECH 'CIEL-MON-MARI))
= 13
```

La fonction Le_LISP **EXPLODECH** est plus générale : elle prend en argument une expression LISP quelconque et livre la suite de caractères composant cette expression. Ainsi, si vous appelez :

```
(EXPLODECH '(CAR '(A B)))
```

le résultat est la liste :

```
(|(|  C A R | | |'| |(| A | | B |)| |)|)
```

Les caractères spéciaux sont entourés par deux '|'. En LISP, c'est un moyen de quoter les caractères, pour faire abstraction de leurs rôles particuliers.

Déterminer la longueur de la liste résultant d'un appel de la fonction **EXPLODECH**, revient au même que de calculer le nombre de caractères constituant le *P-name* d'un atome. Cette indication peut être très utile si l'on construit manuellement de belles sorties, comme nous l'avons fait dans notre version de la fonction **FORMAT** afin de calculer la position exacte du pointeur du tampon de sortie. Etant donné que nous avons relativement souvent besoin de la longueur du P-name d'un atome, Le_LISP a prévu une fonction spéciale à cet effet : c'est la fonction **PLENGTH**, qui pourrait être définie comme :

```
(DE PLENGTH (ELE)
    (IF (ATOM ELE)(LENGTH (EXPLODECH ELE)) 0))
```

Voici quelques appels de la fonction **PLENGTH** :

```
(PLENGTH 'HOP)              →  3
(PLENGTH 'MARYLIN)          →  7
(PLENGTH (CAR '(A B C)))    →  1
```

Naturellement, si nous pouvons séparer le nom d'un atome en une suite de caractères, nous aimerions éventuellement également pouvoir construire un atome à partir d'une liste de caractères. Cette opération, l'inverse de la fonction **EXPLODECH**, est effectuée par la fonction **IMPLODECH**.

IMPLODECH prend comme argument une liste de caractères et livre en résultat un atome dont le nom est constitué par ces caractères. Ainsi nous avons la relation suivante :

(IMPLODECH (EXPLODECH *atome*)) → *atome*

Naturellement, de même que la fonction Le_LISP **EXPLODECH**, la fonction inverse **IMPLODECH** de Le_LISP admet comme argument une suite de caractères quelconque correspondant aux caractères constituant une expression LISP, et livre comme résultat l'expression LISP correspondante. Nous avons donc, en Le_LISP la relation plus générale :

(IMPLODECH (EXPLODECH *expression*)) → *expression*

Regardons quelques exemples d'utilisation de cette fonction :

```
? (IMPLODECH '(A T O M))
= ATOM

? (IMPLODECH '(C U R I E U X))
= CURIEUX

? (IMPLODECH (APPEND (EXPLODECH 'BON)(EXPLODECH 'JOUR)))
= BONJOUR

? (IMPLODECH (EXPLODECH 'IDENTITE))
= IDENTITE
```

IMPLODECH est un moyen fort commode pour créer des nouveaux atomes, ce qui peut – surtout dans des applications concernant le langage naturel – être bien utile de temps à autre. Malheureusement, les fonctions **IMPLODECH**, **EXPLODECH** et **PLENGTH** n'existent pas en CommonLISP. Dans d'autres dialectes, ces fonctions existent principalement pour pouvoir faire des programmes qui impriment des objets LISP de manière très jolie. Etant donné que CommonLISP a introduit la fonction **FORMAT** et un très bel ensemble de fonctions traitant les chaînes de caractères (que nous verrons dans la suite), ces fonctions ne sont plus nécessaires. Néanmoins, essayons de les définir en CommonLISP. A travers cet exercice nous apprendrons quelques caractéristiques et fonctions importantes de CommonLISP.

Tout d'abord la fonction **EXPLODECH** :

```
(DEFUN EXPLODECH (SYMBOL)
    (EXPL (STRING SYMBOL) 0 (LENGTH (STRING SYMBOL))))

(DEFUN EXPL (SEQ COUNT N) (COND
    ((= COUNT N) NIL)
    (T (CONS
        (ELT SEQ COUNT)
        (EXPL SEQ (1+ COUNT) N))) ))
```

Examinons cela en détail : CommonLISP distingue entre différents types d'objets : les nombres, les listes, les chaînes de caractères, les symboles (ce sont des atomes non-numériques), les caractères et les tableaux. Pour ces types d'objets existent des fonctions spéciales permettant leur

accès et leur modification. Par exemple, la fonction **LENGTH** calcule la longueur d'une liste et la fonction **REVERSE** construit une copie de son argument, une liste, où l'ordre des éléments est inversé. De temps à autre il peut être intéressant de compter le nombre de caractères d'une chaîne de caractères ou le nombre d'éléments d'un vecteur (un vecteur est un tableau unidimensionnel). Tous ces comptages de caractères ou d'éléments définissent la *longueur* des objets correspondants. C'est la raison pour laquelle CommonLISP introduit un type de données supplémentaire : la *séquence*. Une séquence est une sorte de type abstrait désignant l'ensemble des listes, vecteurs et chaînes de caractères.[6] Autrement dit, les listes, les vecteurs et les chaînes de caractères sont considérés comme des séquences. Ceci de manière telle que, si nous disons d'une fonction qu'elle s'applique à des séquences, nous voulons dire qu'elle s'applique à la fois aux listes, aux vecteurs et aux chaînes de caractères. **REVERSE**, **DELETE** et **LENGTH** sont des fonctions qui s'appliquent aux séquences.

Dans la définition de la fonction **EXPLODECH** ci-dessus, nous avons utilisé la fonction **LENGTH** sur l'argument **SYMBOL** (qui est un atome) après l'avoir convertie vers une chaîne de caractères avec la fonction **STRING**.

Voici une définition de la fonction **STRING** :

(STRING *objet*) → une chaîne dérivée de *objet*. Si *objet* est une chaîne, c'est cette chaîne qui sera retournée; si c'est un symbole, le P-name de ce symbole sera retourné; si *objet* est un caractère, la chaîne de caractères se composant de ce caractère sera retournée. **STRING** n'admet des arguments que de ces trois types.

Et voici trois exemples d'appels de cette fonction :

```
(STRING 'AHA)          →  "AHA"
(STRING "Et alors?")   →  "Et alors?"
(STRING '#\c)          →  "c"
```

Dans le dernier exemple ci-dessus, l'argument est un objet de type caractère, le caractère *c* : c'est de cette manière qu'on doit écrire des objets du type caractère.

Juste pour vérifier que ça marche, examinons quelques fonctions traitant des séquences :

```
(REVERSE '(A (B C) D E))   →  (E D (B C) A)
(DELETE 'A '(A B A C A))   →  (B C)
(DELETE '#\A "ABACA")      →  "BC"
(REVERSE "aBcDe")          →  "eDcBa"
(LENGTH "ABCDE")           →  5
(LENGTH '(A (B C) D E))    →  4
```

Il existe encore une autre fonction manipulant des séquences : c'est la fonction **CONCATENATE**. Cette fonction concatène deux listes, deux vecteurs ou deux chaînes de caractères. Voici

6. D'ailleurs, une chaîne de caractères est rien d'autre qu'un vecteur de caractères.

sa définition :

(CONCATENATE *type-de-résultat séquence-1 séquence-2*)

→ une nouvelle séquence du type *type-de-résultat* contenant tous les éléments de *séquence-1* suivis des éléments de *séquence-2*.

Voici deux exemples de l'utilisation de **CONCATENATE** :

```
(CONCATENATE 'LIST '(A B) '(C D))       →  (A B C D)
(CONCATENATE 'LIST '(A B) '((C D) E))   →  (A B (C D) E)
```

Tant que cette fonction travaille sur des listes, elle fonctionne comme la fonction **APPEND** que nous avons vue au chapitre 6. Examinons quelques exemples supplémentaires :

```
(CONCATENATE 'STRING "VOUS " "ICI?")          →  "VOUS ICI?"
(CONCATENATE 'LIST '(A B) "ABCD")             →  (A B #\A #\B #\C #\D)
(CONCATENATE 'STRING "AIR" (STRING 'PLANE))   →  "AIRPLANE"
```

Afin de comprendre notre définition de **EXPLODECH**, nous devons connaître encore une fonction : la fonction **ELT**. **ELT** est, comme **REVERSE**, **CONCATENATE**, **DELETE** et **LENGTH**, une fonction manipulant des séquences. Elle est définie comme suit :

(ELT *séquence index*) → l'*index*ième élément de *séquence*.
Si *séquence* est une chaîne de caractères, **ELT** ramène un caractère, si c'est une liste, elle ramène l'élément correspondant de la liste. Notez que l'index du premier élément d'une séquence est 0 (zéro).

Voici quelques exemples de **ELT** :

```
(ELT '(A B C D) 3)                  →  D
(ELT "oh, that is all?" 9)          →  #\i
(ELT '(1 2 3 4) (ELT '(4 3 2 1) 3))  →  2
```

Après tout cela, retournez donc vers notre définition CommonLISP de la fonction **EXPLODECH**; essayez de comprendre l'algorithme sous-jacent (ce ne devrait pas être trop difficile!).

En fait, il y a encore une fonction de manipulation de séquences que nous devrions connaître : la fonction **REDUCE**. **REDUCE** est une fonction à deux arguments : le premier argument est lui-même une fonction, et le second est une séquence. L'effet d'une évaluation d'un appel de **REDUCE** est d'appliquer la fonction premier argument aux deux premiers éléments de la séquence donnée en deuxième argument, ensuite, le résultat obtenu est donné à nouveau à la fonction donnée (en tant que premier argument) et l'élément suivant constitue alors le deuxième argument. Ceci continue jusqu'à ce que tous les éléments de la séquence aient été donnés comme argument à la fonction.

Ceci a l'air très difficile, mais ne l'est pas ! Regardez l'exemple suivant :

```
(REDUCE #'+ '(1 2 3 4 5))      →  15
```

ceci calcule, d'après notre explication, exactement la même chose que :

```
(+ (+ (+ (+ 1 2) 3) 4) 5)
```

Evidemment, c'est une fonction extrêmement utile pour appliquer récursivement une fonction de deux arguments à une séquence d'un nombre arbitraire d'arguments. Par exemple, la **FACTORIELLE** de 6 pourrait être calculée comme ceci :

```
(REDUCE #'* '(1 2 3 4 5 6))
```

et notre fonction **APPEND** du chapitre 6 pourrait être généralisée pour un nombre arbitraire de listes. Tout ce qui est nécessaire, c'est de collecter toutes les listes à *appender* dans une grande liste, et ensuite de *réduire* cette liste avec la fonction **APPEND** que nous avons déjà construite. Ainsi

```
(REDUCE 'APPEND '((A B) (C D) (E F) (G H) (I J) (K L))
```

donne comme résultat la concatenation de toutes les sous-listes du deuxième argument, c.à.d.: la liste :

```
(A B C D E F G H I J K L)
```

Enfin ! Munis de ces fonctions nous pouvons définir les fonctions **IMPLODECH** et **PLENGTH**. Elles sont ici :

```
(DEFUN IMPLODECH (LIST)
  (IF (< (LENGTH LIST) 2) (STRING (CAR LISTE))
      (REDUCE #'STRING-APPEND LISTE)))
```

avec **STRING-APPEND** définie comme suit :

```
(DEFUN STRING-APPEND (X Y)
  (CONCATENATE 'STRING (STRING X) (STRING Y)))
```

et, finalement, la fonction **PLENGTH** :

```
(DEFUN PLENGTH (X) (LENGTH (STRING X)))
```

La suite de ce paragraphe sera un programme pour conjuguer des verbes français réguliers des premiers groupes, c'est-à-dire : des verbes réguliers se terminant en 'er', 'ir' et 're'.

; la fonction **CREE-MOT** *crée un nouveau mot à partir d'une racine et d'une terminaison ;*

```
(DEFUN CREE-MOT (RACINE TERMINAISON)
  (IF (NULL TERMINAISON) RACINE
    (IMPLODECH (APPEND (EXPLODECH RACINE)
                       (EXPLODECH TERMINAISON)))))
```

; la fonction **RACINE** *trouve la racine d'un verbe, simplement en y enlevant les deux derniers caractères ;*

```
(DEFUN RACINE (VERBE)
  (IMPLODECH (REVERSE (CDR (CDR (REVERSE (EXPLODECH VERBE)))))))
```

; la fonction **TYPE** *ramène l'atome constitué des deux derniers caractères de l'atome donné en argument ;*

```
(DEFUN TYPE (VERBE)
  (LET ((AUX (REVERSE (EXPLODECH VERBE))))
    (IMPLODECH (CONS (CADR AUX)
                     (CONS (CAR AUX) ())))))
```

; **CONJUGUE** *est la fonction que l'utilisateur va appeler. Elle prépare les arguments pour la fonction auxiliaire* **CONJ1** *;*

```
(DEFUN CONJUGUE (VERBE)
  (TERPRI)                                  ; pour faire plus joli ;
  (CONJ1 (RACINE VERBE)                       ; trouve la racine ;
         (TYPE VERBE)                           ; et son type ;
         '(JE TU IL ELLE NOUS VOUS ILS ELLES)))
```

; **CONJ1** *fait tout le travail : d'abord elle prépare dans* **TERMINAISONS** *la bonne liste de terminaisons, ensuite, dans la fonction* **DO-IT** *intérieur, elle fait imprimer la conjugaison pour chacun des pronoms ;*

```
(DEFUN CONJ1 (RACINE TYP PRONOMS)
  (LABELS ((DO-IT (PRONOMS RACINE TERMINAISONS)(COND
        ((NULL PRONOMS) "voila, c'est tout")
        (T (PRIN1 (CAR PRONOMS))
           (PRIN1 (CREE-MOT RACINE (CAR TERMINAISONS)))
           (DO-IT (CDR PRONOMS) RACINE (CDR TERMINAISONS))))))
    (DO-IT PRONOMS RACINE (COND
        ((EQ TYP 'ER) '(E ES E E ONS EZ ENT ENT))
        ((EQ TYP 'RE) '(S S T T ONS EZ ENT ENT))
        (T '(IS IS IT IT ISSONS ISSEZ ISSENT ISSENT))))))
```

Etudiez bien ce programme : c'est le premier programme, dans cette introduction à LISP, qui se constitue d'un ensemble de plusieurs fonctions. Afin de comprendre son fonctionnement, donnez-le à la machine et faites le tourner. Ensuite, une fois que ça marche, commencez à le modifier, en y ajoutant, par exemple, les verbes en 'oir'.

Mais d'abord regardons quelques appels, afin de nous assurer qu'il tourne vraiment et qu'il fait bien ce qu'on voulait.

```
? (CONJUGUE 'PARLER)    ; d'abord un verbe en 'er' ;

  JE PARLE
  TU PARLES
  IL PARLE
  ELLE PARLE
  NOUS PARLONS
  VOUS PARLEZ
  ILS PARLENT
  ELLES PARLENT
= voila, c'est tout

  ? (CONJUGUE 'FINIR)    ; ensuite un verbe en 'ir' ;

    JE FINIS
    TU FINIS
    IL FINIT
    ELLE FINIT
    NOUS FINISSONS
    VOUS FINISSEZ
    ILS FINISSENT
    ELLES FINISSENT
  = voila, c'est tout

? (CONJUGUE 'ROMPRE)    ; et finalement un verbe en 're' ;

  JE ROMPS
  TU ROMPS
  IL ROMPT
  ELLE ROMPT
  NOUS ROMPONS
  VOUS ROMPEZ
  ILS ROMPENT
  ELLES ROMPENT
= voila, c'est tout
```

Naturellement, ce programme est *très* limité : des verbes réguliers donnent des résultats corrects. Pour tous les autres, n'importe quoi peut arriver, en fonction de leurs terminaisons. Dans les exercices suivants je vous propose quelques améliorations, ou extensions, à apporter à ce programme.

10.5 EXERCICES

1. Rajoutez à la fonction **CONJUGUE** un argument supplémentaire, indiquant le temps dans lequel vous voulez avoir la conjugaison du verbe donné en premier argument. Ainsi, par exemple, l'appel :

```
(CONJUGUE 'FINIR 'SUBJONCTIF-PRESENT)
```

devra vous imprimer :

QUE JE FINISSE
QUE TU FINISSES
QU'IL FINISSE
QU'ELLE FINISSE
QUE NOUS FINISSIONS
QUE VOUS FINISSIEZ
QU'ILS FINISSENT
QU'ELLES FINISSENT

Modifiez le programme de manière à savoir conjuguer dans les temps suivants :

a. présent (ex : je parle)
b. imparfait (ex : je parlais)
c. futur (ex : je parlerai)
d. passé simple (ex : je parlai)
e. conditionnel (ex : je parlerais)
f. subjonctif présent (ex : que je parle)
g. imparfait du subjonctif (ex : que je parlasse)
h. passé-composé (ex : j'ai parlé)
i. plus-que-parfait (ex : j'avais parlé)
j. passé-antérieur (ex : j'eus parlé)
k. conditionnel-passé 1 (ex : j'aurais parlé)
l. conditionnel-passé 2 (ex : j'eusse parlé)

Pour cela, l'appel simple de la fonction **PRINT** à l'intérieur de la fonction **CONJ1** ne suffit plus. Introduisez donc une fonction **VERBPRINT** qui remplacera l'appel de **PRINT** mentionné, et qui se charge, dans le cas du subjonctif, d'effectuer l'impression des 'que' supplémentaires.

Bien évidemment, pour les temps composés, il faut que le programme connaisse les diverses conjugaisons du verbe 'avoir'. Introduisez une autre fonction supplémentaire, **PR-COMPOSE**, responsable de l'impression des temps composés.

2. Ecrivez un programme qui prend en argument une liste représentant une phrase et qui livre cette phrase transformée au pluriel. Exemple :

(PL '(JE FINIS AVEC CE CHAPITRE)) → (NOUS FINISSONS AVEC CE CHAPITRE)

ou

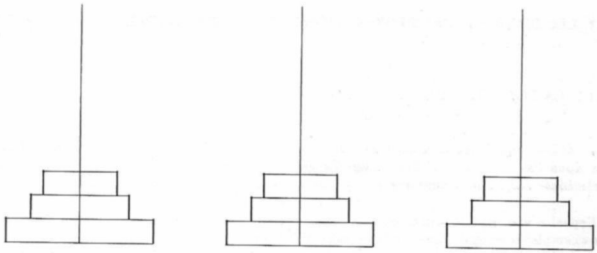
(PL '(IL ME PARLE)) → (ILS ME PARLENT)

ou encore

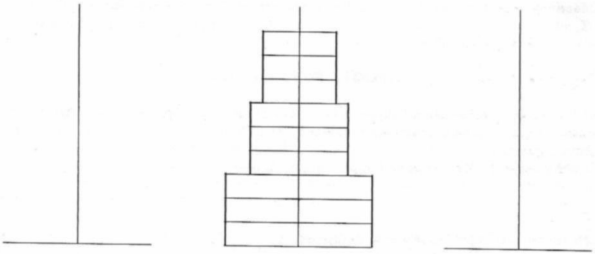
(PL '(TU NE FOULERAS PLUS CETTE TERRE))
→ (VOUS NE FOULEREZ PLUS CETTE TERRE)

3. (pour les algorithmiciens) Ecrivez un programme résolvant le problème de la tour de Hanoi, mais où sont empilés des disques sur chacune des aiguilles à l'état initial. Naturellement, aucune des règles de déplacement n'est modifiée excepté, qu'un disque peut être déposé sur un autre de la même taille. A la fin, l'aiguille d'arrivée doit comporter dans un ordre de grandeur décroissant, tous les disques. Voici graphiquement l'état initial et l'état final :

Etat initial :



Etat final :



11. QUELQUES REMARQUES SUR LES MECANISMES DE LIAISON

11.1 LES DIVERS TYPES DE FONCTIONS UTILISATEURS EN Le_LISP : DE et DF

11.1.1 LA FONCTION DE

Pour définir une fonction utilisateur nous utilisons en Le_LISP la fonction standard **DE**. Telle que nous l'avons vue, **DE** définit une fonction d'un certain *nom* et d'un certain nombre de *paramètres* : autant de paramètres qu'il y a de variables dans la liste de variables.

A l'appel d'une telle fonction, chaque argument est évalué et lié à la variable correspondante. Par exemple, si nous définissons la fonction **FOO** comme :

(DE FOO (X Y Z) (CADR (PRINT (LIST X Y Z))))

nous décrivons une fonction de nom **FOO**, à trois arguments, qui s'appellent respectivement **X**, **Y** et **Z**, et qui imprime les valeurs des arguments. C'est-à-dire, si nous appelons cette fonction avec :

(FOO (CAR '(A B C)) 1 2)

d'abord, la variable **X** est liée à l'atome **A** (la valeur du premier argument), la variable **Y** est liée au nombre **1** (la valeur du deuxième argument), et la troisième variable est liée à la valeur du troisième argument, le nombre **2**. Ensuite, le corps de la fonction est évalué avec les variables liées à ces valeurs. L'effet est donc l'impression de la liste :

(A 1 2)

et le résultat de cet appel est la valeur numérique **1**.

La fonction **DE** permet donc de définir des fonctions utilisateurs à nombre fixe d'arguments, et dont chacun des arguments est évalué à l'appel. Bien souvent, il peut nous arriver de vouloir définir une fonction avec un nombre arbitraire d'arguments, comme la fonction Le_LISP standard **PRINT** qui permet n arguments (avec un $n \in [1,\infty[$). Supposons alors que la fonction standard **+** n'admet que deux arguments et que nous désirons disposer d'une fonction **PPLUS** qui additionne un nombre quelconque d'entiers. Une première solution à ce problème serait de définir la fonction ci-dessous :

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

Mais cette fonction demande une liste d'arguments à l'appel. Ainsi il faut l'appeler par :

```
(PPLUS '(1 2 3 4 5 6))
```

ou :

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

En réalité, il serait préférable d'écrire les arguments l'un après l'autre, comme ci-dessous :

```
(PLUS 1 2 3 4 5 6)
```

ou :

```
(PLUS -1 2 -3 4 -5 6 -7 8)
```

En Le_LISP, il suffit d'écrire :

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

la fonction **PPLUS** étant la même que celle définie ci-dessus.

Regardez bien la différence de cette définition de fonction avec toutes les autres que nous connaissons : au lieu d'une *liste de variables* nous avons juste un *atome variable.*

Si vous définissez une fonction dont le nom-de-fonction est suivi d'un atome, cet atome est, à l'appel de la fonction, lié à *la liste de toutes les valeurs des différents arguments.*

Dans la fonction **PLUS** ci-dessus, la seule et unique variable **L** est, lors de l'appel de :

```
(PLUS 1 2 3 4 5 6)
```

liée à la liste des valeurs des différents arguments, donc à la liste :

```
(1 2 3 4 5 6)
```

Cette liste est ensuite envoyée à la fonction **PPLUS** qui, quant à elle, attend une liste des nombres pour en calculer leur somme.

Prenons un autre exemple : vous avez sûrement constaté qu'il est bien difficile de construire une liste nouvelle à partir d'une suite d'éléments. Chaque fois que nous étions dans une telle situation, nous avons dû écrire quelque chose comme :

(**CONS** $élément_1$ (**CONS** $élément_2$ . . . (**CONS** $élément_n$ ()) . . .))

Par exemple, afin de construire à partir des atomes **A**, **B** et **C** la liste (**A B C**), nous sommes amenés à écrire :

(CONS 'A (CONS 'B (CONS 'C NIL)))

Connaissant maintenant la possibilité de définir des fonctions à un nombre quelconque d'arguments, nous pouvons très simplement écrire une fonction **LIST** qui construit une liste constituée des valeurs de ses différents arguments. La voici :

(DE LIST L L)

C'est tellement simple que cela paraît incroyable ! Regardons ce qui se passe dans l'appel :

(LIST 'A 'B 'C)

1. D'abord la variable **L** est liée à la liste des valeurs des arguments. Les arguments sont **'A**, **'B** et **'C** respectivement. Les valeurs de ces arguments sont donc respectivement **A**, **B** et **C**, et la variable **L** est donc liée à la liste **(A B C)**.

2. Le corps de cette fonction est réduit à l'expression **L**, donc à une évaluation de la variable **L**. La valeur de cette évaluation est la valeur de l'appel de la fonction **LIST**. Bien entendu, la valeur de l'évaluation d'une variable est la valeur à laquelle elle est liée. Cet appel ramène donc en valeur la liste **(A B C)**, la liste que nous voulions construire !

Vous voyez l'utilité de cette chose ?

Notez quand même que nous avons un problème avec cette forme de définition de fonction : nous ne savons pas comment faire des appels récursifs. Afin de voir pourquoi, reprenons la petite fonction **PLUS** ci-dessus et essayons de ne pas utiliser la fonction auxiliaire **PPLUS** :

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

Cette écriture, bien qu'ayant l'air toute naturelle, est évidemment fausse !

Si vous ne savez pas pourquoi, ne continuez pas tout de suite à lire, essayez plutôt de trouver l'erreur vous-même !

Regardez, à l'appel initial :

(PLUS 1 2 3 4)

la variable **L** est liée à la *liste* **(1 2 3 4)**, par le simple mécanisme mis en marche par cette sorte de définition de fonction. Etant donné que cette liste n'est pas vide, LISP procède donc à l'évaluation de la ligne :

(+ (CAR L) *(PLUS (CDR L)))*

qui devrait donc calculer la somme de **1** et du résultat de l'appel récursif de **PLUS** avec l'argument **(2 3 4)**, le **CDR** de la valeur de la variable **L**. MAIS !!! Quelle horreur ! Ce n'est

plus une suite de nombres qui est donnée à la fonction **PLUS** mais une liste. Nous savons que **L** est liée à la liste des valeurs des arguments : **L** recevra donc une liste contenant un seul et unique élément qui est la valeur de l'argument **(2 3 4)**, ce qui n'est sûrement pas ce qu'on voulait faire (Quelle est la valeur de **(2 3 4)** ? Sûrement pas la suite des nombres 2, 3 et 4 !).

C'est cette difficulté qui nous a incité, dans l'écriture de la fonction **PLUS**, à déléguer la boucle (la répétition) vers une fonction auxiliaire, en l'occurrence la fonction **PPLUS**.

Rappelez-vous bien de ce problème d'impossibilité de faire des appels récursifs simples chaque fois que vous construisez une fonction à nombre d'arguments quelconque. Bien entendu, dans la suite de ce livre vous trouverez une solution au problème des appels récursifs de ce type de fonction.

En Le_LISP les fonctions utilisateurs définies avec la fonction **DE** s'appellent des *EXPRs*, et, dans des anciens dialectes de LISP, tels que MACLISP par exemple, les fonctions à nombre d'arguments quelconque s'appellent des *LEXPRs* ou des *NEXPRs*.

11.1.2 LA FONCTION DF

De même qu'il est intéressant de disposer parfois d'une fonction à nombre quelconque d'arguments, où chacun des arguments est évalué, on a de temps à autre besoin de fonctions, à nombre quelconque d'arguments, *sans* évaluation des arguments. De telles fonctions peuvent être réalisées grâce à la fonction de définition de fonctions utilisateurs **DF**, qui est définie comme suit :

(DF *nom-de-fonction liste-de-variables corps-de-la-fonction***)**

Syntaxiquement, **DF** est identique à **DE**. La liaison se fait *exactement* de la même manière que pour **DE**, sauf que les divers arguments ne sont pas évalués.

Voici l'exemple le plus simple :

```
(DF QLIST L L)
```

Cette fonction ressemble fortement à la fonction **LIST** ci-dessus, puisque, comme dans **LIST**, le corps de la fonction se réduit à l'évaluation de l'unique variable paramètre. Etant donné que les arguments ne seront *pas* évalués, l'appel :

```
(QLIST A B C D)
```

ramène donc la liste des arguments tels quels, c'est-à-dire :

```
(A B C D)
```

Notez que les divers arguments *n'ont pas* été quotés. Si on l'appelle comme :

```
(QLIST 'A 'B 'C)
```

le résultat est la liste :

```
('A 'B 'C)
```

Nous verrons dans la suite l'intérêt de ce type de fonctions et comment on peut sélectivement évaluer l'un ou l'autre des arguments.

Les fonctions utilisateurs définies avec la fonction **DF** s'appellent des *FEXPRs*.

11.1.3 EXERCICES

1. Définissez une fonction **TIMES** à nombre d'arguments quelconque qui calcule le produit de tous ses arguments. Exemples :

```
(TIMES 1 2 3 4)                                          →  24
(TIMES (CAR '(1 2 3)) (CADR '(1 2 3)) (CADDR '(1 2 3)))  →   6
```

2. Définissez une fonction **SLIST**, à nombre d'arguments quelconque, qui construit une liste contenant *les valeurs* des arguments de rang pair. Exemples :

```
(SLIST 1 2 3 4 5 6)     →  (2 4 6)
(SLIST 1 2 3 4 5 6 7)   →  (2 4 6)
(SLIST 1 'A 2 'B 3 'C)  →  (A B C)
```

3. Définissez également une fonction **QSLIST**, à nombre d'arguments quelconque, qui construit une liste contenant les arguments de rang pair (tels quels). Exemples :

```
(QSLIST A B C D E F)     →  (B D F)
(QSLIST A B C D E F G)   →  (B D F)
```

4. Finalement, définissez une fonction **MAX** qui trouve le maximum d'une suite de nombres et une fonction **MIN** qui trouve le minimum d'une suite de nombres. Voici quelques exemples d'appels :

```
(MAX 1 2 10 -56 20 6)                          →   20
(MIN 1 2 10 -56 20 6)                          →  -56
(MAX (MIN 0 1 -34 23 6) (MIN 36 37 -38 476 63))  →  -34
```

11.2 QUELQUES DETAILS SUR LES DEFINITIONS DE FONCTIONS EN CommonLISP

En Le_LISP c'est la structure de la liste de variables qui détermine les liaisons des variables.

Pour voir comment cela fonctionne, retournons un instant vers la fonction **PLUS** du début de ce chapitre. Rappelez-vous que notre problème consistait en la définition d'une fonction à

nombre variable d'arguments. En supposant que la fonction d'addition standard + ne prenne que deux arguments, nous voulions écrire une fonction **PLUS** qui calcule la somme d'un ensemble arbitraire de nombres. Etant donné que chaque paramètre de la liste de variables est lié normalement à un seul et unique argument, nous avions besoin d'une construction simulant une infinité de paramètres. En Le_LISP, la solution à ce problème consiste en la liaison d'une variable – *l'unique* variable, représentant la *liste* de variables – à la liste des valeurs de *tous* les arguments.

En CommonLISP, le même effet peut être obtenu grâce à l'utilisation de *mots-clefs* à l'intérieur de la liste de variables. Ainsi, cette même fonction **PLUS** s'écrirait, en CommonLISP, comme suit :

```
(DEFUN PLUS (&REST L) (PPLUS L))

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

Le symbol spécial **&REST** est un *mot-clef* indiquant à CommonLISP de lier la variable suivant ce mot-clef à la liste des valeurs de *tous* les arguments restant. Ainsi, si nous appelons la fonction **PLUS** comme :

```
(PLUS 1 2 3 4 5)
```

la variable **L** – la variable suivant immédiatement le mot-clef **&REST** – sera liée à la liste

```
(1 2 3 4 5)
```

et cette liste sera envoyée à la fonction **PPLUS** qui calculera incrémentalement la somme de ces nombres, i.e: 15.

D'ailleurs, aucun *vrai* programmeur CommonLISP ne définira la fonction **PLUS** de la manière dont nous venons de le faire, c'est à dire en utilisant une fonction auxiliaire **PPLUS**. C'était juste notre manière de faire une transition de Le_LISP vers CommonLISP. En CommonLISP nous utiliserions la fonction **REDUCE** et écririons **PLUS** comme ci-dessous :

```
(DEFUN PLUS (&REST L) (REDUCE #'+ L))
```

qui fait exactement le même calcul, mais de manière plus efficace et plus élégante – par élégant nous entendons juste : cette définition est plus courte et utilise moins d'espace liste.

Remarquez que nous avons quoté la fonction + d'une manière particulière : au lieu de seulement utiliser le caractère quote ('), nous l'avons précédé par le caractère # (qui s'appelle *dièse* en français et *sharp sign* en anglais). Ce n'était pas réellement nécessaire, mais c'est un moyen de dire à CommonLISP que l'objet quoté réfère à une fonction, LISP peut alors traiter l'expression de manière plus efficace que ne l'aurait permis l'écriture avec un quote-caractère seul.

Le paramètre *rest* peut être combiné avec des paramètres normaux *si* il se trouve à la fin de la

liste de variables. Par exemple, si nous voulions écrire une fonction qui multiplie la somme d'un nombre arbitraire de paramètres par une valeur donnée, nous pourrions écrire :

```
(DEFUN MULT-PLUS (FACTEUR &REST L)
     (* FACTEUR (REDUCE #'+ L)))
```

qui, si l'on l'appelle comme :

```
(MULT-PLUS 10 1 2 3 4 5)
```

lie la variable **FACTEUR** à la valeur 10, la valeur du premier argument, et la variable **L** à la liste **(1 2 3 4 5)**, la liste des valeurs du reste des arguments. Naturellement, cet appel retourne la valeur 150.

De même qu'en Le_LISP, en CommonLISP il n'est pas possible d'appeler récursivement une fonction ayant un paramètre *rest*. Regardez par exemple la définition suivante de la fonction **PLUS** :

```
(DEFUN PLUS (&REST L) (COND
     ((NULL L) 0)
     (T (+ (CAR L) (PLUS (CDR L))))  ))
```

Cette définition de fonction est fausse, pour la même raison que la fonction Le_LISP correspondante était fausse aussi : puisque le paramètre **L** est lié à la *liste* des valeurs de tous les arguments, le paramètre **L** sera lié à la liste contenant le résulat de l'évaluation de **(CDR L)** dans l'appel récursif de **PLUS**. Naturellement, ce n'est pas cela que nous voulons, et ce n'est surement pas un nombre arbitraire d'arguments : c'est exactement un seul argument !

De temps à autre on n'a pas besoin d'un nombre indéterminé d'arguments, mais juste de quelques-uns, d'un nombre bien déterminé d'arguments *optionnels*. Rappelons, par exemple, notre définition de la fonction **REVERSE** donnée au chapitre 6 :

```
(DEFUN REVERSE (L RES)
     (IF (NULL L) RES
          (REVERSE (CDR L) (CONS (CAR L) RES))))
```

Cette fonction doit toujours être appelée avec deux arguments, ce qui est conceptuellement un peu maladroit : si nous voulons inverser une liste, c'est *une* liste que nous voulons inverser et nous ne devrions pas être obligés de connaître *comment* cette chose est réalisée, c'est-à-dire que nous ne voulons pas être obligés de donner une valeur initiale pour l'accumulateur **RES**. Au lieu d'écrire :

```
(REVERSE '(A B C) NIL)
```

nous voudrions pouvoir écrire seulement :

```
(REVERSE '(A B C))
```

Mais avec la définition ci-dessus, ce dernier appel produit une erreur : CommonLISP nous informerait avec raison de l'absence d'un argument. Munis de nos connaissances actuelles, la seule manière permettant de circonvenir ce problème est de réécrire cette fonction comme suit :

```
(DEFUN REVERSE (L)
  (LABELS ((REV (L RES)
             (IF (NULL L) RES
                 (REV (CDR L) (CONS (CAR L) RES)))))
    (REV L NIL)))
```

C'est définitivement une solution : la fonction **REVERSE** ne prend qu'un seul argument (la liste à inverser) et elle délègue tout travail à la fonction locale **REV**, qui – elle – correspond à l'ancienne définition de **REVERSE**. Néanmoins, cette définition n'est pas trop élégante.

Le problème ici consiste dans le fait que l'argument **RES** dans la version originale est *optionnel* : **RES** est superflue pour le premier appel de **REVERSE** mais est nécessaire pour tous les appels récursifs.

En CommonLISP, le mot-clef **&OPTIONAL** existe pour exactement ce type de problème : il indique que le paramètre suivant ce mot-clef peut ou ne peut pas obtenir des valeurs par des appels de la fonction. Par exemple, en utilisant ce mot-clef, nous écrivons **REVERSE** comme suit :

```
(DEFUN REVERSE (L &OPTIONAL RES)
    (IF (NULL L) RES
        (REVERSE (CDR L) (CONS (CAR L) RES))))
```

Si nous appelons cette fonction comme :

```
(REVERSE '(A B C))
```

la variable **L** sera liée à la liste **(a b c)**, et le paramètre optionnel **RES** sera lié à la valeur **NIL**, *la valeur par défaut* pour des paramètres optionnels. Naturellement, nous pouvons toujours appeler **REVERSE** avec un deuxième argument explicite (ce qui est le cas dans tous les appels récursifs), ainsi, par exemple, l'appel

```
(REVERSE '(A B C) '(A B C))
```

ramène la liste **(C B A A B C)**

Prenons un autre exemple. Voici la fonction **FOO** qui multiplie un nombre **X** par un autre nombre **Y**. Si aucune valeur n'est donnée pour **Y**, **X** sera multiplié par 2 :

```
(DEFUN FOO (X &OPTIONAL Y)
     (* X (OR Y 2)))
```

Et voici deux exemples de son utilisation :

```
(FOO 24)     →  48
(FOO 24 5)   →  120
```

FOO est une fonction *permettant* deux arguments : le premier, **X**, est un argument obligatoire; s'il n'est pas donné, CommonLISP livre un message d'erreur. Le second argument, **Y**, est optionnel; s'il n'est pas donné, CommonLISP l'initialise à **NIL**.

Il est possible de déterminer de manière explicite des valeurs par défaut pour les arguments optionnels. Une valeur par défaut est cette valeur qu'une variable prend si aucun argument correspondant n'est donné. Dans notre fameuse fonction **FOO**, nous avons déterminé la valeur par défaut 2 à l'intérieur du corps de la fonction – à travers l'évaluation de l'expression **(OR Y 2)**. Nous aurions aussi bien pu écrire cette fonction plus élégamment comme :

```
(DEFUN FOO (X &OPTIONAL (Y 2)) (* X Y))
```

Ici nous utilisons la possibilité de CommonLISP de faire suivre le mot-clef **&OPTIONAL** non seulement par le nom de ce paramètre, mais par une liste composée du nom du paramètre et de sa valeur par défaut. C'est très utile chaque fois que nous avons besoin d'une valeur par défaut différente de **NIL**.

Bien entendu, on peut combiner dans une liste de variables des paramètres obligatoires, optionnels et reste – *toujours dans cet ordre* : d'abord les paramètres obligatoires, ensuite les paramètres optionnels et, à la fin de la liste de variables, le paramètre reste. Pour voir comment les liaisons sont faites, examinons l'exemple simple ci-dessous :

```
(DEFUN BAR (X &OPTIONAL Y &REST Z) (LIST X Y Z))
```

Dans cette fonction, le paramètre obligatoire est **X**, le paramètre optionnel est **Y** et le paramètre reste est **Z**. Regardons quelques appels exemples de cette fonction :

```
(BAR 1 2 3)          →  (1 2 (3))
(BAR 1 2)            →  (1 2 NIL)
(BAR 1)              →  (1 NIL NIL)
(BAR 1 2 3 4 5 6)    →  (1 2 (3 4 5 6))
```

CommonLISP permet un troisième élément dans la liste suivant le paramètre optionnel. Ce troisième argument permet de distinguer, à l'intérieur du corps de la fonction, si une valeur a été donnée ou non au paramètre optionnel. Par exemple, dans la définition de la fonction **FOO** ci-dessous[1] :

```
(DEFUN FOO (X &OPTIONAL (Y 2 YINDIC))
    (IF (NULL YINDIC)
        (PRINC "le facteur par defaut est 2"))
    (* X Y))
```

YINDIC sera liée à **NIL** si le second argument n'a pas été donné, sinon sa valeur sera **T**. Voici quelques exemples :

1. Cet exemple a été inspiré par un exemple similaire dans le livre de Wilensky *Common LISP Craft*.

```
? (FOO 1)
 le facteur par defaut est 2
=2

? (FOO 2 3)
=6

? (FOO 100)
 le facteur par defaut est 2
=200
```

Munis des ces deux mots-clef, **&OPTIONAL** et **&REST**, nous pouvons organiser nos listes de variables de manière très flexible. Il reste juste un problème : que faire si l'on a plusieurs paramètres optionnels indépendants ?

Prenons un exemple abstrait :

(DEFUN EXEMPLE (&OPTIONAL X Y) ...)

Si nous appelons cette fonction sans arguments, comme dans

(EXEMPLE)

X et **Y** seront liées à **NIL**. L'appel :

(EXEMPLE 1)

lie la variable **X** à 1 et **Y** à la valeur par défaut **NIL**, et l'appel :

(EXEMPLE 1 2)

lie **X** à 1 et **Y** à 2. Il n'y a pas de moyen de ne donner qu'un argument pour **Y**. Si nous voulions faire cela, nous serions obligés de donner un premier argument **NIL** explicite, comme dans :

(EXEMPLE NIL 2)

même si ce n'est que la liaison de **Y** qui nous intéresse. Bien que cela ait l'air un peu ésotérique ici, il arrive plutôt souvent pendant la construction de programmes LISP, qu'on ait affaire à plusieurs arguments optionnels indépendants. Dans tous ces cas, donner la valeur explicite **NIL** pour tous les arguments précédents celui qui intéresse – en supposant que l'utilisateur connaisse l'ordre des arguments optionnels – semble plutôt lourd. C'est pourquoi CommonLISP offre un troisième mot-clef : le mot-clef **&KEY**. Pour référer à un paramètre clef (c'est comme cela que nous nommerons les paramètres définis par le mot-clef **&KEY**) dans un appel de fonction, l'utilisateur donne comme argument le *nom de la clef* (précédé par le caractère ':') et ensuite l'argument que ce paramètre doit prendre.

Dans l'exemple ci-dessus, ceci pourrait s'écrire :

(DEFUN EXEMPLE (&KEY X Y) ...)

et des appels possibles pourraient être :

```
(EXEMPLE :X 1 :Y 2)
(EXEMPLE :Y 2 :X 1)
(EXEMPLE :Y 2)
(EXEMPLE :X 1)
(EXEMPLE)
```

Dans les deux premiers appels la variable **X** sera liée à 1 et la variable **Y** à la valeur 2. Le troisième appel exemple lie **X** à **NIL** et **Y** à 2. Le quatrième appel lie **X** à 1 et **Y** à **NIL**. Finalement, le dernier appel lie **X** et **Y** à **NIL**.

Comme tous les autres paramètres optionnels, les paramètres clefs peuvent être définis avec une valeur initiale différente de **NIL** et un indicateur disant si un argument a été donné ou non dans l'appel. La syntaxe est la même que celle du mot-clef **&OPTIONAL**.

Construisez donc quelques fonctions utilisant ces divers mots-clefs et essayez de comprendre ces divers mécanismes de liaisons.

Pour terminer ce chapitre, voici deux définitions de fonctions avec quelques appels exemples :[2]

```
? (DEFUN EXEMPLE1 (&KEY X Y) (LIST X Y))
= EXEMPLE1

? (EXEMPLE1 :X 1 :Y 2)
= (1 2)

? (EXEMPLE1 :Y 2 :X 1)
= (1 2)

? (EXEMPLE1 :Y 1)
= (NIL 1)

? (EXEMPLE1)
= (NIL NIL)

? (DEFUN EXEMPLE2 (A &OPTIONAL (B 3) &REST X &KEY C (D A))
        (LIST A B C D X))
= EXEMPLE2

? (EXEMPLE2 1)
= (1 3 NIL 1 NIL)

? (EXEMPLE2 1 2)
= (1 2 NIL 1 NIL)
```

2. le deuxième exemple est tiré du livre de Guy Louis Steele *CommonLISP*.

```
? (EXEMPLE2 :C 8)
=(:C 8 NIL :C NIL)

? (EXEMPLE2 1 2 :C 8)
=(1 2 8 1 (:C 8))

? (EXEMPLE2 1 2 :D 3)
=(1 2 NIL 3 (:D 3))

? (EXEMPLE2 1 2 :D 8 :C 9 :D 10)
=(1 2 9 8 (:D 8 :C 9 :D 10))
```

11.2.1 EXERCICES

1. Définissez une fonction **TIMES** à nombre d'arguments quelconque qui calcule le produit de tous ses arguments. Exemples :

```
(TIMES 1 2 3 4)                                          →  24
(TIMES (CAR '(1 2 3)) (CADR '(1 2 3)) (CADDR '(1 2 3)))  →  6
```

2. Définissez une fonction **SLIST**, à nombre d'arguments quelconque, qui construit une liste contenant *les valeurs* des arguments de rang pair. Exemples :

```
(SLIST 1 2 3 4 5 6)     →  (2 4 6)
(SLIST 1 2 3 4 5 6 7)   →  (2 4 6)
(SLIST 1 'A 2 'B 3 'C)  →  (A B C)
```

3. Finalement, définissez une fonction **MAX** qui trouve le maximum d'une suite de nombres et une fonction **MIN** qui trouve le minimum d'une suite de nombres. Voici quelques exemples d'appels :

```
(MAX 1 2 10 -56 20 6)                          →  20
(MIN 1 2 10 -56 20 6)                          →  -56
(MAX (MIN 0 1 -34 23 6) (MIN 36 37 -38 476 63))  →  -34
```

4. Au chapitre 7 nous avons défini la fonction **RACINE-ENTIER** comme ci-dessous :

```
(DEFUN RACINE-ENTIER (N)
  (LABELS ((AUXILIAIRE (P K N)
             (IF (< N 0) K
                 (AUXILIAIRE (+ P 2) (1+ K) (- N (+ 2 P))))))
    (AUXILIAIRE 1 0 N)))
```

Réécrivez cette fonction *sans* utiliser une fonction locale auxiliaire, mais sans changer néanmoins la forme des appels, c.à.d.: **RACINE-ENTIER** doit toujours être appelé avec un unique argument.

12. LES FONCTIONS EVAL ET APPLY

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

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

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

(HEAL TEAL TELL TALL TAIL)

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

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

ce qui ramène la valeur

(TEAL TELL TALL TAIL)

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

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

Ce dernier produisant comme résultat la liste :

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

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

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

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

à sa valeur :

(TEAL TELL TALL TAIL)

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

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

12.1 LA FONCTION EVAL

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

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

est la liste

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

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

(TEAL TELL TALL TAIL)

ce qui est également la valeur de l'expression :

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

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

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

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

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

Si maintenant nous donnons à LISP l'expression :

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

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

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

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

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

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

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

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

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

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

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

Voici deux exemples d'appels de cette fonction :

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

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

```
(DEFUN CALCUL (L) (EVAL (PREFIX L)))
```

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

```
(CALCUL '(1 + 2 + 3))                    →  6
(CALCUL '(1 + 2 + (3 * 4) + (80 / 10)))  →  23
```

et finalement :

```
(CALCUL '(2 * 3 * (1 - 5) + 10))         →  36
```

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

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

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

```
(LET ((LE 'THE) (X 'LE)) (PRIN1 X) (PRINT (EVAL X)))
```

qui imprime

```
LE THE
```

et ramène l'atome **THE** en valeur.

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

noms :	X	LE	THE
valeurs :	LE	THE	UNDEFINIE

12.2 LA FONCTION APPLY

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

APPLY est syntaxiquement définie comme :

(**APPLY** *fonction liste*)

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

Donnons quelques exemples :[3]

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

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

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

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

   et

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

2. ensuite elle évalue cette expression nouvellement construite.

Il faut avoir à l'esprit que cette fonction :

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

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

(APPLY *fonction liste***)** ≡ **(EVAL (CONS** *fonction liste1***))**

où *liste1* est la valeur de *list* (le deuxième argument pour **APPLY**) où chacun des éléments est quoté.

La grande utilité de cette fonction est qu'elle résout le problème que nous avions au chapitre

3. Rappelons qu'en CommonLISP il est préférable de quoter les objets fonctionnels avec un *dièse* ('#'). Ainsi, par exemple :

   (APPLY '+ '(20 80))

   sera mieux écrit comme :

   (APPLY #'+ '(20 80))

précédent, à savoir : comment appeler récursivement une fonction utilisateur de type NEXPR.

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

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

(DE PLUS L (PPLUS L))  ; en Le_LISP ;

(DEFUN PLUS (&REST L) (PPLUS L)) ; en CommonLISP ;
```

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

```
(DEFUN PLUS (&REST L)
  (IF (NULL L) 0
    (+ (CAR L) (APPLY 'PLUS (CDR L)))))
```

Bien évidemment, nous aurions aussi bien pu écrire :

```
(DEFUN PLUS (&REST L)
  (IF (NULL L) 0
    (+ (CAR L) (EVAL (CONS 'PLUS (CDR L))))))
```

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

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

```
(DEFUN P (X Y) (+ X Y))
```

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

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

12.3 EXERCICES

1. Reprenez chacun des trois premiers exercices du chapitre précédent et transformez les solutions de manière à inclure des appels récursifs utilisant la fonction **EVAL** ou la fonction **APPLY**.

2. Ecrivez un petit traducteur du français vers l'anglais. Pour l'instant ignorez toute question de grammaire : traduisez mot par mot.

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

   (DEFUN CHAT () 'CAT)

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

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

   Bien évidemment, il faut absolument éviter de définir des fonctions de traduction pour des mots qui correspondent aux noms des fonctions standard, comme, par exemple, **DEFUN** en CommonLISP ou la fonction **DE** en Le_LISP.

13. LES ENTREES / SORTIES (DEUXIEME PARTIE)

Tant que tous les objets de travail sont soit déjà à l'intérieur du programme, soit donnés en argument aux appels des fonctions, le système LISP se charge, de manière implicite, de lire les données. Par contre, si nous voulons au fur et à mesure de l'exécution d'un programme entrer des données supplémentaires, nous devons utiliser des appels explicites à des fonctions de lecture. Tout système LISP a au moins une telle fonction : c'est la fonction **READ**. Elle n'a pas d'arguments, et ramène en valeur l'expression LISP lue dans le flot d'entrée.

Avant de continuer avec cette fonction de lecture, esquissons les problèmes liés aux entrées/sorties : dans l'interaction normale avec votre système LISP, vous entrez les fonctions et les données sur le clavier de votre terminal et vous recevez les résultats imprimés sur l'écran de votre terminal. Ceci veut dire : dans le cas standard, le *fichier-d'entrée* de LISP est le clavier, et le *fichier-de-sortie* de LISP est l'écran. Ce n'est pas obligatoire. Eventuellement vous voulez écrire les résultats du calcul dans un fichier disque, pour le garder afin de l'utiliser ultérieurement, ou vous voulez les écrire sur une bande magnétique, une cassette ou même un ruban perforé (si ça existe encore !). De même, vos entrées peuvent venir non seulement du clavier, mais également d'un fichier disque, d'un lecteur de cartes, d'un lecteur de ruban, d'une bande magnétique ou d'une cassette. Dans ces cas, vous dites préalablement à LISP, que vous voulez changer de *fichier d'entrée* ou de *fichier de sortie*. Nous verrons dans la suite comment le faire. Rappelons-nous pour l'instant, que les entrées/sorties sont *dirigées* vers des *fichiers d'entrées/sorties* qu'on peut déterminer par programme, et qui sont par défaut (et jusqu'à nouvel ordre) le clavier, pour l'entrée de données, et l'écran de visualisation pour la sortie des résultats de calcul.

13.1 LA FONCTION GENERALE DE LECTURE : READ

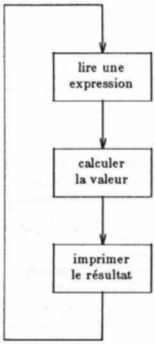
Revenons pour l'instant à la fonction **READ**. Voici sa définition :

(READ) → *expression lue*
READ est une fonction sans argument qui lit une expression LISP (soit une liste, soit un atome) dans le flux d'entrée et qui ramène cette expression en valeur.

La fonction **READ** n'évalue donc pas l'expression qu'elle lit.

Cette fonction est primordiale dans l'interprète LISP même. Un interprète est un programme qui permet à la machine de comprendre un langage de programmation. Un interprète LISP est un programme qui permet à l'ordinateur de comprendre le langage LISP. Il se compose (au minimum) de trois parties : une partie qui lit ce que l'utilisateur (le programmeur) entre dans la machine, une partie qui calcule la valeur de ce qui a été lu, et une troisième partie qui imprime

le résultat du calcul. Normalement, un interprète répète infiniment ces trois opérations :



Voici un petit interprète LISP, écrit en LISP :

```
(DEFUN LISP ()
   (FORMAT T "--> ~A~%" (EVAL (READ)))
   (LISP))
```

C'est un programme qui ne se termine jamais ! Voici une petite trace de l'utilisation de cette fonction :

? (LISP)	*le point d'interrogation est imprimé par LISP, indiquant que LISP attend une expression à évaluer. On lance la fonction* **LISP** *définie ci-dessus.*
? 1	*c'est toujours LISP qui imprime le point d'interrogation. Plus précisément : la fonction* **READ** *imprime un point d'interrogation chaque fois qu'elle attend une entrée du terminal. On donne l'expression numérique* **1** *à évaluer.*
--> 1	*notre fonction* **LISP** *a évalué l'expression et imprime sa valeur, précédée du signe '-->'*
? (CAR '(A B C))	**LISP** *s'appelle récursivement et redemande donc une expression à évaluer. On demande le* **CAR** *de la liste* **(A B C)**.
--> A	*et, encore,* **LISP** *nous donne le résultat de l'évaluation (l'atome* **A***), précédée du signe '-->'*

? *et ainsi de suite. Visiblement, tout se passe exactement de la même manière que pendant une interaction avec l'interprète LISP ordinaire, excepté que le résultat est précédé du signe --> et non pas du signe '='.*

Pour donner un autre exemple d'utilisation de la fonction **READ**, reprenons le petit calculateur que nous avons décrit au chapitre précédent. La fonction principale s'appelait **CALCUL**. Nous allons construire un programme qui va lire une expression arithmétique après l'autre, l'envoyer à la fonction **CALCUL** pour calculer la valeur de cette expression, imprimer cette valeur, et redemander une nouvelle expression. Ainsi de suite jusqu'à ce qu'on donne l'atome **FIN** : on sort alors de cette fonction calculateur et on se remet à l'état normal. C'est encore un interprète, mais cette fois un interprète d'expressions arithmétiques :

```
(DEFUN CALCULATEUR ()
   (FORMAT T "donnez une expression arithmétique s.v.p.")
   (LABELS ((CALCULATEUR-INTERNE (X) (COND
            ((EQ X 'FIN) 'FINI)
            (T (FORMAT T
                  "~A~%une autre expression arithmétique s.v.p."
                  (CALCUL X))
               (CALCULATEUR-INTERNE (READ))))))
      (CALUCALATEUR-INTERNE (READ))))
```

Ci-dessous un instantané de l'utilisation de cette fonction :

```
? (CALCULATEUR)                          ; on le lance
 donnez une expression arithmétique s.v.p.  ; l'invitation
? (1 + 2 + 3)                            ; la première expression arithmétique
 6                                       ; et le résultat
 une autre expression arithmétique s.v.p.  une nouvelle invitation
? (1 + 2 + (3 * 4) + (80 / 10))
 23
 une autre expression arithmétique s.v.p.
? (1 + 2 + 3 * 4 + 80 / 10)
 39
 une autre expression arithmétique s.v.p.
? FIN                                    l'entrée indiquant que nous voulons
                                         terminer la session de calcul
= FINI                                   on sort en ramenant l'atome FINI en
                                         valeur
?                                        voilà, on est retourné vers LISP
```

13.2 LES AUTRES FONCTIONS D'ENTREE / SORTIE : READ-CHAR, PEEK-CHAR ...

Revenons vers les fonctions de lecture. Les entrées se font – comme les sorties – en deux temps : d'abord le remplissage d'un tampon (le buffer d'entrée) à partir d'un périphérique d'entrée, ensuite l'analyse – par LISP – de ce tampon. L'analyse ne débute normalement qu'après lecture d'une ligne complète.

La fonction **READ** réalise donc deux opérations : si le buffer d'entrée est vide, elle lance le programme auxiliaire qui est chargé des opérations de lectures physiques (lire physiquement l'enregistrement suivant dans le flux d'entrée), ensuite elle lit une expression LISP à partir de ce buffer (l'expression LISP pouvant être un atome ou une liste). Si l'expression n'est pas complète à la fin de l'enregistrement (un enregistrement est une ligne de texte si l'entrée se fait à partir du terminal), **READ** lance une nouvelle lecture physique.

Imaginez que vous êtes en train d'exécuter un appel de la fonction **READ** et que vous tapez alors la ligne suivante :

(CAR '(A B C)) (CDR '(A B C)) <R>[1]

Cette ligne sera mise – par le programme de lecture physique – dans le buffer d'entrée. Au début de l'analyse de la ligne lue, nous avons donc le tampon suivant :

```
(CAR '(A B C)) (CDR '(A B C)) <R>
↑
```

avec un pointeur vers le début du buffer. Ce pointeur est représenté ici par la flèche en dessous du rectangle représentant le buffer.

Après l'exécution du **READ**, *seule* la première expression aura été lue. Le pointeur du buffer sera positionné à la fin de cette première expression, au début de la suite de caractères non encore analysée.

```
(CAR '(A B C)) (CDR '(A B C)) <R>
               ↑
```

Il faudra un appel supplémentaire de la fonction **READ** pour *lire* également la deuxième expression. (Voyez-vous la différence entre lecture physique et lecture LISP ?) Evidemment, si LISP rencontre pendant une lecture le caractère <RETURN> (représenté ici par le signe '<R>'), LISP relance cette fonction auxiliaire de lecture physique.

Ce qui est important, c'est de bien voir la différence entre lecture physique et analyse du texte lu. La fonction **READ** peut induire ces *deux* processus.

Si dans l'état

```
(CAR '(A B C)) (CDR '(A B C)) <R>
               ↑
```

nous appelons la fonction **READ-CHAR**[2], LISP va lire *sans faire une analyse quelconque* le *caractère* suivant du buffer d'entrée. *Lire* veut dire ici : ramener le caractère suivant en valeur et avancer le pointeur du buffer d'entrée d'une position. Après nous nous trouvons donc dans

1. le signe <R> indique ici le caractère <RETURN>

2. La fonction CommonLISP **READ-CHAR** s'appelle **READCH** en Le_LISP.

l'état suivant :

```
(CAR '(A B C)) (CDR '(A B C)) <R>
               ↑
```

Le caractère lu est (dans ce cas-ci) le caractère <ESPACE>, normalement un caractère séparateur ignoré.

Voici la définition de la fonction **READ-CHAR** :

(READ-CHAR) → *caractère*
lit le caractère suivant dans le flot d'entrée et le ramène en valeur sous forme d'un atome mono-caractère. **READ-CHAR** lit vraiment n'importe quel caractère.

Cette fonction sert beaucoup si vous voulez lire des caractères non accessibles par la fonction **READ**, tels que les caractères '(', ')', ';', '.', <RETURN> ou <ESPACE>.

La fonction **PEEK-CHAR**[3] est exactement comme la fonction **READ-CHAR** mais elle ne fait pas avancer le pointeur du buffer d'entrée. Le caractère lu reste donc disponible pour le **READ-CHAR** ou **READ** suivant. Voici sa définition :

(PEEK-CHAR) → *caractère*
lit le caractère suivant dans le flot d'entrée et le ramène en valeur sous forme d'un atome mono-caractère. **PEEK-CHAR** lit vraiment n'importe quel caractère, mais le laisse disponible dans le flot d'entrée pour des lectures suivantes.

En plus des fonctions **READ-CHAR** et **PEEK-CHAR**, CommonLISP offre la fonction **READ-CHAR-NO-HANG**. Cette fonction est comme **READ-CHAR** *si* un caractère est disponible dans le buffer d'entrée, mais elle retourne **NIL** dans le cas contraire – au contraire de la fonction **READ-CHAR** qui, elle, lance une nouvelle lecture physique si aucun caractère n'est disponible. Ceci est très utile si l'on veut construire des programmes à réaction immédiate, par exemple agir dès que quelqu'un a tapé un caractère sur le clavier.

La fonction de sortie correspondante à **READ-CHAR** est la fonction **WRITE-CHAR**, qui prend comme argument un caractère et l'imprime sur le flux de sortie.

Le_LISP met deux fonctions d'entrée/sortie immédiates à votre disposition : les fonctions **TYI** et **TYO**. *Immédiat* veut dire que ces deux fonctions n'utilisent pas les tampons d'entrée/sortie. Voici leurs définitions respectives :

3. La fonction CommonLISP **PEEK-CHAR** s'appelle **PEEKCH** en Le_LISP.

(TYI) → *valeur numérique*
Cette fonction lit un caractère du *terminal* et retourne la valeur ascii du caractère *aussitôt* après la frappe de ce caractère. Nul besoin de terminer la ligne par un <RETURN> ! Le caractère tapé n'apparaît pas sur l'écran, on dira que l'écho est inhibé.

(TYO *n*) → *n*
Cette fonction affiche sur l'écran, à la position courante du curseur, le caractère dont la valeur ascii est *n*.

Remarquons qu'en LE_LISP, la fonction **TYO** admet un nombre quelconque d'arguments et affiche donc la suite de caractères correspondante.

Ces deux fonctions sont complémentaires, ainsi si vous voulez un écho (c'est-à-dire : si vous voulez imprimer le caractère frappé) pour un appel de la fonction **TYI**, il suffit d'écrire :

(TYO (TYI))

TYO vous permet de sortir n'importe quel caractère sur votre terminal. Si vous voulez, par exemple, faire faire un petit 'beep' à votre terminal, il suffit d'appeler **(TYO 7)**, où 7 est le code ascii du caractère <BELL>.

Naturellement, si vous voulez faire la même chose en CommonLISP (c.à.d.: produire un 'beep' sur le terminal), vous devez utiliser la fonction **WRITE-CHAR** avec <ctrl>G comme argument. Ainsi :

(WRITE-CHAR #\^G)

où le **^G** est obtenu en enfonçant la touche <ctrl> en même temps que la touche **G**, produira un petit 'beep' sur votre terminal.

Bien évidemment, ces quatre dernières fonctions sont les fonctions d'entrée/sortie les plus élémentaires de LISP. Avec elles vous pouvez écrire toutes les interfaces possibles.

Après toute cette théorie il nous faut absolument revenir vers un peu de pratique de programmation, afin de voir comment se servir de ces fonctions.

En Le_LISP, la fonction **TYI** est très souvent utilisée pour la programmation de jeux interactifs (à cause de son effet immédiat dès que vous avez tapé le caractère demandé, et donc à cause de la grande nervosité des programmes utilisant cette forme de lecture) ou pour éviter un défilement trop accéléré des choses que vous imprimez (puisque la machine *attend* la frappe d'un caractère), mais le caractère particulier que vous donnez n'a pas vraiment d'importance.

Pour illustrer ce dernier point, ci-dessous la fonction **COMPTE** qui imprime tous les nombres de **X** jusqu'à 0, l'un en-dessous de l'autre :

```
(DEFUN COMPTE (X)
    (IF (LE X 0) "boum"
        (PRINT X)
        (COMPTE (1- X))))
```

Si vous appelez cette fonction avec **(COMPTE 4)**, elle vous imprime :

```
4
3
2
1
= boum
```

Le problème avec cette fonction est que, si vous l'appelez avec, par exemple, le nombre 1000, les nombres successifs seront imprimés à une telle vitesse que la lecture ne pourra plus suivre leur défilement sur l'écran. Pour prévenir cela, dans la deuxième version un compteur qui compte les lignes imprimées a été inseré. Après l'impression de 23 lignes,[4] le programme attend que vous tapiez un caractère quelconque avant de continuer. De cette manière vous pouvez interactivement déterminer la vitesse d'affichage :

```
(DE COMPTE (X)
   (LETN COMPTE-INTERNE ((X X)(Y 1))
      (IF (<= X 0) "boum"
         (PRINT X)
         (COMPTE-INTERNE (1- X) (IF (< Y 23) (1+ Y) (TYI) 1)))))
```

Observez donc la différence entre les deux versions de cette fonction.

La différence entre cette fonction Le_LISP et la version CommonLISP ci-dessous, est que **TYI** ne livre pas d'écho du caractère tapé (i.e.: il n'imprime pas ce caractère) et que **TYI** ne regarde pas dans le buffer, ce qui a comme conséquence qu'il n'est pas nécessaire de faire suivre le caractère tapé par un <return>, ce qui est nécessaire avec la fonction CommonLISP **READ-CHAR**.

Voici donc la fonction **COMPTE** en CommonLISP :

```
(DEFUN COMPTE (X)
   (LABELS ((COMPTE-INTERNE (X Y) (COND
                ((<= X 0) (TERPRI) "boum")
                (T (PRINT X)
                   (COMPTE-INTERNE (1- X) (COND
                            ((< Y 23) (1+ Y))
                            (T (READ-CHAR) 1)))))))
      (COMPTE-INTERNE X 1)))
```

4. D'ailleurs, le nombre 23 est non seulement le nombre de lignes de la majorité des terminaux actuels moins un, mais c'est également le premier nombre n, tel que la probabilité pour que deux personnes parmi n possèdent la même date anniversaire soit supérieure à ½ .

Construisons maintenant un micro-interprète LISP utilisant une fonction de lecture qui, en parallèle à la lecture, garde des statistiques sur l'utilisation des atomes : à chaque atome sera associé un compteur d'occurrences qui sera mis à jour au cours de la lecture.

Tout d'abord, construisons notre nouvelle fonction de lecture : **STAT-READ**.

```
(DEFUN STAT-READ ()
  (LET ((X (PEEK-CHAR))) (COND
        ((EQUAL X #\( ) (READ-CHAR) (READ-LIST))
        ((EQUAL X #\) ) (READ-CHAR) 'PARENTHESE-FERMANTE)
        ((MEMBER X '(#\Space #\Newline #\Return #\Tab))
                (READ-CHAR) (STAT-READ))
        (T (READ-ATOM)))))
```

Cette fonction débute, avec l'appel de la fonction **PEEK-CHAR**, par un regard tentatif vers le buffer d'entrée : si le caractère prêt à être lu est une parenthèse ouvrante, nous appelons la fonction **READ-LIST**, une fonction chargée de lire des listes. Si le caractère en tête du buffer est une parenthèse fermante, nous enlevons ce caractère du buffer (avec l'appel de la fonction **READ-CHAR**) et sortons de **STAT-READ** en ramenant l'indicateur **PARENTHESE-FERMANTE**. Dans le cas où nous trouvons un séparateur standard en tête du buffer, nous l'éliminons et continuons à examiner les caractères restants. Si aucun de ces cas ne se présente, alors nous sommes sûrs que nous avons affaire à un atome, et ce sera alors la fonction **READ-ATOM** qui se chargera de la lecture de l'atome.

Notons les divers caractères spéciaux :

#\Space #\Newline #\Return #\Tab

qui sont des écritures commodes pour les caractères <ESPACE>, <A-LA-LIGNE>, <RETURN> et le caractère <TABULATION>. Ce sont juste des *noms* pour quelques caractères spéciaux. Ils existent principalement pour garder les programmes les utilisant lisibles pour l'utilisateur humain.

Regardez bien l'utilité de la fonction **PEEK-CHAR** : c'est elle qui permet de faire toute cette analyse en cas distincts, *sans* toucher réellement au caractère analysé. Celui-ci reste disponible pour des lectures ultérieures. On aura besoin du caractère analysé dans la suite de la lecture d'un atome : il sera le premier caractère de son P-name.

La fonction **READ-LIST** doit lire un élément après l'autre de la liste à lire, jusqu'à la rencontre d'une parenthèse fermante, et construire réellement cette liste. La voici :

```
(DEFUN READ-LIST ()
    (LABELS ((READ-LIST-INTERNE (X)
        (IF (EQ X 'PARENTHESE-FERMANTE) ()
            (CONS X
                (READ-LIST-INTERNE (STAT-READ))))))
      (READ-LIST-INTERNE (STAT-READ))))
```

A l'appel de la fonction **READ-ATOM** nous savons qu'en tête du buffer nous trouvons un

atome : un appel de la fonction **READ** standard ne lira donc *que* cet atome ! Tout ce que nous avons à faire est d'incrémenter nos compteurs privés **USE**.

```
(DEFUN READ-ATOM ()
    (LET ((X (READ))) (COND
        ((NUMBERP X) X)
        (T (SETF (GET X 'USE)
                 (IF (GET X 'USE)
                     (1+ (GET X 'USE)) 1))
           X))))
```

Maintenant nous avons une fonction qui lit des expressions LISP quelconques et qui compte le nombre d'occurrences des atomes.

Voici finalement la fonction **LISP**, le moteur de notre micro-interprète :

```
(DEFUN LISP ()
   (PRINC "une expression: ")
   (LABELS ((LISP-INTERNE (X)
            (IF (NOT (EQ X 'PARENTHESE-FERMANTE))
                (PRINT (EVAL X)))
            (FORMAT T "~%une expression: ")
            (LISP-INTERNE (STAT-READ))))
     (LISP-INTERNE (STAT-READ))))
```

Ci-dessous une fonction Le_LISP qui imprime nos statistiques. Cette fonction utilise Le_LISP's fonction standard **OBLIST**, qui livre une liste de tous les atomes connus du système LISP. On appelle cette liste de tous les atomes : l'*obliste.*[5] La fonction **STAT** prend donc un élément après l'autre de l'obliste, regarde si l'indicateur **USE** se trouve sur sa P-liste et, si oui, imprime le nom de l'atome suivi du nombre d'utilisations.

```
(DE STAT ()
   (LETN STAT-INTERNE ((Y (OBLIST)))
      (IF (NULL Y) ()
         (IF (GET (CAR Y) 'USE)
             (PRINT (CAR Y) (GET (CAR Y) 'USE)))
         (STAT-INTERNE (CDR Y)))))
```

En CommonLISP il n'y a pas de fonction **OBLIST** (une fois que vous serez plus avancé dans votre connaissance de CommonLISP, essayez donc d'en définir une pour vous). Alors, afin d'obtenir les statistiques dans notre version CommonLISP du programme, nous allons légèrement modifier la fonction, de manière telle qu'elle admet les objets sur lesquels on veut avoir des statistiques en argument. Ci-dessous il y a la fonction CommonLISP qui imprime nos statistiques. Cette fonction, **STAT**, prend comme arguments les divers atomes sur lesquels nous voulons avoir des statistiques. Elle prend un atome après l'autre de cette liste, regarde si l'indicateur **USE** est positionné, et si oui, elle imprime le nom de l'atome suivi du nombre indiquant combien de fois cet atome a été utilisé. Notez que, afin de pouvoir ignorer les nombres d'occurrences des atomes donnés en argument, nous décrémentons leurs compteurs associés.

5. L'obliste peut être considérée comme un sorte de micro-manuel LISP, accessible en ligne.

Ceci est nécessaire puisque tout appel de la fonction **STAT** sera évalué à l'intérieur de notre micro-évaluateur LISP privé, qui, lui, compte tous les atomes lus : nous ne voulons pas compter les occurrences à l'intérieur des appels de **STAT**.

Donc, voici finalement la fonction :

```
(DEFUN STAT (&REST QQ-OBJETS) (COND
      ((NULL QQ-OBJETS) NIL)
      (T (WHEN
           (GET (CAR QQ-OBJETS) 'USE)
                (SETF (GET (CAR QQ-OBJETS) 'USE)
                      (1- (GET (CAR QQ-OBJETS) 'USE)))
                (FORMAT T "~%~A ~A"
                      (CAR QQ-OBJETS)
                      (GET (CAR QQ-OBJETS) 'USE)))
         (APPLY 'STAT (CDR QQ-OBJETS)))))
```

Testons donc cette nouvelle fonction. Voici le début d'une interaction possible :

```
? (LISP)                          on lance la fonction interprète
une expression: ?  1              évaluons le nombre 1
1                                 la valeur de l'évaluation
une expression: ?  (QUOTE a)      une deuxième expression
a                                 sa valeur
une expression: ?  'a             une troisième expression
<un message d'erreur>             sa valeur ?
```

Que se passe-t-il ?

C'est très compliqué ! Le tout est évidemment dû au quote-caractère. Rappelez-vous ce que nous disions de ce caractère au chapitre 2 : nous insistions sur le fait que le caractère ' n'est qu'une abréviation commode pour un appel de la fonction **QUOTE**, et si vous donnez à la machine LISP l'expression

```
'a
```

pour elle tout se passe comme si vous aviez donné l'expression

```
(QUOTE a)
```

C'est effectivement cette deuxième expression qui aura été enregistrée par le lecteur LISP. Cette transformation se joue directement au niveau de la lecture. Des caractères qui modifient le comportement du lecteur LISP s'appellent des *macro-caractères*. Nous les verrons plus en détail au paragraphe suivant.

Regardons quand même ce qui se passait ici :

1. Tout d'abord, la fonction **LISP** appelle la fonction **STAT-READ**.

2. La fonction **STAT-READ** teste les caractères se trouvant en tête du buffer d'entrée (ici nous trouvons la suite de caractères "a") pour des occurrences de caractères spéciaux tels que ')', '(', '<ESPACE>', '<TABULATION>' ou '<RETURN>'. Aucun de ces caractères n'étant présent, **STAT-READ** appelle donc la fonction **READ-ATOM**.

3. La fonction **READ-ATOM** débute par un appel, dans la ligne :

   (LET ((X (READ))) ...

   de la fonction **READ** standard. La valeur de la variable **X** sera, d'après tout ce que nous savons maintenant, la liste :

   (QUOTE a)

   C'est cette liste qui sera le premier argument de la fonction **GET**. Horreur ! Nous étions sûrs que la valeur de **X** serait toujours un atome; ici c'est une liste ! D'ailleurs, la fonction **GET** attend un argument de type atome. Que fait donc la fonction **GET** ? Elle génère une erreur.

La correction de cette erreur consistera à insérer dans la fonction **STAT-READ** une ligne spéciale traitant le macro-caractère "'"; ce qui nous donne la version suivante :

```
(DEFUN STAT-READ ()
  (LET ((X (PEEK-CHAR))) (COND
          ((EQUAL X #\( ) (READ-CHAR) (READ-LIST))
          ((EQUAL X #\) ) (READ-CHAR) 'PARENTHESE-FERMANTE)
          ((EQUAL X #\' ) (READ-CHAR)
                          (LIST 'QUOTE (STAT-READ)))
          ((MEMBER X '(#\Space #\Newline #\Return #\Tab))
                    (READ-CHAR) (STAT-READ))
          (T (READ-ATOM)))))
```

Ci-dessous une interaction avec cette nouvelle version de notre micro-interprète LISP :

```
? (LISP)                                  on relance la machine
une expression: ? 1                       une première expression à évaluer
1                                         sa valeur
une expression: ? '2                      une deuxième expression
2                                         sa valeur
une expression: ? 'a                      encore une expression quotée
a                                         ça semble marcher
une expression: ? (CAR '(a b c))
a
une expression: ? (CONS '(a b) '(c d))
((a b) c d)
une expression: ?
```

```
(STAT                                    regardons si ça compte bien le nombre
                                         d'occurrences de chacun des atomes
 'CAR 'CONS 'A 'B 'C 'D 'STAT)
CAR 1
CONS 1
a 3
b 2
c 2
d 1
STAT 1
nil
une expression: ? (CAR '(a b c))         continuons encore un peu
a
une expression: ? (CAR (CAR '((a b c))))
a
une expression: ?
(STAT                                    et encore une statistique
 'CAR 'CONS 'A 'B 'C 'D 'STAT)
CAR 4
CONS 1
d 1
a 5
b 4
c 4
STAT 2
nil
...
```

Bien évidemment le vrai lecteur de LISP est considérablement plus compliqué (nous en construirons un dans un chapitre ultérieur), mais ce modèle donne une bonne première approximation. Examinez bien le rôle des divers caractères séparateurs, ainsi que l'activité du quote-caractère.

13.3 LES DIVERS TYPES DE CARACTERES

13.3.1 Détermination du type d'un caractère

Les divers caractères du jeu ascii n'ont pas tous le même effet pendant la lecture. La plupart sont considérés comme des caractères constituant des noms des atomes. Par contre, les parenthèses, l'espace <ESPACE>, la tabulation <TABULATION>, le return <RETURN>, le point-virgule, le guillemet et l'accent aigu ont des rôles particuliers.

Résumons ce que nous savons jusqu'à maintenant du rôle de ces différents caractères :

<ESPACE>
<TAB>
<RETURN>

Ces caractères ont le rôle de séparateur. Normalement, ils ne peuvent pas faire partie des noms des atomes; ils servent à séparer des suites d'atomes.

;

Le point-virgule est le caractère *début de commentaire.* Le caractère <RETURN> est le caractère *fin de commentaire.* Si vous voulez mettre des commentaires à l'intérieur de votre programme, ils doivent donc débuter par un point-virgule et ils se terminent par un <RETURN>, donc à la fin de la ligne. Le lecteur LISP standard ignore tout commentaire.

"

Le guillemet délimite des chaînes de caractères. A l'intérieur d'une chaîne de caractères aucun contrôle quant au rôle des caractères n'est effectué. Par contre, comme dans le langage C, il existe une convention pour y mettre des caractères bizarres. Pour cela, il est suffisant de faire précéder le caractère spécial par un caractère *antislash* ('\'). Naturellement, les deux seuls caractères spéciaux dans une chaîne sont le guillemet (") et l'antislash même. Ainsi, pour insérer un guillemet dans une chaîne, nous devons écrire, par exemple :

```
"ab\"ba"
```

pour obtenir la chaîne **ab"ba**.

(et)

Bien évidemment, les caractères parenthèse ouvrante et parenthèse fermante débutent et finissent l'écriture de listes. Ces deux caractères sont également des séparateurs et ne peuvent donc pas, normalement, être utilisés à l'intérieur des noms des atomes.

'

Le quote-caractère indique au lecteur LISP de générer un appel de la fonction **QUOTE** avec comme argument l'élément suivant directement ce caractère. Des caractères qui modifient le comportement du lecteur LISP s'appellent des *macro-caractères.*

\

Le caractère antislash est utilisé pour quoter des caractères spéciaux, tels qu'ils perdent temporairement leur signification.

#

Nous avons déjà vu le *dièse* : 1) en combinaison avec l'antislash pour nommer des objets de type caractère, 2) en combinaison avec le quote caractère afin de quoter des objets fonctionnels. Le dièse est un caractère très spécial qui existe toujours en combinaison avec d'autres caractères. Nous allons en voir quelques cas supplémentaires dans la suite.

|

La barre verticale est, comme le guillemet toujours utilisée en paires. Les deux barres verticales servent alors à délimiter des noms d'objets qui peuvent contenir des caractères spéciaux. La différence entre la barre verticale et le guillemet est que la première délimite des chaînes de caractères dénotant des atomes, ou, comme on dit en LISP : elle délimite des chaînes de symboles internés. Le guillemet, lui, délimite toujours des chaînes de caractères.

Pour un exemple d'utilisation de la barre verticale, regardez l'exemple suivant :

(LET ((\(\a\n\ \a\t\o\m\'\s\ \v\a\l\u\e\) 3) ...

Ceci crée un symbole nommé "**(an atom's value)**" et lui donne la valeur 3. Afin d'obtenir ce symbole bien particulier, nous avons dû quoter chacun des caractères. Les caractères normaux ont été quotés afin d'inhiber leur transformation en des lettres majuscules en Common-LISP, et les caractères spéciaux (,), ' et <ESPACE> ont été quotés afin de prévenir le lecteur LISP de ne pas interpréter ces caractères.

Ce n'est sûrement pas très aisé à lire. Nous pouvons faire exactement la même chose en écrivant :

(LET ((|an atom's value)| 3) ...

où les deux barres verticales indiquent au lecteur LISP de prendre tous les caractères inclus tels quels – sans les interpréter et sans les transformer en caractères majuscules. C'est définitivement beaucoup plus facile à lire (au moins pour nous).

Tous les autres caractères (sauf le point, '.', qui sera examiné par la suite) n'ont aucun rôle particulier et peuvent donc être utilisés sans prendre des précautions particulières pour les noms des atomes.

En Le_LISP, si à un instant quelconque de votre interaction avec LISP vous ne vous rappelez plus du rôle d'un caractère, vous pouvez appeler la fonction **TYPECH** qui vous donne le *type* du caractère. Voici sa définition :

(TYPECH *e***)** → *type*
l'argument *e* doit être un objet de type caractère. *type* est le type courant de ce caractère.

(TYPECH *e type***)** → *type*
l'argument *type* doit avoir une valeur d'un type de caractère. *type* sera le nouveau type du caractère *e*.

Le type d'un caractère est défini comme suit :

type LE_LISP	*caractère*
CNULL	caractères ignorés
CBCOM	début de commentaire
CECOM	fin de commentaire
CQUOTE	caractère QUOTE en LE_LISP
CLPAR	parenthèse ouvrante
CRPAR	parenthèse fermante
CDOT	le point .
CSEP	séparateur
CMACRO	macro-caractère
CSTRING	délimiteur de chaînes
CPNAME	caractère normal
CSYMB	délimiteur symbole spécial |
CPKGC	délimiteur de package
CSPLICE	splice-macro
CMSYMB	symbole mono-caractère

Munis de ces connaissances et de cette nouvelle fonction, vous pouvez écrire des choses étranges en Le_LISP. Par exemple, si les parenthèses ne vous plaisent plus, après les instructions

```
(TYPECH #\{ (TYPECH #\())
(TYPECH #\} (TYPECH #\)))
```

Vous pouvez aussi bien écrire

```
(CAR '(DO RE MI))
```

que

```
{CAR '{DO RE MI}}
```

En CommonLISP, le typage de caractères est légèrement différent : CommonLISP distingue entre des caractères *illégaux*, *blancs*, *constituants*, d'*échappement simple*, d'*échappement multiple* et des *macro-caractères*. Les macro-caractères sont encore subdivisés en macro-caractères *terminants*

et ceux qui sont *non-terminants*.

Ci-dessous, une liste des types syntaxiques des caractères :

blanc	return, tabulation, espace, page, newline, linefeed
échappement simple	\\
ec'happement multiple	|
macro terminant	" ' ` ( ) ;
macro non-terminant	#
constituant	tous les autres caractères

Constituant veut dire que le caractère correspondant peut être utilisé à l'intérieur de noms de symboles sans aucun problème. Tous les autres caractères doivent être *échappés* si l'on veut les utiliser comme partie du nom d'un symbole. On peut échapper un caractère unique avec le caractère d'*échappement simple* (l'antislash); on peut *échapper* des caractères *multiples* avec les barres verticales. Des caractères *blancs* sont des séparateurs, ils terminent un nom de symbole ; autrement, ils n'ont aucune signification. Des *macro-caractères terminants* terminent également les noms des symboles, mais ils sont interprétés par LISP comme ayant une signification supplémentaire. Les *macro-caractères non-terminants* peuvent être partie d'un nom, aucun besoin de les échapper.

En CommonLISP, il n'existe aucune fonction donnant le type d'un caractère. Le mieux est de regarder dans le manuel ! Mais pour changer la syntaxe d'un caractère, il existe une fonction presque comme **TYPECH** : c'est la fonction **SET-SYNTAX-FROM-CHAR**. Voici sa définition :

(SET-SYNTAX-FROM-CHAR *caractère-vers caractère-de***)**

Cette fonction prend deux arguments : *caractère-vers* et *caractère-de*. Son effet est de donner au caractère *caractère-vers* la même définition syntaxique que celle du caractère *caractère-de*.

Par exemple, l'évaluation de :

(SET-SYNATX-FROM-CHAR #\^ #\')

donne au caractère circonflexe (^) la même syntaxe *et* la même fonctionnalité que celle du quote-caractère. Après cela, nous pouvons écrire :

(QUOTE A)

comme :

'A

ou :

^A

Les trois formes d'écriture ont le même effet.

Normalement, ceci fonctionne très bien avec des caractères spéciaux définis par l'utilisateur (des

macro-caractères), mais soyez très attentif quand vous manipulez des macro-caractères standard : l'effet peut être différent d'une version de CommonLISP à une autre. Surtout, notez que la fonctionnalité du caractère parenthèse fermante est trop cachée à l'intérieur du lecteur LISP, il n'existe aucun moyen simple – en CommonLISP – de transférer sa fonctionnalité vers un autre caractère. Là aussi, le mieux sera de consulter le manuel de votre version particulière de CommonLISP.

13.3.2 Les macro-caractères

Les macro-caractères modifient le comportement du lecteur LISP de manière à générer des appels de fonctions. Ainsi le quote-caractère, l'unique macro-caractère que nous connaissons actuellement, traduit :

'quelque-chose

en :

(QUOTE quelque-chose)

L'utilisateur LISP peut définir ses macro-caractères personnels grâce à une fonction de définition de macro-caractères. En Le_LISP, c'est la fonction **DMC** et en CommonLISP c'est la fonction **SET-MACRO-CHARACTER**. Puisque Le_LISP et CommonLISP traitent les macro-caractères de manière légèrement différente, nous allons d'abord examiner la manière de Le_LISP (et des dialectes similaires) pour ensuite étudier la même chose sous l'aspect CommonLISP.

13.3.2.1 *Définition de macro-caractères en Le_LISP*

La fonction **DMC**, la fonction de Le_LISP pour **D**éfinir des nouveaux **M**acro-**C**aractères, est syntaxiquement définie comme suit :

(DMC *c* **()** *corps-de-fonction*)

Le premier argument, *c*, doit être un symbole mono-caractère, de préférence écrit avec les deux barres verticales l'entourant. Lorsque le lecteur LISP rencontrera ce caractère, il lancera l'évaluation du *corps-de-fonction*, Ensuite, le résultat de l'évaluation sera placé à la place du dit caractère dans le flot d'entrée.

Par exemple, si le quote-caractère n'était pas défini, nous pourrions le définir comme :

(DMC |'| () (LIST (QUOTE QUOTE) (READ)))

Cette définition peut se lire comme un ordre au lecteur LISP disant qu'à chaque rencontre d'un caractère ' dans le flot d'entrée, il faut remplacer ce caractère par une liste dont le premier élément est l'atome **QUOTE** et le deuxième élément sera l'expression LISP qui suit directement ce caractère.

Prenons un autre exemple : construisons des macro-caractères qui nous permettent d'écrire

[*argument*$_1$ *argument*$_2$ . . . *argument*$_n$]

au lieu de

(LIST *argument*$_1$ *argument*$_2$ . . . *argument*$_n$**)**

Pour cela nous devons définir deux macro-caractères : le macro-caractère '[' qui doit construire l'appel de la fonction **LIST** avec autant d'arguments qu'il y a d'expressions LISP jusqu'au caractère ']' suivant. Ce dernier caractère doit devenir également un macro-caractère, soit seulement pour le rendre séparateur. Voici les deux définitions :

```
(DMC |[| ()
    (LETN LA-LISTE ((X 'LIST))
          (AND (NEQUAL X '|]|) (CONS X (LA-LISTE (READ))))))

(DMC |]| () '|]|)
```

La fonction **NEQUAL** est une abréviation pour **(NULL (EQUAL**,[6] de manière telle que nous avons la relation suivante :

(NEQUAL *x y***)** ≡ **(NULL (EQUAL** *x y***))**

Comme vous voyez, les définitions de macro-caractères peuvent être de complexité arbitraire. Leur rôle est, dans tous les cas, de simplifier l'écriture de programmes.

Regardons donc quelques exemples d'utilisation de ces macro-caractères :

```
[1 2 3]                    →   (1 2 3)
['a 'b (CAR '(1 2 3)) 'c]  →   (a b 1 c)
'[1 2 [3 4] 4]             →   (LIST 1 2 (LIST 3 4) 4)
(CONS 1 [2 3 4])           →   (1 2 3 4)
```

Maintenant nous pouvons utiliser le macro-caractère quote (') pour entrer des listes et les macro-caractères '[' et ']' pour générer des appels de la fonction **LIST**.

13.3.2.2 *Définition de macro-caractères en CommonLISP*

Dans ce paragraphe, nous allons examiner comment définir des macro-caractères en CommonLISP. La fonction de définition de macro-caractères de CommonLISP s'appelle **SET-MACRO-CHARACTER**.[7] Elle est définie syntaxiquement comme :

(SET-MACRO-CHARACTER *caractère fonction***)**

L'évaluation d'un appel de cette fonction transforme l'argument *caractère*, un objet de type caractère, en un macro-caractère qui, quand il est rencontré par **READ**, appellera la fonction

6. Si cette fonction n'existe pas dans votre LISP, vous pouvez la définir aisément comme :

(DEFUN NEQUAL (X Y) (NULL (EQUAL X Y)))

7. Notez l'orthographe anglaise du mot *caractère*, qui s'écrit avec *ch*.

fonction. **SET-MACRO-CHARACTER** est une fonction qui associe une fonction à un caractère et change le type de ce caractère, comme la fonction **DMC** en Le_LISP. La différence avec **DMC** est que 1) l'argument est un objet de type caractère (et non un atome mono-caractère), et 2) le deuxième argument est une fonction (et non une expression).

Supposons encore une fois que le quote-caractère ' ne soit pas défini. Nous pourrions le définir, en tant que macro-caractère, en attachant à ce caractère la fonction suivante :

```
(DEFUN MY-QUOTE (FLUX CARACTERE)
       (LIST (QUOTE QUOTE) (READ FLUX T NIL T)))
```

Examinons cette fonction en détail : tout d'abord, notez que c'est une fonction à deux arguments, **FLUX** et **CARACTERE**. A l'appel de cette fonction, c'est-à-dire : quand le lecteur LISP rencontre un quote-caractère, le premier argument reçoit un flux. Nous verrons ultérieurement comment manipuler des flux. Pour l'instant, notons juste que toute lecture est faite à travers un flux, qui peut être relié au terminal ou à un fichier (ou même à d'autres objets LISP). Par défaut, le flux de lecture est le terminal. Ordinairement, on n'est pas obligé de le spécifier : c'est la fonction **READ** qui, automatiquement, transmet son flux à la fonction associée au macro-caractère.

Le deuxième argument transmis à la fonction associée au macro-caractère est le macro-caractère lui-même. Dans cet exemple, nous n'utilisons pas du tout ce deuxième argument.

La fonction **MY-QUOTE** spécifie donc que, si elle est appelée avec un flux et un caractère, elle retourne une liste composée de deux éléments : le symbole **QUOTE** et l'expression LISP suivant directement le quote-caractère dans le flux d'entrée. Notez que la fonction **READ** possède ici quatre arguments. Cela change ce que nous avons dit sur cette fonction : **READ** est une fonction avec quatre arguments optionnels. Si aucun argument n'est donné, comme nous avons fait jusqu'à maintenant, **READ** lit une expression dans le flux actuellement actif. Le premier argument peut indiquer un flux. Ici, puisque de toute façon nous avons accès à ce flux, nous donnons ce premier argument. Les deuxième et troisième arguments indiquent ce que **READ** doit faire quand elle rencontre la fin d'un fichier. Pour l'instant notez juste que nous devons donner ces deux arguments *toujours* dans cette forme dans les appels de **READ** à l'intérieur des fonctions associées aux macro-caractères. Le quatrième argument doit être **T** si nous appelons **READ** à partir d'une fonction associée à un macro-caractère. Cet argument est nécessaire (pour d'obscures raisons d'implémentation) chaque fois qu'il y a un appel *récursif* à **READ**. Ici, *récursif* veut dire que nous appelons la fonction **READ** de l'intérieur de la fonction **READ**. Naturellement, c'est le cas ici, puisque nous voulons appeler cette fonction durant la lecture d'expressions LISP contenant des quote-caractères.

Il manque encore l'association de la fonction **MY-QUOTE** avec le quote-caractère '. Pour faire cela, nous utilisons la fonction **SET-MACRO-CHARACTER** comme suit :

```
(SET-MACRO-CHARACTER #\' #'MY-QUOTE)
```

Un appel de **SET-MACRO-CHARACTER** retourne toujours **T**.

Réécrivons en CommonLISP les macro-caractères '[' et ']', qui, comme dans le paragraphe

précédent, nous permettent d'écrire :

$$[argument_1\ argument_2\ \ldots\ argument_n]$$

au lieu de

$$(\mathbf{LIST}\ argument_1\ argument_2\ \ldots\ argument_n)$$

Afin de faire cela, comme précédemment, nous allons écrire une fonction pour le macro-caractère '[' qui doit construire l'appel de la fonction **LIST** et lire toutes les expressions jusqu'au caractère ']' suivant.

Voici une définition CommonLISP pour ce macro-caractère :

```
(DEFUN CROCHET-OUVRANT (FLUX CARACTERE)
    (LABELS ((MY-LOOP (X)
                (AND (NOT (EQUAL X '|]|))
                    (CONS X
                        (MY-LOOP (READ FLUX T NIL T))))))
        (MY-LOOP 'LIST)))

(SET-MACRO-CHARACTER #\[ #'CROCHET-OUVRANT)
```

Notez, encore, que nous avons donné deux arguments à la fonction **CROCHET-OUVRANT** : un pour le flux actuel, un pour le macro-caractère. Notez également l'appel *récursif* de **READ**. Vos appels de **READ** à l'intérieur des macro-caractères doivent *toujours* être de cette forme.

Il y a encore un problème : tel que nous avons défini le macro-caractère '[', il fonctionne sur des exemples comme :

```
[1 2 3 ]
```

ou

```
['A 'B 'C ]
```

mais, il ne fonctionne pas sur les deux exemples suivants :

```
[1 2 3]
```

ou

```
['A 'B 'C]
```

Dans le deuxième groupe d'exemples, le crochet fermant ne sera pas reconnu en tant que caractère isolé, mais fera juste partie du nom des symboles **3]** et **C]**. Jusqu'à maintenant, le crochet fermant est un caractère normal : c'est un caractère *constituant*, comme on dit en CommonLISP. Afin de le transformer en un caractère séparateur, nous devons le transformer en un

macro-caractère. Voici, donc sa définition :

```
(DEFUN CROCHET-FERMANT (FLUX CARACTERE) '|]|)

(SET-MACRO-CHARACTER #\] #'CROCHET-FERMANT)
```

Tout ce que fait ce macro-caractère, c'est de retourner le symbole], qui peut ensuite être reconnu par le crochet ouvrant.

A la fin du paragraphe précédent, vous trouvez quelques exemples d'utilisation de ces deux nouveaux macro-caractères.

13.3.2.3 *Un exemple plus élaboré*

Construisons donc un dernier jeu de macro-caractères et définissons le macro-caractère *antiquote* (‘). Ce macro-caractère, existe dans la plupart des systèmes LISP. Il est une sorte de combinaison des caractères ', [et]. On l'utilise quand on a une longue liste quotée à l'intérieur de laquelle on veut mettre un élément à évaluer, ou, autrement dit : on l'utilise quand on construit une liste ou la majorité des éléments sont quotés.

Prenons un exemple. Supposons que la variable **X** est liée à la liste **(D E)** et qu'on veuille construire une liste avec, dans l'ordre, les éléments **A**, **B**, **C**, la valeur de **X** suivie de l'élément **F**. L'unique manière disponible actuellement est d'écrire :

```
(LIST 'A 'B 'C X 'F)
```

ou

```
['A 'B 'C X 'F]
```

Si nous voulons construire à partir de ces éléments la liste :

```
(A B C D E F)
```

nous devons écrire :

```
(CONS 'A (CONS 'B (CONS 'C (APPEND X (LIST 'F)))))
```

Finalement, si nous voulons, toujours à partir de ces quelques éléments disponibles, construire la liste :

```
(A B C E F)
```

nous devons écrire :

```
(CONS 'A (CONS 'B (CONS 'C (CONS (CADR X) (LIST 'F)))))
```

Toutes ces écritures semblent bien lourdes, sachant qu'un seul élément dans la liste est calculé, tous les autres sont donnés tels quels.

Le macro-caractère *antiquote* nous livre un dispositif typographique permettant d'écrire ces formes beaucoup plus aisément. Voici, comment construire les trois listes données en exemple, en utilisant ce nouveau macro-caractère ‘ :

```
‘(A B C ,X F)          →   (A B C (D E) F)
‘(A B C ,@X F)         →   (A B C D E F)
‘(A B C ,(CADR X) F)   →   (A B C E F)
```

Le macro-caractère antiquote utilise deux macro-caractères supplémentaires : la virgule (,) et l'escargot (@). A l'intérieur d'une liste *antiquotée* une expression précédée d'une virgule donne sa valeur qui est ensuite insérée comme *élément* dans le résultat final. Une expression précédée des caractères , et @ donne sa valeur qui est ensuite insérée comme *segment* dans le résultat final. Toutes les autres expressions sont prises telles quelles. Les résultats d'un appel de la fonction **QUOTE** et d'un appel de la fonction **BACKQUOTE** sont donc identiques s'il n'y a pas de virgules ou d'escargots à l'intérieur de l'expression antiquotée.

Il ne reste qu'à écrire la définition de ce macro-caractère, dont voici la définition Le_LISP (très compliquée) :

```
(DMC |‘| ()
   (LET ((VIRGULE (TYPECH #\,)))
        (TYPECH #\, (TYPECH #\‘))
        (LET ((QUASI ['EVAL ['BACKQUOTE ['QUOTE (READ)]]]))
             (TYPECH #\, VIRGULE)
             QUASI)))
```

Ce que nous avons fait ici correspond à deux choses : d'abord nous avons limité la portée du macro-caractère , (virgule) : par la sauvegarde de son type de caractère à l'entrée de la définition et la détermination de son type à la valeur du type du macro-caractère quote, nous nous assurons qu'à l'intérieur de la portée du macro-caractère ‘ (antiquote) le macro-caractère , (virgule) soit défini. A la sortie de la fonction associée à l'antiquote nous restaurons, par l'instruction :

```
(TYPECH #\, VIRGULE)
```

le type de ce caractère à l'extérieur de l'antiquote. Actuellement, dans cette définition temporaire du macro-caractère ‘,’ nous avons un peu triché, puisque, en Le_LISP, **TYPECH** ne change pas la définition associée à un macro-caractère, mais juste son type. Ce que nous avons fait, c'est de garder l'ancien type de caractère dans la variable **VIRGULE**, de forcer ensuite ce caractère au type macro-caractère et, finalement, de restaurer l'ancien type de caractère. Dans tout cela nous n'avons pas touché à la fonction associée à ce caractère : nous avons juste supposé que, si le caractère virgule n'est pas un macro-caractère à l'extérieur de l'antiquote, il y restera un caractère ordinaire.

Dans la définition CommonLISP de ce même macro-caractère, nous changeons la fonction associée au caractère ‘,’ à l'intérieur de l'antiquote, et nous restaurons l'ancienne définition avant de sortir de l'antiquote. Nous faisons cela par une sauvegarde de la définition de la fonction associée à la virgule dans la variable **VIRGULE**, et nous la remettons avec un appel à la

fonction **SET-MACRO-CHARACTER**. La fonction **GET-MACRO-CHARACTER** prend en argument un caractère et ramène la définition de fonction associée à ce caractère, si elle existe, ou **NIL**, si aucune fonction n'est associée à ce caractère.

Voici la définition CommonLISP du macro-caractère antiquote :

```
(DEFUN |BQ-'| (FLUX CARACTERE)
   (LET ((VIRGULE (GET-MACRO-CHARACTER #\,)))
        (SET-MACRO-CHARACTER #\, #'VIRGULE-INTERNE)
        (LET ((QUASI ['EVAL ['BACKQUOTE ['QUOTE (READ STREAM T NIL T)]]]))
           (SET-MACRO-CHARACTER #\, VIRGULE)
           QUASI)))

(SET-MACRO-CHARACTER #\' #'|BQ-'|)
```

Après la sauvegarde et redéfinition du macro-caractère virgule, nous déléguons toute l'activité de l'antiquote à une fonction auxiliaire nommée **BACKQUOTE**. Cette fonction devra construire les appels des fonctions **LIST** et **CONS** nécessaires pour construire la bonne liste résultat. Evidemment, pour réellement construire cette liste, nous devons évaluer les appels générés; ce que nous faisons par l'appel explicite de la fonction **EVAL**.

Regardons donc la fonction **BACKQUOTE** :

```
(DEFUN BACKQUOTE (EXPR) (COND
      ((NULL EXPR) NIL)
      ((ATOM EXPR) ['QUOTE EXPR])
      ((EQ (CAR EXPR) '*UNQUOTE*) (CADR EXPR))
      ((AND (CONSP (CAR EXPR))(EQ (CAAR EXPR) '*SPLICE-UNQUOTE*))
            ['APPEND (CADAR EXPR) (BACKQUOTE (CDR EXPR))])
      ((COMBINE-EXPRS (BACKQUOTE (CAR EXPR))
                  (BACKQUOTE (CDR EXPR)) EXPR))))
```

BACKQUOTE reçoit comme argument l'expression qui suit directement le caractère ' (antiquote). Cette expression est liée à la variable **EXPR** afin de procéder à une analyse de cas. Les cinq cas suivants peuvent se présenter :

1. L'expression **EXPR** est vide : alors il n'y a rien à faire;

2. L'expression **EXPR** est un atome : alors **BACKQUOTE** génère un appel de la fonction **QUOTE**;

   exemple : si la valeur de **EXPR** est *atome*, **BACKQUOTE** génère :

   (QUOTE *atome***)**

3. L'expression débute par l'atome spécial ***UNQUOTE*** : cet atome a été généré par le caractère , (virgule); il faut alors ramener la valeur de l'argument de ,. Ceci sera fait en remplaçant l'expression **(*UNQUOTE*** *arg***)** dans l'expression lue, par **arg** dans l'expression générée.

exemple : si la valeur de **EXPR** est (***UNQUOTE*** *argument*), **BACKQUOTE** génère :

argument

4. Le premier élément de **EXPR** est une liste débutant par l'atome spécial ***SPLICE-UNQUOTE***: cet atome a été généré par les caractères ,@; il faut alors concaténer la valeur de l'argument de ***SPLICE-UNQUOTE*** à la suite de la liste. Ceci est fait en générant un appel de la fonction **APPEND** qui a comme premier argument l'argument de ***SPLICE-UNQUOTE*** et comme deuxième argument le résultat de l'appel récursif de **BACKQUOTE** avec le **CDR** de l'expression **EXPR**.

   exemple : si la valeur de **EXPR** est ((***SPLICE-UNQUOTE*** *argument*) *reste-des-arguments*), **BACKQUOTE** génère :

   (**APPEND** *argument* (**BACKQUOTE** *reste-des-arguments*))

5. Dans tous les autres cas, il faut combiner, de manière adéquate les résultats de l'analyse du **CAR** et du **CDR** de **EXPR**. Nous y distinguons quatre cas :

   a. le **CAR** et le **CDR** sont des constantes : dans ce cas il suffit de générer un appel de la fonction **QUOTE** avec l'expression entière.

   b. le **CDR** est **NIL**, alors il faut générer un appel de la fonction **LIST** avec le **CAR** comme argument.

   c. le **CDR** est une liste qui commence par un appel de **LIST** : on sait alors que cet appel a été généré par **BACKQUOTE** et on peut donc insérer le **CAR** comme nouveau premier argument de cet appel.

   d. Dans tous les autres cas, on génère un appel de la fonction **CONS** avec le **CAR** comme premier et le **CDR** comme deuxième argument, ceci restaure donc bien la liste originale.

   Voici la fonction **COMBINE-EXPRS** :

```
(DEFUN COMBINE-EXPRS (LFT RGT EXPR) (COND
     ((AND (ISCONST LFT) (ISCONST RGT)) [QUOTE EXPR])
     ((NULL RGT) ['LIST LFT])
     ((AND (CONSP RGT) (EQ (CAR RGT) 'LIST))
          (CONS 'LIST (CONS LFT (CDR RGT))))
     (['CONS LFT RGT])))
```

   qui utilise la fonction auxiliaire **ISCONST** :

```
(DEFUN ISCONST (X)
     (OR (NULL X) (EQ X T)(NUMBERP X)(CHARACTERP X)
          (AND (CONSP X)(EQ (CAR X) 'QUOTE))))
```

Finalement, il ne reste qu'à définir le macro-caractère ,. Ce macro-caractère ne devant être

valide qu'à l'intérieur du **BACKQUOTE**, il faut bien faire attention que la définition même du macro-caractère ne modifie pas *définitivement* son type. C'est pourquoi, en Le_LISP, nous mettons la **DMC** à l'intérieur d'un **LET** qui se contente de sauvegarder et restaurer le type du caractère , avant et après sa définition en tant que macro-caractère. Voici donc ce bout de programme :

```
(LET ((VIRGULE (TYPECH #\,)))
     (DMC |,| ()
        (LET ((X (PEEKCH)))
           (IF (NULL (EQUAL X #\@)) ['*UNQUOTE* (READ)]
               (READCH)
               ['*SPLICE-UNQUOTE* (READ)])))
     (TYPECH #\, VIRGULE))
```

Regardez bien l'utilisation de la fonction **PEEKCH** !

Et ci-dessous la version CommonLISP correspondante. Rappelez-vous qu'ici nous ne définissons que la fonction associée, l'association effective entre cette fonction et le caractère virgule sera faite à l'intérieur de l'antiquote.

```
(DEFUN VIRGULE-INTERNE (FLUX CARACTERE)
   (LET ((X (PEEK-CHAR FLUX T NIL T))) (COND
          ((NULL (EQUAL X #\@))
                ['*UNQUOTE* (READ FLUX T NIL T)])
          (T (READ-CHAR FLUX T NIL T)
                ['*SPLICE-UNQUOTE* (READ FLUX T NIL T)])))))
```

Après tout ce qui précède, examinons donc quelques exemples de ce nouveau macro-caractère. Voici un premier exemple : si vous donnez à LISP l'expression :

```
'(A B C)
```

puisque tous les éléments de cette liste sont des constantes, la fonction **BACKQUOTE** génère l'appel :

```
(QUOTE (A B C))
```

et son évaluation donne donc la liste :

```
(A B C)
```

Reprenons les exemples ci-dessus et supposons que la variable **X** soit liée à la liste **(D E)** et la variable **Y** à l'atome **F**.

Si nous livrons à LISP l'expression :

```
'(A B C ,X Y)
```

le lecteur LISP traduira ceci en :

```
(EVAL (BACKQUOTE (QUOTE (A B C (*UNQUOTE* X) Y))))
```

ce qui sera, après évaluation de l'appel de **BACKQUOTE** :

```
(EVAL '(CONS (QUOTE A)
             (CONS (QUOTE B)
                   (CONS (QUOTE C)
                         (CONS X (QUOTE (Y)))))))
```

dont l'évaluation nous livre finalement :

```
(A B C (D E) Y)
```

Pour notre deuxième exemple, l'expression :

```
'(A B C ,@X ,Y)
```

sera traduite par le lecteur LISP en :

```
(EVAL (BACKQUOTE
        (QUOTE (A B C
                  (*SPLICE-UNQUOTE* X)
                  (*UNQUOTE* Y)))))
```

ce qui se transformera, après l'évaluation de l'argument de **EVAL** en :

```
(EVAL '(CONS (QUOTE A)
             (CONS (QUOTE B)
                   (CONS (QUOTE C)
                         (APPEND X (LIST Y))))))
```

qui produira la liste :

```
(A B C D E F)
```

Laissons pour le moment les macro-caractères regardons brièvement comment changer de fichier d'entrée/sortie.

13.4 COMMENT CHANGER DE FICHIERS D'ENTREE/SORTIE

Jusqu'à maintenant nous ne pouvons entrer des fonctions LISP qu'à travers le terminal, et nous ne pouvons écrire les résultats d'un calcul que vers ce même terminal. On désire souvent préparer ses programmes sous éditeur pour seulement ensuite les charger en LISP, ou sauvegarder ses résultats dans des fichiers afin de pouvoir les réutiliser par la suite. Ce paragraphe vous montrera comment réaliser ce genre d'opérations.

Tout d'abord, LISP met à votre disposition une fonction de *lecture* de fichiers. Cette fonction

s'appelle **LOAD**. Elle est fort utile pour *charger* un programme qui se trouve dans un fichier. Voici la définition de **LOAD** :

(LOAD *nom-de-fichier***)** → **T**
en plus : lit le fichier de nom *nom-de-fichier* en entier, en évaluant chacune des expressions qu'il contient.

Ainsi, si vous travaillez sous un système UNIX, et si le programme du paragraphe précédent (la définition de l'*antiquote*) se trouve dans le fichier **antiquote.lisp** dans le catalogue **my-lisp**, la commande

(LOAD "my-lisp/antiquote.lisp")

va lire ce fichier, définition par définition.

L'argument *nom-de-fichier* est une chaîne de caractères correspondant à la manière dont les fichiers sont spécifiés dans votre système d'exploitation. Par exemple, si vous voulez charger, sous UNIX, le fichier /usr/hw/lisp/exemple.lisp vous spécifiez la chaîne **"/usr/hw/lisp/exemple.lisp"**. Sous Multics, la spécification du fichier correspondant serait quelque chose comme **">udd>hw>lisp>exemple.lisp"**, ou, sous quelques systèmes étranges ça pourrait être **"\hw\lisp\exemple.lisp"**.

La fonction **LOAD** est probablement la fonction d'entrée/sortie manipulant des fichiers la plus utilisée : tout programme sauvegardé dans des fichiers doit être chargé avant de pouvoir être exécuté.

Naturellement, il existe encore d'autres raisons pour rediriger des entrées/sorties : on veut pouvoir écrire des résultats de claculs, ou des résultats intermédiaires dans des fichiers, ou on veut pouvoir lire des données dans un fichier. CommonLISP traite toutes ces possibilités en donnant un argument supplémentaire aux fonctions d'entrée/sortie. Cet argument supplémentaire doit être un *flux*. Un flux est un type de donnée spécial utilisé pour la manipulation des entrées/sorties, et sert comme une source (pour des opérations d'entrées) ou comme un réservoir (pour des opérations de sorties). Normalement, des flux sont connectés à des fichiers, terminaux ou autres périphériques. Ils forment l'interface entre LISP et le système d'exploitation. Ainsi, afin de pouvoir faire des opérations de lecture ou d'écriture sur un fichier, on doit d'abord positionner un flux qui connecte à ce fichier. Cette opération de positionnement se réalise avec la fonction **OPEN**. De même que **LOAD**, la fonction **OPEN** attend en argument un *nom de fichier*. De plus, on doit spécifier si l'on veut lire de ce fichier, si l'on veut écrire vers ce fichier ou si l'on veut juste savoir si ce fichier existe. Le résultat d'un appel à la fonction **OPEN** est le positionnement d'un flux interfaçant LISP avec ce fichier. C'est ce flux qui est retourné.

Prenons un exemple : afin d'ouvrir le fichier test.lisp du catalogue courant pour des opérations de sortie, nous devons appeler **OPEN** comme suit :

(OPEN "test.lisp" :DIRECTION :OUTPUT)

Ceci ramène quelque chose comme :

#<output stream "test.lisp">

indiquant qu'un flux vers le fichier spécifié a été ouvert en écriture. En fait, ce que la machine imprime peut varier d'une machine à une autre (et sera en général beaucoup moins lisible). Mais dans tous les cas un *flux* sera ramené.

Pour utiliser ce flux dans une opération d'écriture, nous devons le garder quelque part. Il est donc mieux d'écrire quelque chose comme :

(LET ((MON-FLUX-DE-SORTIE (OPEN "test.lisp" :DIRECTION :OUTPUT))) ...

et, par la suite, nous pouvons référer à ce flux de sortie à travers la variable **MON-FLUX-DE-SORTIE**.

Une fois que nous avons fini d'écrire dans un fichier, nous devons *fermer* le flux qui s'y connecte. Ceci se fait grâce à la fonction **CLOSE**, qui attend un flux en argument et dont l'effet est de fermer la connexion existante entre LISP et le fichier, donc : **CLOSE** ferme le flux donné en argument. Ici dans notre exemple, une fois que nous avons fini d'écrire dans le fichier **test.lisp**, nous fermons la connexion par :

(CLOSE MON-FLUX-DE-SORTIE)

On peut également créer un flux pour des opérations d'entrée. Ainsi, si vous avez l'intention de lire des données qui se trouvent dans le fichier data.lisp (dans le catalogue courant), vous devez d'abord appeler la fonction **OPEN** comme suit :

(OPEN "data.lisp" :DIRECTION :INPUT)

qui retourne un flux, tel que par exemple :

#<input stream "data.lisp">[8]

Ce flux est utilisable ensuite pour des opérations de lecture. On peut également ouvrir un fichier à la fois pour l'écriture et la lecture, c'est-à-dire : on peut ouvrir un flux bidirectionnel avec la *direction* **:IO**. Ainsi l'appel :

(OPEN "mon-fichier" :DIRECTION :IO)

retourne un flux qui peut être utilisé à la fois pour des opérations d'écriture et de lecture. Il existe une direction de plus pendant l'ouverture d'un flux, c'est la direction **:PROBE**. Si l'on donne cette direction comme argument à un appel de **OPEN**, CommonLISP ne crée pas

8. La même remarque que pour l'appel précédent de cette fonction s'impose : 1) la manière selon laquelle le flux sera imprimé varie d'une implémentation de CommonLISP à une autre et 2) il faut bien garder ce flux dans une variable si l'on veut l'utiliser ensuite et si l'on veut le fermer proprement à la fin des opérations de lecture.

réellement un flux, mais ramène **NIL** si le fichier donné en argument n'existe pas, et quelque chose différent de **NIL** s'il existe. Ainsi, l'appel :

```
(OPEN "mon-fichier-test" :DIRECTION :PROBE)
```

retourne #<**probe stream "mon-fichier-test**"> si le fichier **mon-fichier-test** existe et sinon cet appel retourne **NIL**.

Remarquons que l'ouverture d'un fichier inexistant dans les directions **:OUTPUT** et **:IO** crée un fichier du nom donné en argument à **OPEN**. Si l'on essaye d'ouvrir un flux vers un fichier inexistant dans la direction **:INPUT**, LISP donne un message d'erreur.

Jusqu'ici nous savons comment créer des flux pour la lecture, l'écriture ou les deux. Pour écrire quelque chose dans un flux de sortie nous utilisons la fonction **PRINT** (ou toute autre fonction de sortie) avec le flux comme deuxième argument. Voici un exemple :

```
? (DEFUN ALL-CDR-REVERSE (L FLUX)
        (AND L (PRINT (REVERSE L) FLUX)
              (ALL-CDR-REVERSE (CDR L) FLUX)))
= ALL-CDR-REVERSE
```

Et voici un appel de cette fonction :

```
? (LET ((FLUX (OPEN "mon-fichier.lisp" :DIRECTION :OUTPUT)))
        (ALL-CDR-REVERSE '(A B C) FLUX)
        (CLOSE FLUX))
= T
```

Dans cet exemple, nous avons d'abord défini une fonction qui imprime l'inverse des CDR's successifs d'une liste. Remarquons que l'appel de la fonction **PRINT** possède deux arguments : une liste à imprimer et un flux pour imprimer.

Le **LET** fait ensuite trois choses :

1. D'abord il ouvre un flux de sortie vers le fichier **mon-fichier.lisp**. Ce flux est gardé dans la variable **FLUX**.

2. Ensuite, il appelle la fonction **ALL-CDR-REVERSE** avec la liste **(A B C)** et le flux **FLUX**.

3. Finalement, il ferme le flux **FLUX**, ce qui interrompt la connexion vers le fichier **mon-fichier.lisp**.

La dernière ligne dans l'exemple est la valeur retournée par l'appel du **LET**, l'atome **T**, qui, à son tour, est la valeur retournée par la fonction **CLOSE**.

Aucune sortie de ces appels de **PRINT** n'apparaît sur le terminal, car toutes les sorties étaient dirigées vers le fichier **mon-fichier.lisp**. Essayons de lire le contenu du fichier que nous venons d'écrire, pour voir ce qu'il contient :

```
? (LET ((FLUX (OPEN "mon-fichier.lisp" :DIRECTION :INPUT)))
        (PRINT (READ FLUX))
        (PRINT (READ FLUX))
        (PRINT (READ FLUX))
        (CLOSE FLUX))
(C B A)
(C B)
(C)
= T
```

Ce que nous avons fait ici, c'est :

1. nous avons ouvert un flux de lecture vers le fichier **mon-fichier.lisp** que nous avons gardé dans la variable **FLUX** ;

2. ensuite nous avons imprimé ce que nous lisions sur ce flux. Remarquez que les appels de la fonction **PRINT** ici n'ont qu'un seul argument. Si ceci est le cas, ou si le deuxième argument de **PRINT** est **NIL**, CommonLISP redirige les sorties vers le terminal. **READ** a un argument : le flux que nous voulons lire. Finalement,

3. nous avons fermé le flux d'entrée ouvert précédemment. C'est encore la valeur **T** ramenée par l'appel de la fonction **CLOSE**, qui est ramenée en valeur par le **LET**.

Toutes les fonctions d'entrée/sortie acceptent un flux comme argument supplémentaire. Si cet argument est absent, ou s'il est égal à NIL, l'entrée ou la sortie sont redirigées vers le terminal, plus précisement : elles sont redirigées vers le flux indiqué dans les variables CommonLISP globales ***STANDARD-INPUT*** et ***STANDARD-OUTPUT***. Bien entendu, si cet argument est donné, les entrées/sorties sont redirigées vers le flux donné en argument.

Cette règle explique également l'occurrence du premier argument **T** dans tous les exemples d'appels de la fonction de formatage **FORMAT**. Le **T** comme premier argument indique que l'impression doit avoir lieu vers la sortie standard (***STANDARD-OUTPUT***). Si l'on veut imprimer vers un autre flux, il suffit de donner ce flux à la place de l'argument **T**.

Une dernière remarque sur la fonction **READ** : dans la section précédente nous avons vu des appels de cette fonction avec quatre arguments, comme dans :

```
(READ FLUX T NIL T)
```

Le premier argument est le flux duquel on veut lire. Nous savons également que le dernier argument indique à **READ** si c'est un appel récursif ou non, c'est le *recursive-p* argument pour des appels à partir de macro-caractères. Il reste deux arguments supplémentaires, le deuxième et le troisième. Ces deux arguments indiquent à la fonction **READ** que faire si elle rencontre la fin d'un fichier. Si le deuxième argument est égal à **T**, **READ** interrompt le calcul et donne un message d'erreur si l'on veut lire plus loin que la fin du fichier. Dans ce cas, le troisième argument n'a pas de signification. Par contre, si le deuxième argument est **NIL**, le troisième indique alors quelle valeur **READ** devrait retourner si on lui demande de lire plus loin que la fin d'un fichier. Par exemple, l'appel suivant :

```
(READ MON-FLUX-D-ENTREE NIL '|The End|)
```

lit normalement une expression s'il y en a une, sinon, si **READ** se trouve à la fin d'un fichier, cet appel retourne l'atome **The End**. Ceci est très utile si on veut lire toutes les expressions d'un fichier et qu'on ne sait pas a priori combien d'expressions s'y trouvent.

Prenons un exemple et supposons que nous ayons un fichier appelé **valeurs** qui contient la suite d'atomes suivante :

```
1
1
2
3
5
8
13
21
34
55
```

et construisons une fonction qui calcule la somme de tous les nombres d'un fichier. Voici cette fonction :

```
(DEFUN SOMME-DE-NOMBRES (FICHIER)
  (LABELS ((AUX-SUM (NOMBRE FLUX) (COND
             ((EQUAL NOMBRE '|The End|) 0)
             ((NUMBERP NOMBRE)
                (+ NOMBRE (AUX-SUM (READ FLUX NIL '|The End|))))
             (T (AUX-SUM (READ FLUX NIL '|The End|))))))

    (LET ((MON-FLUX (OPEN FICHIER :DIRECTION :INPUT)))
      (LET ((RESULTAT (AUX-SUM 0 MON-FLUX)))
        (CLOSE MON-FLUX)
        RESULTAT))))
```

Notons qu'ici nous avons utilisé la possibilité du **READ** de CommonLISP de ramener une valeur donnée quand il rencontre une fin de fichier. On utilise cette valeur comme test d'arrêt de la récursion.

Pour calculer la somme des nombres contenus dans le fichier **valeurs**, il suffit d'appeler cette fonction avec le fichier en argument :

```
(SOMME-DE-NOMBRES "valeurs")
```

Notes sur les fonctions d'entrées/sorties en Le_LISP

Pour terminer ce chapitre, examinons brièvement les particularités des fonctions d'entrées/sorties de Le_LISP.

La fonction CommonLISP **LOAD** s'appelle **LOADFILE** en Le_LISP. **LOAD** et **LOADFILE** prennent les mêmes arguments et ont un comportement identique.

Au lieu d'avoir une masse de mots-clefs associée à la fonction **OPEN** comme le fait CommonLISP, Le_LISP propose un ensemble de fonctions différentes pour chacun des rôles de la fonction CommonLISP **OPEN**. Pour ouvrir un fichier en lecture, Le_LISP offre la fonction **OPENI** (pour *open* input). Cette fonction prend un argument, le fichier dans lequel on veut lire, et retourne un flux d'entrée connectant vers ce fichier. Pour ouvrir un fichier en écriture, Le_LISP utilise la fonction symétrique **OPENO** (pour *open* output) – avec un comportement similaire à **OPENI**. Finalement, pour tester si un fichier existe, le programmeur Le_LISP utilise la fonction **PROBEFILE**, qui prend un nom en argument et retourne **T** si un fichier de ce nom existe, **NIL** s'il n'existe pas. Comme en CommonLISP, un flux est fermé avec la fonction **CLOSE**.

Des différences plus profondes existent si l'on veut rediriger ses entrées ou/et sorties. Le_LISP, comme CommonLISP, peut avoir plusieurs flux ouverts, mais toutes les opérations de sorties passent par un **canal**, et c'est ce canal qui est connecté vers un flux. Ainsi avant de pouvoir appeler la fonction **PRINT** (ou d'autres fonctions de sortie), on doit associer le canal de sortie au flux désiré. Cette association se fait avec la fonction **OUTCHAN**. Par exemple, les cinq instructions ci-dessous vont successivement

- ouvrir le fichier **test.ll**,
- associer le canal de sortie avec le flux connecté au fichier **test.ll**,
- écrire le message **the time has come, the walross said, to talk of many things** dans le fichier **test.ll**,
- re-associer le canal de sortie avec le terminal,
- et finalement fermer le flux connecté au fichier **test.ll**.

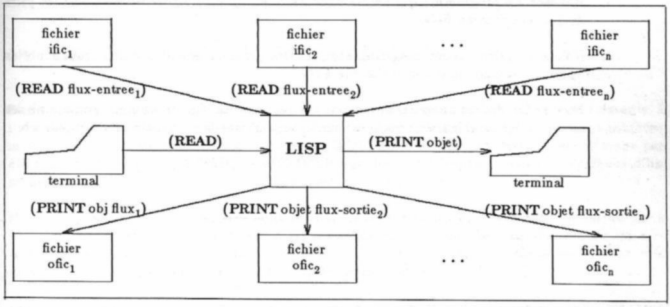
Voici donc le code Le_LISP faisant tout cela :

```
(LET ((MON-FLUX (OPENO "test.ll")))
     (OUTCHAN MON-FLUX)
     (PRINT "the time has come, the walross said, to talk of many things")
     (OUTCHAN NIL)
     (CLOSE MON-FLUX))
```

La fonction symétrique à **OUTCHAN** est appelée **INCHAN** ; elle associe le canal d'entrée avec un flux.

Cette manière de rediriger les entrées/sorties a quelques avantages si l'on doit écrire des grandes masses de données dans un même fichier : la fonction **PRINT** de Le_LISP est plus performante que celle de CommonLISP car elle est libérée de la tâche de déterminer vers où il faut envoyer les sorties. La manière de CommonLISP a des avantages si l'on doit souvent changer les directions des entrées ou/et des sorties.

Finalement voici graphiquement, les possibilités d'échanges entre LISP et le monde extérieur (les flèches correspondent aux flux connectant LISP à des fichiers ou à des périphériques) :



13.5 EXERCICES

1. Au lieu d'écrire **(LOAD** *nom-de-fichier*) chaque fois que vous voulez charger un fichier, écrivez un macro-caractère '^' qui vous permette de charger des fichiers simplement par la commande

   ^*nom-de-fichier*

2. Supposant que votre LISP ne possède pas les fonctions >= et <=, définissez des macro-caractères >= et <= correspondants.

   Bien entendu, ceci implique que > et < deviennent également des macro-caractères, et que les appels ne seront plus de la forme :

   (< *premier-argument deuxième-argument*)

   ou :

   (<= *premier-argument deuxième-argument*)

   mais de la forme :

   < *premier-argument deuxième-argument*

   ou :

   <= *premier-argument deuxième-argument*

   c'est-à-dire : sans les parenthèses !

3.

a. Réalisez un programme qui écrive dans un fichier nommé **pair.nb** les nombres pairs de 1 à *n* et l'atome **FIN**.

b. Réalisez également un programme qui écrive dans un fichier nommé **impair.nb** les nombres impairs de 0 à *m* et l'atome **FIN**.

c. Ecrivez un troisième programme qui écrive dans un fichier nommé **somme.nb** les sommes des doublets de nombres pairs et impairs des fichiers **pair.nb** et **impair.nb**.

14. LE FILTRAGE (PREMIERE PARTIE)

Dans la programmation en LISP il est très souvent nécessaire de chercher des éléments à l'intérieur d'une liste ou de comparer deux listes. Les fonctions que nous connaissons actuellement pour traiter ce type de problème, sont **EQUAL**, qui compare deux listes élément par élément, et les fonctions **MEMQ** et **MEMBER** qui cherchent respectivement l'occurrence d'un atome ou d'un élément quelconque à l'intérieur d'une liste donnée.

Bien que fort utiles, ces fonctions sont relativement restreintes et – surtout – n'aident pas à comparer deux listes *approximativement* : par exemple savoir si une liste comporte dans un certain ordre quelques éléments particuliers, ou savoir si une liste est composée de seulement trois éléments avec comme dernier élément un atome particulier.

Bien évidemment, nous pouvons toujours écrire un programme particulier pour résoudre indépendamment chaque problème de comparaison. Par exemple, pour savoir si une liste **L** comporte les trois éléments **DO**, **RE** et **MI** dans cet ordre, nous pouvons définir la fonction **DOREMI** que voici :

```
(DEFUN DOREMI (L)
    (LET ((X (MEMQ 'DO L)))
        (IF (NULL X) ()
            (AND (EQ (CADR X) 'RE) (EQ (CADDR X) 'MI)) T))))
```

ou, pour savoir si une liste n'a que trois éléments avec comme dernier élément l'atome **MI**, nous pouvons construire la fonction **BIDON** que voici :

```
(DEFUN BIDON (L)
   (AND (= (LENGTH L) 3) (EQ (CAR (REVERSE L)) 'MI)))
```

mais ce type de fonctions risque de se multiplier très vite, avec une fonction particulière pour chaque cas particulier qui se présente.

Le *filtrage* est une technique qui permet de comparer une liste à un "*canevas*" de liste, où le *canevas* est une manière de décrire la *structure* d'une liste. Pour reprendre les deux exemples ci-dessus, les canevas correspondants pourraient être :

```
(? DO RE MI ?)
```

et

```
(? $ MI)
```

Dans ces canevas, le signe '**?**' indique une suite d'éléments quelconques et le signe '**$**' indique un unique élément quelconque.

Ainsi, les canevas sont des descriptions, visuellement compréhensibles, de structures de listes. En termes plus techniques un tel canevas s'appelle un *filtre* ou un *pattern* et le processus de comparaison d'un *filtre* avec une liste s'appelle le *filtrage* ou le *pattern-matching*. Dans ce chapitre nous allons voir comment construire un tel programme de *filtrage*.

14.1 PREMIERE VERSION D'UNE FONCTION DE FILTRAGE

Commençons par nous restreindre à des filtres composés exclusivement de *constantes* (c'est-à-dire des atomes quelconques) et des signes '$'. Voici quelques exemples de ce que notre fonction de comparaison **MATCH** doit pouvoir faire :

```
(MATCH '(DO RE MI) '(DO RE MI))          →  T
(MATCH '(DO RE MI) '(DO RE DO))          →  NIL
(MATCH '(DO $ MI) '(DO RE MI))           →  T
(MATCH '(DO $ MI) '(DO DO MI))           →  T
(MATCH '(DO $ MI) '(DO (DO RE MI) MI))   →  T
(MATCH '(DO $ MI) '(DO RE))              →  NIL
(MATCH '(DO $ MI) '(DO FA RE))           →  NIL
(MATCH '($ $ $) '(1 2 3))                →  T
```

Le premier filtre, **(DO RE MI)**, ne compare que des listes identiques puisqu'aucun des signes particuliers **$** ou **?** n'est utilisé.

Le deuxième filtre, **(DO $ MI)**, laisse passer toutes les listes à trois éléments qui débutent par l'atome **DO** et se terminent par l'atome **MI**. Le deuxième élément peut être n'importe quel atome ou n'importe quelle liste. Le signe **$** indique juste : à l'endroit où il se trouve il doit y avoir *un* élément.

Le troisième, **($ $ $)**, filtre toute liste de trois éléments exactement.

Le filtrage réduit au seul signe particulier '$' se comporte donc presque comme la fonction **EQUAL** : si aucune occurrence de '$' ne se trouve dans le filtre, le filtrage *est identique* au test d'égalité. S'il y a occurrence du signe '$' la comparaison peut juste *sauter* l'élément correspondant dans la liste donnée.

Avant d'écrire notre première version de la fonction **MATCH**, rappelons-nous la définition de la fonction **EQUAL** (cf §6.3) :

```
(DEFUN EQUAL (ARG1 ARG2) (COND
      ((ATOM ARG1) (EQ ARG1 ARG2))
      ((ATOM ARG2) NIL)
      ((EQUAL (CAR ARG1) (CAR ARG2)) (EQUAL (CDR ARG1) (CDR ARG2)))))
```

Maintenant, l'écriture de **MATCH** ne pose plus de problèmes :

```
(DEFUN MATCH (FILTRE DONNEE) (COND
        ((ATOM FILTRE) (EQ FILTRE DONNEE))
        ((ATOM DONNEE) NIL)
        ((EQ (CAR FILTRE) '$)
             (MATCH (CDR FILTRE) (CDR DONNEE)))
        ((MATCH (CAR FILTRE) (CAR DONNEE))
             (MATCH (CDR FILTRE) (CDR DONNEE)))))
```

Cette fonction permet de filtrer avec le signe '$', donc d'extraire des données ayant le même nombre d'éléments que le filtre.

Si nous voulons également permettre le signe '?', qui remplace toute une *séquence* d'éléments, nous devons introduire une clause supplémentaire :

```
((EQ (CAR FILTRE) '?) . . .)
```

Afin de voir par quoi remplacer les '. . .', regardons d'abord un exemple. Le filtre :

```
(? DO ?)
```

récupère toute liste qui contient au moins une occurrence de l'atome **DO**. Donc, ce filtre marche pour la liste **(DO)**, ainsi que pour toutes les listes où cet atome est précédé d'un nombre indéfini d'éléments, comme, par exemple, les listes :

```
(1 DO)
(1 2 DO)
(1 1 2 DO)
(A B C D E F G DO)
etc
```

ainsi que pour les listes où cet atome est suivi d'un nombre quelconque d'éléments quelconques, comme, par exemple :

```
(DO 1)
(DO 1 2 3)
. . .
(1 DO 1)
(1 2 DO 1 2 3)
etc
```

Le signe '?' peut donc ne correspondre à rien (c'est le cas si la liste donnée est **(DO)**) ou à un nombre quelconque d'éléments.

La manière la plus simple de réaliser cette forme de filtrage est d'utiliser récursivement la fonction **MATCH** elle-même. Ainsi, si nous rencontrons le signe '?' dans le filtre, nous allons d'abord essayer de filtrer le reste du filtre avec la donnée entière (en supposant alors qu'au signe '?' correspond la séquence vide) et si ça ne marche pas, nous allons essayer de filtrer le reste du filtre avec les **CDR**s successifs de la donnée – jusqu'à ce que nous ayons trouvé une donnée qui

fonctionne. Voici alors le nouveau code pour notre première version de MATCH :

```
(DEFUN MATCH (FILTRE DONNEE) (COND
        ((ATOM FILTRE) (EQ FILTRE DONNEE))
        ((EQ (CAR FILTRE) '?) (COND
              ((NULL (CDR FILTRE)) T)
              ((MATCH (CDR FILTRE) DONNEE))
              (DONNEE (MATCH FILTRE (CDR DONNEE)))))
        ((ATOM DONNEE) NIL)
        ((EQ (CAR FILTRE) '$)
              (MATCH (CDR FILTRE) (CDR DONNEE)))
        ((MATCH (CAR FILTRE) (CAR DONNEE))
              (MATCH (CDR FILTRE) (CDR DONNEE)))))
```

Voici quelques exemples d'appels ainsi que leurs résultats :

```
(MATCH '(A B C) '(A B C))       →  T
(MATCH '(A $ C) '(A B C))       →  T
(MATCH '(A $ C) '(A 1 C))       →  T
(MATCH '(A $ C) '(A 1 2 3 C))   →  NIL
(MATCH '(A ? C) '(A 1 2 3 C))   →  T
(MATCH '(A ? C) '(A C))         →  T
```

Regardez bien la différence dans l'effet des deux signes '$' et '?' : l'un filtre *un* élément *unique* et l'autre filtre une *séquence* (éventuellement vide) d'éléments.

Faire des tentatives d'appels d'une fonction est très courant dans la programmation en intelligence artificielle. Regardons donc une trace de l'exécution de l'appel :

```
(MATCH '(A ? C) '(A 1 2 3 C))
```

```
---> (MATCH (A ? C) (A 1 2 3 C))      ; l'appel initial
  ---> (MATCH A A)                    ; le premier appel récursif
  <--- MATCH T                        ; ça marche
  ---> (MATCH (? C) (1 2 3 C))        ; le deuxième appel récursif
    ---> (MATCH (C) (1 2 3 C))        ; essayons la séquence vide
      ---> (MATCH C 1)                ; est-ce que ça marche ?
      <--- MATCH NIL                  ; non
    <--- MATCH NIL                    ; c'est pas la séquence vide
    ---> (MATCH (? C) (2 3 C))        ; essayons la séquence d'un élément
      ---> (MATCH (C) (2 3 C))        ; et réptons l'essai
        ---> (MATCH C 2)              ; ça s'annonce mal
        <--- MATCH NIL                ; évidemment !
      <--- MATCH NIL                  ; c'est pas la séquence à 1 élément
      ---> (MATCH (? C) (3 C))        ; essayons à 2 éléments
        ---> (MATCH (C) (3 C))
          ---> (MATCH C 3)
          <--- MATCH NIL              ; ça ne marche pas
        <--- MATCH NIL                ; non plus
        ---> (MATCH (? C) (C))        ; et avec 3 éléments ?
          ---> (MATCH (C) (C))        ; ça a l'air bien
            ---> (MATCH C C)          ; ici ça marche
            <--- MATCH T              ; réellement
            ---> (MATCH NIL NIL)      ; on est à la fin
          <------ MATCH T             ; et ça marchait réellement
<---------- MATCH T                   ; enfin !
= T                                   ; voici le résultat final
```

Nous reviendrons au filtrage pour le généraliser et pour le rendre plus puissant. Pour cela, nous avons absolument besoin de revoir notre compréhension des listes. C'est ce que nous allons faire au chapitre suivant.

14.2 EXERCICES

1. Lesquels des appels récursifs de **MATCH** sont récursifs terminaux et lesquels ne le sont pas ?

2. Comment modifier le programme **MATCH** ci-dessus pour permettre l'utilisation de filtres donnant le choix entre plusieurs éléments possibles ? Par exemple, si nous voulons trouver à l'intérieur d'un programme tous les appels des fonctions **CAR** et **CDR** nous pouvons désigner ces appels par les deux filtres :

```
(CAR ?)
(CDR ?)
```

   que l'on pourrait contracter en un seul filtre, par exemple :

```
((%CAR CDR) ?)
```

   c'est-à-dire : chaque fois que notre fonction rencontre, dans le filtre, une sous-liste

commençant par le signe '%, le reste de la liste est un ensemble d'éléments dont *un* doit se trouver à l'endroit correspondant dans la donnée.

Voici quelques exemples de cette nouvelle fonction **MATCH** avec ce nouveau type de filtre :

(MATCH '(A (%B 1) C) *quelquechose*)

ne filtre que les deux listes (A B C) et (A 1 C), et le filtre

(MATCH '((%A B) (%A B)) *quelquechose*)

ne filtre que les quatre listes : (A A) (A B) (B A) (B B).

15. RETOUR VERS LES LISTES

15.1 L'ORGANISATION DE LA MEMOIRE LISP

Nous savons qu'en LISP nous avons toute sorte d'objets : les listes, les nombres, les chaînes de caractères, les atomes. Ils se trouvent dans des zones spéciales de la mémoire de l'ordinateur.

La figure de la page suivante montre graphiquement l'utilisation de l'espace mémoire de l'ordinateur sous LISP :[1] Les numéros à gauche du tableau représentent des adresses. Ainsi, dans notre figure, les premiers α mots de la mémoire d'ordinateurs sont utilisés par le système d'exploitation de l'ordinateur. Naturellement, cette zone peut se situer ailleurs, son emplacement précis dépend de l'ordinateur particulier sur lequel votre LISP est implémenté. Ce qui compte, c'est que *quelque part* une zone (ou des zones) existe qui ne peut pas être utilisée par LISP.

Regardons donc chacune des zones de LISP : la première, allant de l'adresse α à l'adresse β-1, est occupée par l'interprète LISP même. C'est ici qu'on trouvera les programmes d'entrée/sortie, la boucle *toplevel* que nous avons vue au chapitre 13, etc. Cette zone n'est pas accessible pour l'utilisateur : c'est la zone système de votre machine LISP, de même que la première zone est la zone système de votre ordinateur.

Ensuite, il y a des zones pour chacun des objets existants en LISP : une zone pour stocker les nombres, une pour les chaînes de caractères, une pour les atomes et une pour les listes. Les listes ne se trouvent donc *que* dans la zone liste, les nombres *que* dans la zone nombre, etc.

La dernière zone de votre machine LISP est réservé pour la pile. C'est une zone de travail pour l'interprète : on y stocke, par exemple, des résultats intermédiaires. Nous verrons l'utilité de cette zone plus loin. Concentrons-nous sur la zone liste, puisque c'est elle qui nous intéresse pour l'instant.

1. L'organisation que nous proposons ici n'est pas la seule possible. D'autres systèmes LISP le font différemment. Nous proposons cette organisation puisqu'elle est conceptuellement la plus simple et expose néanmoins toutes les caractéristiques les plus importantes.

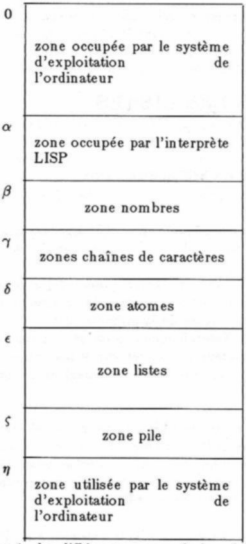


Figure 1. les différentes zones de la mémoire

15.2 LA ZONE LISTE

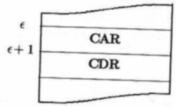
Les listes ont été inventées pour pouvoir utiliser la mémoire sans contrainte de taille. Les idées de base sont :

1. d'organiser cette mémoire de manière non continue : une liste ne doit pas obligatoirement occuper des mots mémoires contigus. Pour cela, les listes sont réalisées de telle façon qu'il soit toujours possible à partir d'*un* élément de la liste de trouver le suivant.

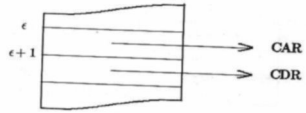
2. de disposer d'un algorithme de *récupération de mémoire.* Cet algorithme, qui s'appelle en anglais le *garbage collector,* doit pouvoir trouver des listes (ou des parties de listes) qui ne sont plus utilisées et il doit les inclure à nouveau dans l'espace utilisable.

Tout cela à l'air bien abstrait ! Examinons donc l'organisation de cette zone plus en détail.

Une liste se constitue d'un **CAR** et d'un **CDR**, d'un premier élément et d'un reste. A l'intérieur de la machine, donc à l'intérieur de la zone liste, ces deux parties se retrouvent en permanence, et le **CAR** et le **CDR** d'une liste occupent deux mots adjacents de cette zone :

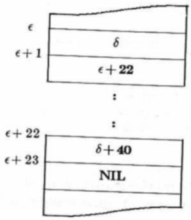


Chacune des boîtes représente un mot mémoire dans la zone liste. Les étiquettes **CAR** et **CDR** à l'intérieur des boîtes sont des *pointeurs* vers les objets LISP constituant le **CAR** et le **CDR** de cette liste. Graphiquement :



Le **CAR** ou le **CDR** d'une liste peuvent être de n'importe quel type : atome ou liste. De fait, les pointeurs CAR et CDR peuvent pointer dans n'importe quelle *zone* des objets LISP. Ce sont donc, en réalité, des adresses de la mémoire.

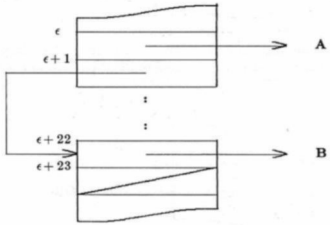
Ainsi, disposant d'une liste **(A B)**, d'un atome **A** se trouvant en mémoire à l'adresse δ, et d'un atome **B** se trouvant à l'adresse $\delta+40$, cette liste peut être implémentée en mémoire de la manière suivante :



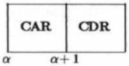
Les listes sont donc des couples (on dit : des *doublets*) de mots constituées d'un pointeur vers l'emplacement de la valeur du **CAR** et d'un pointeur vers l'emplacement de la valeur du **CDR**,

c'est-à-dire : l'*adresse* du mot-mémoire où se trouve la valeur correspondant. La fin d'une liste est indiquée par un pointeur vers l'atome *très* particulier **NIL**.

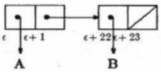
Afin de ne pas surcharger la représentation graphique par trop d'adresses, nous allons représenter les pointeurs vers des listes par des flèches vers les doublets correspondants, la fin d'une liste par une diagonale dans le **CDR** du dernier doublet et au lieu de marquer les adresses des atomes, nous allons juste indiquer des pointeurs vers les noms des atomes. Ce qui donne pour la même liste **(A B)** :



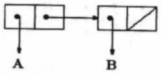
Bien évidemment, nous ne pouvons pas constamment prendre en considération les adresses réelles, c'est pourquoi nous utiliserons une représentation graphique des listes qui ne prend en considération que les *doublets* directement utilisés dans une liste. Une liste sera donc représentée comme une suite de *doublets* composée des **CAR**s et **CDR**s de cette liste :



La liste **(A B)** de l'exemple ci-dessus sera donc représentée comme :

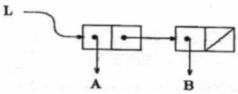


Normalement, quand nous allons donner une représentation graphique d'une liste, nous allons ignorer les adresses d'implémentation réelle et dessiner, par exemple, la liste **(A B)**, comme :



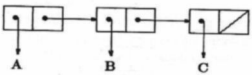
Mais, même si nous prenons cette représentation (plutôt) abstraite, il ne faut jamais oublier qu'en réalité toute la représentation interne se fait en terme d'adresses réelles !

Si cette liste **(A B)** est la valeur de la variable **L**, au lieu de représenter graphiquement l'atome L avec comme C-valeur (cf §8.1) l'adresse de cette liste, nous allons représenter ce fait simplement par une flèche de **L** vers cette liste, comme ci-dessous :

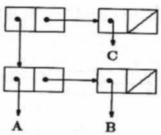


On peut aisément reconnaître ce que font les fonctions **CAR** et **CDR** : elles ramènent respectivement le pointeur du champ **CAR** et **CDR**.

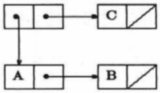
Regardons quelques exemples de cette représentation : La liste **(A B C)** sera donc implémentée comme :



et la liste **((A B) C)** sera, en terme de doublets, implémentée comme :

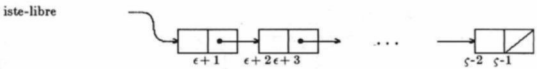


Nous pouvons encore simplifier en ne fléchant que les pointeurs vers des listes; ce qui donne pour cette dernière liste :



N'oublions jamais que tout pointeur et tout atome n'est – en réalité – qu'une adresse: l'adresse mémoire où se trouve l'objet correspondant.

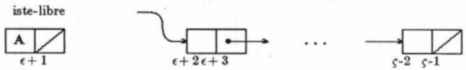
Au lancement de LISP toute la zone liste est organisée comme une *liste libre*, c'est-à-dire : tous les doublets sont *chaînés* entre eux pour former une énorme liste :



Chaque appel de la fonction **CONS** enlève un doublet de cette liste libre. Ainsi, si le premier appel de **CONS** est

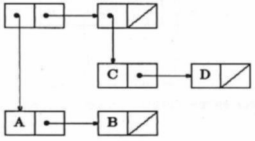
(CONS 'A NIL)

CONS cherche un doublet dans la liste libre, l'enlève de cette liste, insère dans son champ **CAR** un pointeur vers l'atome **A** et dans son champ **CDR** un pointeur vers l'atome **NIL**, ce qui donne :

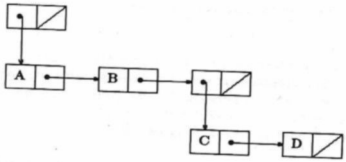


Quand la liste libre est vide, **CONS** met en marche un mécanisme de récupération de mémoire, qui construit une nouvelle liste libre à partir des doublets qui ne sont plus utilisés.

Ci-dessous encore deux exemples de représentation de listes sous forme de doublets : d'abord la liste **((A B) (C D))** :

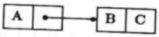


et ensuite la liste **((A B (C D)))** :



Si nous avons dit que le contenu du champ **CAR** ou du champ **CDR** d'un doublet peut être un pointeur vers n'importe quel objet LISP, il peut arriver que nous trouvions dans le champ **CDR** d'un doublet un pointeur non pas vers une liste mais un pointeur vers un atome, comme, par exemple dans le doublet ci-dessus :

ou encore dans la liste suivante :



Actuellement, nous ne pouvons pas représenter ces deux listes de façon externe. C'est pourquoi, pour lire ou écrire de telles listes, nous avons besoin d'une notation supplémentaire : pour représenter des listes qui contiennent dans le champ **CDR** un pointeur vers un atome différent de **NIL**[2] nous utiliserons le caractère '.' (point) comme séparateur entre le champ **CAR** et le champ **CDR** de tels doublets. Ce qui donne, pour la première liste :

(A . B)

et, pour la deuxième liste :

(A B . C)

Naturellement, ceci change notre définition de la fonction **CONS** du chapitre 2 : plus aucune raison n'existe pour demander que le deuxième argument du **CONS** soit une liste. Bien au contraire, la liste **(A . B)** est le résultat de l'appel suivant :

(CONS 'A 'B)

et le **CDR** de cette liste est l'*atome* **B** !

Bien évidemment, nous avons besoin d'une représentation externe (comme suite de caractères) reflétant l'*implémentation* des listes sous formes de doublets. Pour cela, nous allons reprendre la

2. Rappelons qu'un pointeur vers l'atome **NIL** dans le champ **CDR** indique la fin d'une liste.

définition classique des expressions symboliques (ou des S-expressions) de John McCarthy, l'inventeur de LISP :

Tout objet LISP est une S-expression. Le type le plus élémentaire d'une S-expression est un *atome*. Tout les autres S-expressions sont construites à partir de symboles atomiques, les caractères "(", ")" et "." (le point). L'opération de base pour former une S-expression consiste à en combiner deux pour produire une S-expression plus grande. Par exemple, à partir des deux symboles **A** et **B**, on peut former la S-expression **(A . B)**.

La règle exacte est : une S-expression est soit un symbole atomique soit une composition des éléments suivants (dans l'ordre) : une parenthèse ouvrante, une S-expression, un point, une S-expression et une parenthèse fermante.

Voici, suivant cette définition, quelques exemples de S-expressions et leur représentation interne sous forme de doublets :

A

évidemment, c'est un atome et ne peut donc pas être représenté sous forme de doublet.

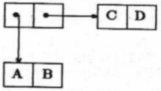
(A . NIL)

(A . (B . NIL))

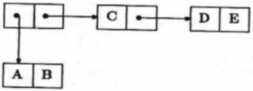
(A . B)

((A . B) . C)

((A . B) . (C . D))



((A . B) . (C . (D . E)))



Pour passer de cette notation des S-expressions à la notation des listes à laquelle nous sommes habitués, il suffit de :

1. remplacer chaque occurrence d'un point suivi d'une parenthèse ouvrante par un caractère espace et d'éliminer la parenthèse fermante correspondante, ce qui transforme par exemple **(A . (B . NIL))** en **(A B . NIL)**,

2. éliminer les occurrences d'un point suivi par **NIL**, ce qui transforme par exemple **(A B . NIL)** en **(A B)**.

Ainsi, pour les S-expressions ci-dessus nous avons les correspondances :

A	≡	**A**
(A . NIL)	≡	**(A)**
(A . (B . NIL))	≡	**(A B)**
(A . B)	≡	**(A . B)**
((A . B) . C)	≡	**((A . B) . C)**
((A . B) . (C . D))	≡	**((A . B) C . D)**
((A . B) . (C . (D . E)))	≡	**((A . B) C D . E)**

Naturellement, les *paires pointées*, c'est-à-dire les doublets où le **CDR** est un pointeur vers un atome différent de **NIL**, restent représentées à l'aide du caractère '.'.

15.3 LES FONCTIONS DE MODIFICATION DE LISTES

Les fonctions de traitement de listes que nous connaissons jusqu'à maintenant, **CAR**, **CDR**, **REVERSE** etc, ne modifient pas les listes données en argument. Ainsi, si à un instant donné la valeur de la variable **L** est la liste **(A B C)**, le calcul du **CDR** de **L** livre bien la liste **(B C)**, et la

valeur de **L** est toujours l'ancienne liste **(A B C)**.

Dans ce paragraphe nous allons étudier quelques fonctions *destructives*, de traitement de listes, c'est-à-dire des fonctions qui modifient après leur évaluation les listes passées en argument.

Les deux fonctions les plus générales dans ce domaine sont les fonctions **RPLACA** et **RPLACD**. La première *modifie physiquement* le **CAR** de son premier argument, la deuxième *modifie physiquement* le **CDR** de son argument. Voici leur définition :

(RPLACA *liste expression***)** → *nouvelle-liste*
en plus : **RPLACA** remplace physiquement le **CAR** de la *liste* premier argument par la valeur de l'*expression* donnée en deuxième argument. La *liste* ainsi modifiée est ramenée en valeur.

(RPLACD *liste expression***)** → *nouvelle-liste*
en plus : **RPLACD** remplace physiquement le **CDR** de la *liste* premier argument par la valeur de l'*expression* donnée en deuxième argument. La *liste* ainsi modifiée est ramenée en valeur.

Voici quelques exemples :

```
(RPLACA '(A B C) 1)              →  (1 B C)
(RPLACA '((A B) (C . D)) '(1 2 3))  →  ((1 2 3) (C . D))
(RPLACD '(A B C) 1)              →  (A . 1)
(RPLACD '((A B) (C . D)) '(1 2 3))  →  ((A B) 1 2 3)
```

Dans les exemples ci-dessus nous utilisons ces fonctions pour calculer une valeur. Mais pensons aussi au fait que ces deux fonctions *modifient* les listes données en argument. Pour cela, regardons donc la fonction suivante :

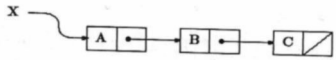
```
(DEFUN FOO (X)
       (CONS (RPLACA X 1) X))
```

Que fait cette fonction ? Est-ce que le résultat est le même si nous remplaçons le corps par **(CONS (CONS 1 (CDR X)) X)** ?

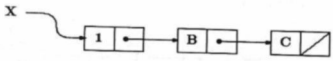
Regardons donc en détail. Si nous appelons cette fonction avec :

```
(FOO '(A B C))
```

la variable **X** sera liée à la liste **(A B C)**. Avant de commencer l'évaluation du corps de la fonction nous avons donc :



L'évaluation du corps de la fonction se réduit à l'évaluation du **CONS**. Il faut donc d'abord évaluer les deux arguments **(RPLACA X 1)** et **X**. **RPLACA** modifie la liste **X**, ce qui donne :



et c'est cette nouvelle liste qui sera le premier argument pour **CONS**.

L'évaluation de **X** (deuxième argument) produira également cette nouvelle liste **(1 B C)**, puisque maintenant cette liste est la valeur de **X**. Le résultat de l'appel **(FOO '(A B C))** est donc la liste :

((1 B C) 1 B C)

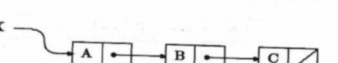
et non point la liste **((1 B C) A B C)**.

Quand vous utilisez ces fonctions, faites attention, leur usage est très dangereux, soyez sûrs que vous voulez vraiment *modifier* la structure interne des listes !

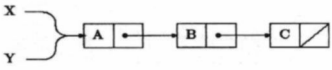
Imaginons que la personne qui a construit le programme **FOO** ci-dessus veuille, en réalité, construire une fonction qui produise une liste contenant en **CAR** la même liste que le **CDR** avec le premier élément remplacé par l'atome **1**, qu'il voulait donc que **(FOO '(A B C))** livre **((1 A B) A B C)**. Constatant l'erreur, elle construit alors la nouvelle version que voici :

```
(DEFUN FOO (X)
  (LET ((X X) (Y X))
       (CONS (RPLACA X 1) Y)))
```

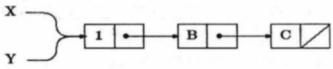
Est-ce que cette solution est meilleure ? L'idée est claire : puisque **RPLACA** modifie la liste, on sauvegarde, avant de faire le **RPLACA**, la valeur de **X** dans la variable **Y**, et cette variable sera ensuite le deuxième argument du **CONS**. Regardons donc ce qui se passe, et appelons à nouveau **(FOO '(A B C))**. Comme préalablement, la variable **X** sera liée à la liste **(A B C)** :



ensuite, dans le **LET**, **X** sera reliée à cette même liste, et **Y** sera liée à la valeur de **X** ce qui donne :

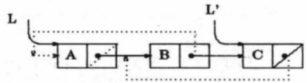


ensuite, comme préalablement, l'évaluation du premier argument du **CONS** livre la liste **(1 B C)** et modifie la valeur de **X**, ce qui donne :

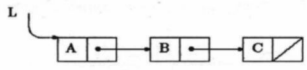


Etant donné que la variable **X** *et* la variable **Y** sont liées à la *même* liste (c'est-à-dire : la C-valeur de **X** et la C-valeur de **Y** contiennent la même adresse de la zone liste), la fonction **RPLACA** modifie les valeurs des deux variables ! Ainsi, même avec cette sauvegarde initiale de la valeur de **X**, rien n'est changé : on se trouve dans le même cas que précédemment, c'est-à-dire : le résultat sera la liste **((1 B C) 1 B C)**.

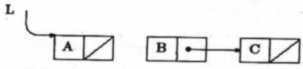
Faisons un petit exercice et écrivons une fonction **FREVERSE** qui doit inverser une liste *sans* construire une nouvelle liste, c'est-à-dire : sans faire un seul appel à la fonction **CONS**. Pour résoudre ce problème nous changerons les pointeurs de la liste originale, de telle sorte que les *mêmes* doublets seront, après exécution de la fonction, chaînés à l'envers du chaînage original. Ci-dessous, la liste **L** aux trois éléments **A**, **B** et **C**, où j'ai dessiné en pointillé les pointeurs tels qu'ils doivent être après inversement de la liste et où **L'** est la valeur de **L** à la fin :



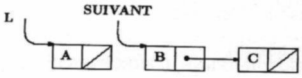
Regardons, sur une trace graphique comment nous devons procéder. Au début nous avons un pointeur, nommons-le **L**, vers la liste initiale :



Evidemment, la première chose à faire est de remplacer le **CDR** du premier doublet par **NIL**, ce qui nous donne :

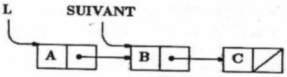


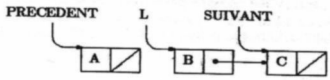
Le pas suivant consiste à remplacer le pointeur du **CDR** du deuxième doublet par un pointeur vers ce premier doublet juste modifié. Mais, après cette opération du *remplacement* du **CDR** du premier doublet par **NIL**, nous n'avons plus aucun accès au deuxième doublet. Donc : avant de faire cette opération nous devons garder quelque part un pointeur vers la suite de la liste. Appelons ce pointeur **SUIVANT**. Nous devons donc avoir la configuration suivante :

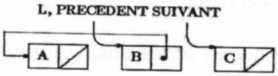


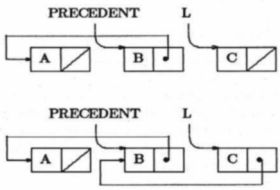
Si maintenant nous remplaçons le **CDR** du deuxième doublet par un pointeur vers la valeur de **L**, nous perdons à nouveau la suite de la liste. Nous devons donc d'abord avancer le pointeur **SUIVANT**, mais sans perdre sa valeur actuelle, puisque, dans la suite nous devons établir un pointeur vers ce doublet pointé actuellement par **SUIVANT**.

La solution qui s'impose sera donc d'introduire un pointeur supplémentaire, nommons-le **PRECEDENT**, et que **L** pointera à tout instant sur la nouvelle tête de liste, **PRECEDENT** pointera vers le doublet qui précédait dans la liste originale l'élément pointé par **L**. Le pointeur **SUIVANT** indiquera en permanence les éléments suivants, pas encore modifiés. Ce qui nous donne la suite d'opérations ci-après :









Voici donc la définition LISP de **FREVERSE** :

```
(DEFUN FREVERSE (L)
  (LABELS ((FREV-INTERNE (L PRECEDENT)
             (IF L
                 (FREV-INTERNE (CDR L) (RPLACD L PRECEDENT))
                 PRECEDENT)))
    (FREV-INTERNE L NIL)))
```

Dans cette version nous n'avons pas besoin d'un pointeur **SUIVANT** explicite puisque la liaison de **L** au **CDR** de **L**, donc à l'adresse du doublet suivant se fait en parallèle à la destruction de ce même champ **CDR** !

Voilà donc une fonction qui inverse une liste sans faire le moindre **CONS**, c'est-à-dire : une fonction qui n'utilise aucun doublet supplémentaire. **FREVERSE** sera donc sensiblement plus rapide que la fonction **REVERSE** que nous avons vue au chapitre 6.

Mais n'oublions jamais que **FREVERSE** est *destructive*, et qu'il faut prendre les mêmes précautions qu'avec la fonction **FOO** ci-dessus. Cela signifie que si nous avons, par exemple, la fonction :

```
(DEFUN BAR (L)
  (APPEND (FREVERSE L) L))
```

l'appel **(BAR '(A B C))** ne nous livre pas la liste **(C B A A B C)**, mais la liste **(C B A A)**. Vous voyez pourquoi ?

Les deux fonctions **RPLACA** et **RPLACD** sont suffisantes pour faire n'importe quelle opération (destructive) sur des listes. Ainsi nous pouvons définir une fonction de concaténation de deux listes :

```
(DEFUN LAST (L) (IF (ATOM (CDR L)) L (LAST (CDR L))))

(DEFUN NCONC (LISTE1 LISTE2)
  (RPLACD (LAST LISTE1) LISTE2) LISTE1)
```

Nous pouvons construire une fonction **ATTACH** qui insère un nouvel élément en tête d'une

liste, de sorte que le premier doublet de la liste originale soit également le premier doublet de la liste résultat :

```
(DEFUN ATTACH (ELE LISTE)
  (LET ((AUX (LIST (CAR LISTE))))
       (RPLACD (RPLACA LISTE ELE) (RPLACD AUX (CDR LISTE)))))
```

Nous pouvons construire la fonction inverse de **ATTACH** : **SMASH**, qui enlève physiquement le premier élément d'une liste, de manière telle que le premier doublet de la liste originale soit également le premier doublet de la liste résultat :

```
(DEFUN SMASH (LISTE)
  (RPLACD (RPLACA LISTE (CADR LISTE)) (CDDR LISTE)))
```

Toutes ces fonctions ont en commun le fait de modifier physiquement les structures des listes données en arguments, et donc que les valeurs de *toutes* les variables qui pointent vers une telle liste se trouvent modifiées.

15.4 LES FONCTIONS EQ ET EQUAL

Au chapitre 6 nous avons construit la fonction **EQUAL**, qui teste l'égalité de deux listes en traversant les deux listes en parallèle et en comparant l'égalité des atomes avec la fonction **EQ**. Cette fonction teste donc si les deux listes ont la même structure et si elles sont composées des mêmes atomes dans le même ordre.

La fonction **EQ**, nous disions, ne peut tester que l'égalité des atomes. Précisons : chaque atome LISP n'existe qu'une seule fois. Ainsi, si nous avons l'expression :

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

les deux occurrences de l'atome **A** sont deux occurrences d'un pointeur vers le *même* atome. Pour prendre un autre exemple, chaque occurrence de l'atome **CONS** à l'intérieur d'un programme se réfère précisément au même atome ! De ce fait, le test d'égalité de la fonction **EQ** est en réalité un test d'égalité de deux adresses (et non point un test d'égalité d'une suite de caractères). C'est donc un test d'identité : en fait, les deux atomes **A** dans l'expression ci-dessus sont des références à un seul et unique objet.

Sachant que **EQ** se réduit à une comparaison d'adresses et que des listes sont également accessibles à travers leurs adresses, nous pouvons utiliser cette fonction pour tester l'*identité* de deux listes.

Afin d'expliciter construisons deux listes (que nous nommerons **X** et **Y**) : la première, **X**, sera construite par l'appel

```
(LIST '(A B) '(A B))
```

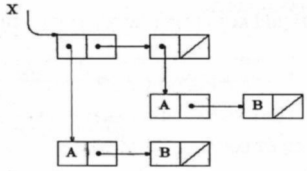
et la deuxième, **Y**, par l'appel :

(LET ((X '(A B))) (LIST X X))

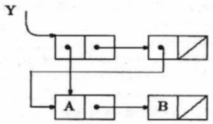
Regardez bien la différence entre ces deux listes ! Les deux listes s'écrivent de façon externe :

((A B) (A B))

mais, de façon interne, la première est représentée par :



et la deuxième par :



Bien que ces deux dernières listes soient – de façon interne – très différentes, elles ont toutes les deux une écriture externe identique !

La différence entre la liste **X** et la liste **Y** devient apparente si nous testons l'identité des deux sous-listes :

(EQUAL (CAR X) (CADR X))	→	**T**
(EQUAL (CAR Y) (CADR Y))	→	**T**
(EQ (CAR X) (CADR X))	→	**NIL**
(EQ (CAR Y) (CADR Y))	→	**T**

Evidemment, le premier et le deuxième élément de **Y** sont des pointeurs vers la même liste : la fonction **EQ** livre donc **T**.

On dit que de telles listes sont *partagées*. Au niveau de l'impression standard, aucun moyen n'existe pour distinguer des listes partagées des listes non-partagées. Naturellement, des listes

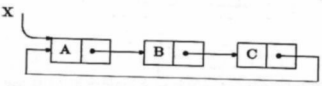
partagées sont toujours issues d'opérations sur des listes (c.à.d.: elles ne peuvent pas être données telles quelles au terminal).

15.5 LES LISTES CIRCULAIRES

Regardons encore quelques listes partagées : quel est le résultat de l'opération suivante ?

(LET ((X '(A B C))) (NCONC X X))

NCONC est la fonction précédemment définie qui concatène physiquement deux listes : nous concaténons donc à la liste **X**, c'est-à-dire : **(A B C)**, la liste **X** elle-même, ce qui donne :



Il s'agit d'une liste *circulaire* : une liste sans fin. Son impression donne quelque chose comme :

(A B C A B C A B C A B C . . .)

Donc, a priori, il n'est pas possible d'imprimer de telles listes. Remarquons que ces listes peuvent être très utiles pour représenter des structures de données infinies, mais qu'elles sont très dangereuses : beaucoup de fonctions de recherche risquent de ne jamais se terminer sur de telles listes. Par exemple, si nous cherchons l'atome **D** à l'intérieur de cette liste **X**, l'appel

(MEMQ 'D X)

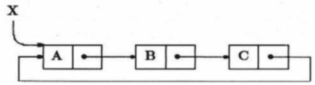
va boucler éternellement, puisque la fonction **MEMQ** compare le premier élément de la liste avec l'argument cherché, et si ce n'est pas le même, elle recommence l'opération sur le **CDR** de la liste, ceci jusqu'à ce qu'elle arrive soit à une occurrence de l'élément cherché soit à la fin de la liste. Etant donné que notre liste **X** n'a pas de fin, la recherche de l'élément qui effectivement ne se trouve pas dans la liste n'a pas de fin non plus.

Pour écrire une fonction qui cherche l'occurrence d'un élément donné dans une liste circulaire, nous avons donc besoin de garder en permanence un pointeur vers le début de la liste, afin de pouvoir tester, avec la fonction **EQ**, si le **CDR** de la liste est 'EQual' à la liste entière et ainsi d'éviter de répéter la recherche à l'infini.

Ci-après la définition du prédicat **C-MEMQ** testant l'occurrence d'un élément à l'intérieur d'une liste circulaire :

```
(DEFUN C-MEMQ (OBJ L)
  (LABELS ((INTERN-C-MEMQ (LL OBJ L) (COND
        ((ATOM LL)(EQ LL OBJ)) ; le test d'égalité ;
        ((EQ (CAR LL) OBJ) T)  ; le vrai test d'égalité ;
        ((EQ L (CDR LL)) NIL)  ; le test de circularité ;
        (T (INTERN-C-MEMQ (CDR LL) OBJ L)))))
    (INTERN-C-MEMQ L OBJ L)))
```

Si on appelle cette fonction avec l'atome **D** comme premier argument, et comme deuxième argument la liste circulaire :



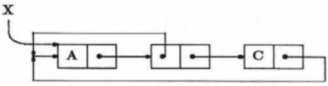
le résultat est **NIL**.

Naturellement, si le premier argument se trouve à l'intérieur de la liste le résultat est **T**.

Modifions-donc notre liste **X** avec l'opération suivante :

```
(RPLACA (CDR X) X)
```

Cette opération nous livre une liste doublement circulaire : une boucle sur un **CAR**, et une boucle sur un **CDR** :



C-MEMQ, la fonction de recherche d'un élément dans une liste circulaire ne marche plus sur cette liste, car aucun test n'est effectué vérifiant la circularité sur le **CAR** de la liste.

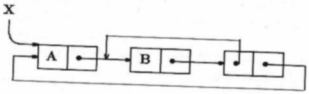
Pour généraliser notre fonction **C-MEMQ** nous devons tester aussi bien le **CAR** que le **CDR** de la liste en vue d'une circularité vers le début de la liste. Voici alors la nouvelle fonction **C-MEMQ** d'abord en Le_LISP :

```
(DE C-MEMQ (OBJ L)
  (IF (EQ (CAR L) OBJ) T
      (LETN CM-INTERNE ((LL (CDR L))) (COND
            ((ATOM LL) (EQ LL OBJ))
            ((EQ L LL) ())
            (T (OR (CM-INTERNE (CAR LL)) (CM-INTERNE (CDR LL))))))))
```

ensuite en CommonLISP :

```
(DEFUN C-MEMQ (OBJ L)
  (LABELS ((INTERN-C-MEMQ (LL OBJ L) (COND
        ((ATOM LL)(EQ LL OBJ))
        ((EQ L LL) NIL)
        (T (OR (INTERN-C-MEMQ (CAR LL) OBJ L)
               (INTERN-C-MEMQ (CDR LL) OBJ L))))))
    (IF (EQ (CAR L) OBJ) T
        (INTERN-C-MEMQ (CDR L) OBJ L))))
```

Finalement, nous pouvons généraliser notre approche à des listes à circularités arbitraires, c'est-à-dire à des circularités qui ne portent que sur des sous-parties de la liste, comme par exemple la liste **X** ci-dessous :



Dans ces cas généraux, en plus des tests de circularités 'globales' à partir des champs **CAR** et **CDR**, il faut garder une pile de toutes les sous-listes rencontrées, afin de reconnaître une circularité sur des sous-structures arbitraires. Ce qui donne, encore d'abord en Le_LISP :[3]

```
(DE C-MEMQ (OBJ L)
   (LETN CM-INTERNE ((L L) (AUX NIL)) (COND
         ((ATOM L) (EQ OBJ L))
         ((MEMQ L AUX) ())
         (T (OR (CM-INTERNE (CAR L) (CONS L AUX))
                (CM-INTERNE (CDR L) (CONS L AUX)))))))
```

et ensuite en CommonLISP :

```
(DEFUN C-MEMQ (OBJ L &OPTIONAL AUX) (COND
        ((ATOM L)(EQ L OBJ)) ; test d'égalité ;
        ((MEMQ L AUX) NIL)   ; test de circularité ;
        (T (OR (C-MEMQ OBJ (CAR L) (CONS L AUX))
               (C-MEMQ OBJ (CDR L) (CONS L AUX))))))
```

Evidemment, la fonction **MEMQ** utilisée dans la ligne

```
((MEMQ L AUX) ())
```

est la fonction standard **MEMQ**, dont la définition se trouve au chapitre 6 dans l'exercice 4 : c'est une fonction qui procède au test d'égalité avec la fonction **EQ** !

3. Cette version de la fonction **C-MEMQ** est inspirée de la fonction **SKE** de Patrick Greussay.

15.6 LES AFFECTATIONS

Après tant de fonctions de modifications terminons ce chapitre par un bref regard sur les fonctions de modifications des valeurs de variables.

Jusqu'à maintenant, l'unique moyen que nous connaissions pour donner des valeurs à des variables est la *liaison dynamique*, c'est-à-dire : la liaison des valeurs à des variables pendant les appels de fonctions. Cette liaison dynamique se comporte de telle sorte qu'à l'appel d'une fonction les variables prennent les nouvelles valeurs déterminées par les arguments de l'appel, et qu'à la sortie de la fonction les anciennes valeurs de ces variables soient restituées.

En LISP, il existe deux autres manières pour donner des valeurs à des variables en utilisant les fonctions d'*affectation* **SETQ** et **SET**. La fonction **SETQ** prend – au minimum – deux arguments, le premier doit être le nom d'une variable, le second une expression LISP quelconque. L'*effet* de l'évaluation d'un appel de la fonction **SETQ** est de donner à la variable premier argument comme nouvelle valeur le résultat de l'évaluation de l'expression deuxième argument. C'est donc une fonction de modification de la C-valeur des atomes (sans possibilité de restauration automatique ultérieure). La *valeur* de l'évaluation d'un appel de la fonction **SETQ** est le résultat de l'évaluation de l'expression.

Voici un exemple d'une interaction avec LISP utilisant cette fonction :

```
? (SETQ X 1)            ; X prend la valeur numérique 1
=1
? X                     ; examinons la valeur de X
=1
? (SETQ X (+ X 3))      ; modifions la valeur de X
=4
? X                     ; examinons la nouvelle valeur de X
=4
```

et voici une manière élégante d'échanger les valeurs numériques de **X** et **Y** sans passer par une variable intermédiaire :

```
? (SETQ X 10)           ; la valeur initiale de X
=10
? (SETQ Y 100)          ; la valeur initiale de Y
=100
? (SETQ X (- X Y))
=-90
? (SETQ Y (+ X Y))
=10
? (SETQ X (- Y X))
=100
? X                     ; la valeur de X est bien
=100                    ; l'ancienne valeur de Y
? Y                     ; et la valeur de Y est bien
=10                     ; l'ancienne valeur de X
```

La fonction **SETQ** est principalement utilisée dans la construction et la modification de bases de données partagées par un ensemble de fonctions ; elle permet de donner à l'extérieur de toute fonction des valeurs à des variables, appelées alors des variables *globales*.

Notons finalement qu'en général la fonction **SETQ** admet $2*n$ arguments. Voici alors la définition formelle de **SETQ** :

(**SETQ** var_1 val_1 . . . var_n val_n) → *valeur*(val_n)
en plus : la C-valeur de var_1 est le résultat de l'évaluation de val_1, . . ., la C-valeur de var_n est le résultat de l'évaluation de val_n.

La fonction **SET** est identique à **SETQ**, mis à part le fait qu'elle évalue son premier argument. L'évaluation du premier argument doit donc livrer le nom d'une variable. En guise de définition de la fonction **SET** disons que les expressions :

(**SETQ** *variable expression*)

et

(**SET** (**QUOTE** *variable*) *expression*)

sont équivalentes. La fonction **SET** permet des *indirections*. Par exemple, la suite d'instructions :

```
(SETQ A 'B)
(SET A 3)
```

donne à la variable **B** la valeur numérique **3**.

A titre d'exemple, voici une version supplémentaire de la fonction **REVERSE**. Sa particularité est de modifier la valeur de la variable donnée en argument. C'est une fonction Le_LISP, nous verrons au chapitre suivant comment faire la même chose en CommonLISP.

```
(DF PREVERSE (L RES)
    (LETN PREVERSE ((LISTE (EVAL L)))
        (IF (NULL LISTE) (SET L RES)
            (SETQ RES (CONS (CAR LISTE) RES)
                LISTE (CDR LISTE))
            (PREVERSE LISTE))))
```

Voici une utilisation de cette fonction :

```
? (SETQ K '(A B C))      ; pour donner une valeur à la variable K
=(A B C)
? (REVERSE K) NIL        ; appelons la fonction REVERSE standard
=(C B A)                 ; le résultat
? K                      ; examinons la valeur de K
=(A B C)                 ; ça doit bien être la même
? (PREVERSE K)           ; appelons maintenant la nouvelle fonction
=(C B A)                 ; le résultat
```

```
? K                    ; regardons la valeur de K
=(C B A)               ; elle a bien été modifiée
                       ; voila to'ute la différence
```

D'ailleurs, la fonction **SETQ** peut être définie en Le_LISP avec la fonction **SET** de la manière suivante :

```
(DF SETQ X
    (LET ((VARIABLE (CAR X)) (VALEUR (EVAL (CADR X))) (RESTE (CDDR X)))
         (SET VARIABLE VALEUR)
         (IF RESTE (EVAL (CONS 'SETQ RESTE)) VALEUR)))
```

Pour finir ce chapitre sur la programmation *impure*,[4] examinons brièvement la fonction d'affectation générale de CommonLISP, la fonction **SETF**. Nous avons déjà vu cette fonction au chapitre 8, comme un moyen d'écrire la fonction **PUTPROP**. Ici nous l'examinerons plus en détail.

D'abord, rappelons les diverses fonctions pour changer des structures de données LISP déjà vues : ce sont les fonctions **SET** et **SETQ**, que nous venons de voir, et les fonctions **RPLACA** et **RPLACD**, les fonctions pour changer le **CAR** et **CDR** d'une liste. L'effet de toutes ces fonctions peut être obtenu avec la fonction **SETF**. Par exemple :

```
(SETQ X 1)
```

fait exactement la même chose que :

```
(SETF X 1)
```

et l'appel :

```
(SET X 1)
```

fait exactement la même chose que :

```
(SETF (SYMBOL-VALUE 'X) 1)
```

sachant que la fonction **SYMBOL-VALUE** retourne la valeur d'un symbole.

SETF attend deux arguments : une fonction d'*accès*, telle que **SYMBOL-VALUE** dans l'exemple ci-dessus, et la nouvelle *valeur* de la partie accédée. L'effet d'un appel de **SETF** est de changer la valeur de cette partie d'une structure LISP qui a été accédée. **SETF** fonctionne avec presque toutes les fonctions d'accès standard. Voici quelques exemples de son utilisation et des expressions équivalentes qui n'utilisent que des fonctions standard déjà connues :[5]

4. Un programme LISP purement fonctionnel, donc sans effets de bords, est dit être en *LISP pur*; un programme LISP avec des effets de bords est dit être écrit en *LISP impur*. C'est dans ce sens que nous parlons d'une programmation pure ou impure.

(SETF (CAR X) 1)

est la même chose qu'un changement du premier élément d'une liste **X** à la valeur **1**, c'est-à-dire : c'est la même chose que :

(LET ((G824 X)(G825 1)) (PROGN (RPLACA G824 G825) G825))

qui, à son tour, est juste une manière compliquée de dire :

(RPLACA X 1)

La fonction standard **PROGN** utilisée ici est une fonction permettant de grouper plusieurs expressions en une seule. C'est exactement la même chose que les *blocs* des langages algorithmiques classiques, comme par exemple les blocs définis en langage **C** avec les accolades. Sa définition est :

(**PROGN** $expression_1$ $expression_2$ . . . $expression_n$) → $valeur(expression_n)$
et $expression_1$ à $expression_n$ sont évaluées.

Nous l'utilisons ici, puisque la fonction **IF** de CommonLISP n'admet qu'une seule expression dans le cas où l'évaluation du test donne faux. Ainsi, puisque toutes les expressions d'un **PROGN** sont considérées comme une seule, nous utilisons un appel à cette fonction si nous voulons évaluer plus d'une expression dans une action-vraie ou dans une action-fausse d'un **IF**.

Voici l'équivalent **SETF** de **RPLACD** :

(SETF (CDR X) '(A B C))

qui se traduit en quelque chose comme :

(LET ((G833 X)(G834 '(A B C)))
(PROGN (RPLACD G833 G834) G834))

Encore une fois, c'est une manière très compliquée d'écrire la même chose que :

(RPLACD X '(A B C))

Jusqu'à maintenant, les utilisations de **SETF** que nous avons vues, ont toutes été des équivalentes d'appels de fonctions standard. Mais **SETF** est plus générale : elle accepte toute les fonctions d'accès et affecte des valeurs aux parties des structures LISP correspondantes. Nous avons déjà vu l'équivalent de **PUTPROP** avec :

(SETF (GET X Y) *valeur*)

5. Les expressions équivalentes à des appels de la fonction **SETF** ci-dessous correspondent à des macro-expansions. En effet, **SETF** est une macro-fonction, du même type que celles que nous verrons dans le chapitre suivant.

Voici un dernier exemple :

```
(SETF (CDDR X) '(A B C))
```

qui fait la même chose que :

```
(LET ((G824 X) (G825 '(A B C)))
     (PROGN (RPLACD (CDR G824) G825) G825))
```

qui, encore, est juste une façon compliquée d'écrire :

```
(RPLACD (CDR X) '(A B C))
```

CommonLISP effectivement traduit des appels de **SETF** en des appels de fonctions standard. Ainsi, tous les exemples de **SETF** ci-dessus sont effectivement traduits dans leurs expressions équivalentes. Ces fonctions, des fonctions qui traduisent leurs appels dans des expressions équivalentes, sont nommées des *macro-fonctions*. Elles sont le sujet du chapitre suivant. Ici, notons juste que **SETF** accepte comme fonction d'accès toutes les fonctions d'accès standard, telles que **GET**, **CAR**, **CDR**, **CADR**, etc, *et* aussi des fonctions d'accès utilisateurs, *si* ces fonctions ont été définies en tant que macro-fonctions.

Si vous ne vous rappelez pas, à un certain instant, la fonction changeant la valeur d'une structure donnée, ou d'une sous-partie de cette structure, vous pouvez toujours la remplacer par un appel à la fonction d'affectation générale **SETF**.

15.7 EXERCICES

1. Montrez graphiquement l'effet des instructions suivantes :

   a. **(SET 'X '(A B C)) (ATTACH 1 X)**

   b. **(LET ((X '(A B C))) (SMASH X))**

   c. **(SETQ X '(A B C)) (RPLACA (RPLACD X X) X)**

2. Ecrivez la fonction **FDELQ** à deux arguments **ELE** et **LISTE**, qui enlève physiquement toutes les occurrences de l'élément **ELE** à l'intérieur de la liste **LISTE**. (Attention au cas où la liste commence par l'élément **ELE**.)

3. Ecrivez une fonction qui inverse physiquement une liste circulaire simple.

4. Modifiez la fonction **FREVERSE** de manière à inverser une liste sur tous ses niveaux.

5. Définissez la fonction **RPLACB** qui combine les fonctions **RPLACA** et **RPLACD** en remplaçant le **CAR** de la liste premier argument par le **CAR** de la liste deuxième argument et le **CDR** de la liste premier argument par le **CDR** de la liste deuxième argument. Ainsi si la variable **X** a la valeur **(X Y Z)**, le résultat de l'appel

(RPLACB X '(1 2 3))

affecte la valeur **(1 2 3)** à la variable **X**. De plus le premier doublet de la liste originale se trouve à la même adresse que le premier doublet de la liste résultat.

Dans LE_LISP, cette fonction s'appelle **DISPLACE**.

6. Cet exercice ne s'adresse qu'aux programmeurs Le_LISP : La suite d'instructions

(SETQ *variable* **(CAR** *la-liste*))
(SETQ *la-liste* **(CDR** *la-liste*))

est très courante. C'est pourquoi beaucoup de systèmes LISP ont une fonction **NEXTL** définie comme suit :

(NEXTL *variable*) → **(CAR** *(valeur (variable))*) et *variable* prend comme nouvelle valeur **(CDR** *variable*).

Avec cette fonction, la fonction **PREVERSE** peut être redéfinie comme suit :

```
(DF PREVERSE (L RES)
    (LETN PREV-INTERNE ((LISTE (EVAL (CAR L))))
        (IF (NULL LISTE) (SET (CAR L) RES)
            (SETQ RES (CONS (NEXTL LISTE) RES))
            (PREV-INTERNE LISTE))))
```

Définissez alors cette fonction **NEXTL**.

16. UN MICRO-PROJET

Dans les années 30, le mathématicien anglais Alain Turing a développé une machine abstraite nommée d'après lui *la machine de Turing*. C'est une machine à la fois très simple et très puissante : tout ce qu'on sait calculer peut être calculé avec une telle machine. Bien entendu, on ne sait pas démontrer cette affirmation, mais, on ne sait pas la réfuter non plus : c'est une version de la *thèse de Church*, due à Alonzo Church, un autre mathématicien de l'epoque.

Dans ce chapitre, nous allons d'abord apprendre comment programmer une machine de Turing, et, ensuite nous allons construire un interprète pour une telle machine. Cet interprète nous permettra d'exécuter des programmes écrits en *turingule*, le langage des machines de Turing.

Tout d'abord regardons ce qu'est une telle machine de Turing : elle est composée de quatre parties :

- une bande (potentiellement infinie) avec une tête de lecture et d'écriture (un peu comme un magnétophone)
- un ensemble de symboles S_0, S_1, ... $\in \Sigma_{\text{fini}}$ ou 'Σ_{fini}' veut juste dire : un alphabet fini. Nous prendrons, en général, l'alphabet composé juste des caractères **0**, **1** et d'un caractère **B** qui indiquera l'absence d'un caractère 0 ou 1, c'est-à-dire : **B** indique une place sur la bande où rien n'a été écrit.
- une unité centrale comprenant un ensemble fini d'états ou de configurations. Les états sont généralement nommés q_1, q_2, ..., q_m.
- un programme : chaque programme se compose d'un ensemble de quintuplets de la forme

$$(q_i\, S_j\, S_k\, \{\mathbf{R}, \mathbf{L}\}\, q_l)$$

Cette instruction se lit de la manière suivante :

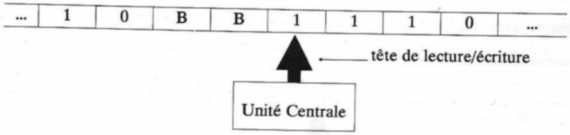
> Si la machine (de Turing) se trouve dans l'état q_i et la tête de lecture/écriture pointe vers une case contenant le caractère S_j il faut alors le remplacer par le caractère S_k. Ensuite, il faut déplacer la tête de lecture d'une position vers la gauche ou vers la droite (ce qui est représenté par les lettres **L** et **R** respectivement) et mettre la machine dans l'état q_l.

Tous ce que sait faire la machine est résumé ici : elle sait reconnaître le caractère qui se trouve dans la case au-dessus de la tête de lecture, elle sait écrire un caractère dans cette case qui peut être le même caractère que celui qui était écrit. Cela veut dire que la lecture est destructive, comme c'est le cas dans les "vrais" ordinateurs. Finalement, elle sait

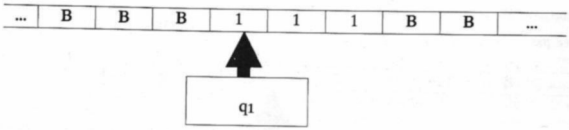
avancer la tête de lecture d'une case vers la gauche – par la commande **L**, pour *left* – ou vers la droite – par la commande **R**, pour *right*.

Bien entendu, un mouvement de la tête de lecture d'une case vers la droite revient au même qu'avancer la bande une case vers la gauche, et avancer la tête de lecture une case vers la gauche revient au même qu'avancer la bande une case vers la droite. Dans la suite nous disons simplement *aller à droite* ou *avancer vers la droite* pour *avancer la tête de lecture une case vers la droite*.

Ci-dessous la présentation graphique d'une machine de Turing :



Pour chaque programme existe une machine de Turing particulière, et le résultat d'un calcul avec une telle machine est l'état de la bande après l'arrêt du calcul.[1] Prenons un exemple, et supposons que nous avons une machine de Turing dans l'état suivant :



c'est à dire, avant le lancement, la tête de lecture pointe vers la première case contenant la lettre **1** et qu'il n'y a sur toute la bande qu'une suite de lettres **1** à droite de cette première occurrence. Dans l'exemple ci-dessus, la suite des caractères **1** se limite à exactement 3. A partir de la case où cette suite de **1** s'arrête la bande est vide. Bien entendu, nous pouvons comprendre cet état de la machine comme indiquant un nombre, le nombre égal au nombre de 1, c'est-à-dire : le nombre correspondant à la longueur de la chaîne de caractères qui n'est composée que par des caractères **1**. Dans l'exemple ci-dessus nous avons le nombre 3. Ce serait donc une représentation des nombres en base unaire. Toute l'arithmétique sur une telle représentation se

1. Naturellement, la présentation des machines de Turing sera plutôt succincte dans ce livre : nous les examinerons exclusivement pour des raisons de leur simulation, comme un exercice de programmation. Mais si vous êtes interessés par un traitement plus détaillé de telles machines, lisez donc l'excellent livre *Finite and Infinite Machines* de Marvin Minsky. Ce livre est non seulement un bon livre sur les automates et les machines abstraites, mais c'est également un livre des plus stimulants pour l'Intelligence Artificielle – évidemment, puisque M. Minsky est un des pères fondateurs de cette discipline.

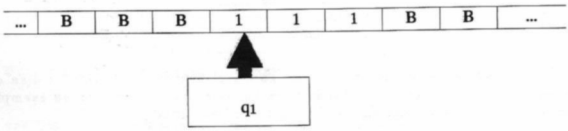
fait de manière analogue à l'arithmétique des *bâtons* que nous avons pratiquée dans notre enfance. L'addition de 1 à un nombre correspond au rajout d'un bâton, et la somme de deux nombres correspond à la juxtaposition des deux nombres.

Construisons donc un programme qui prend une telle situation et incrémente de 1 le nombre représenté. Prenons la décision que l'état q_1 est toujours l'état initial de la CPU. Voici donc ce programme :

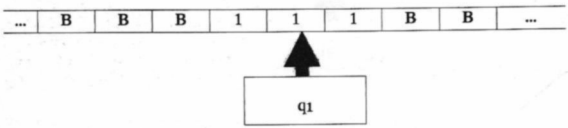
q_1 1 1 R q_1
q_1 B 1 R q_2

Ci-dessous une trace de l'exécution de ce programme :

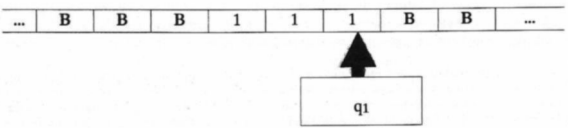
état initial :



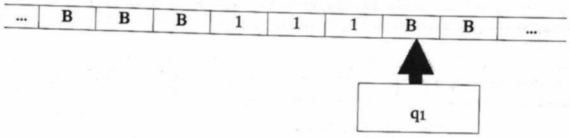
premier pas :



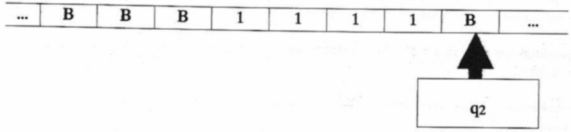
deuxième pas :



troisième pas :



état final :



Dans l'état final la bande contient un caractère **1** de plus que dans l'état initial : le programme a bien incrémenté le nombre initial.

La machine s'arrête dès qu'elle se trouve dans un état q_i qui ne se trouve en tête d'aucun quintuplet, c'est-à-dire dès qu'elle se trouve dans un état pour lequel aucune indication d'action n'est donnée. L'autre cas d'arrêt d'une telle machine est que la machine se trouve dans un état q_j pointant vers un caractère S_k et aucun quintuplet commençant par q_j ne contient le caractère S_k en deuxième position, c'est-à-dire : dans l'état q_j aucune action n'est prévue pour le cas d'un caractère S_k.

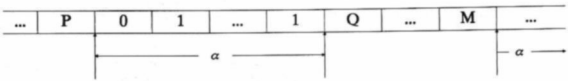
Revenons un instant sur notre programme *turingule* (c'est comme cela qu'on appelle les programmes pour les machines de Turing) d'incrémentation d'un nombre : si nous voulons que la machine s'arrête avec la tête de lecture au début du nombre il suffit de modifier le programme comme suit :

```
q1 I I R q1
q1 B I L q2
q2 I I L q2
q2 B B R q3
```

Remarquez que l'ordre dans lequel nous donnons les quintuplets (les instructions) n'a aucune importance. Quel beau langage où l'on peut donner les instructions dans un ordre quelconque ! Malheureusement ce n'est pas le cas en LISP....

Visiblement, la programmation des machines de Turing n'est point compliquée ! Juste pour le

plaisir, prenons un deuxième exemple un peu plus complexe. Imaginons que nous avons une machine de Turing qui sait reconnaître des lettres de l'alphabet $\Sigma = \{0, 1, B, P, Q, M\}$. Sur la bande initiale il y a exactement une occurrence des lettres P, Q et M, comme sur la bande ci-dessous :



Le problème est alors de copier le segment α, compris entre les caractères P et Q, vers la suite de cases commençant à la case contenant la lettre M.

C'est peut-être un bon instant pour interrompre la lecture et essayer de résoudre ce problème par vous même.

Bon, probablement vous avez trouvé la solution tous seuls. Bravo ! Juste pour comparer, voici ma solution :

Supposons que la tête de lecture se trouve sur la case contenant le caractère P. Si vous voulez généraliser, il faudrait alors d'abord chercher la case contenant ce caractère. C'est une petite boucle de déplacements successifs vers la droite, si vous supposez que ce caractère se trouve quelque part à droite de la tête de lecture, ou l'inverse si vous supposez que le caractère se trouve à gauche de la tête de lecture. Dans tous les cas, vous devez faire une des deux suppositions, sinon vous risquez de boucler indéfiniment. Rappelez-vous que la bande est de longueur infinie des deux côtés.[2]

Une fois la tête de lecture pointant vers le P, il faut bouger d'une case vers la droite, lire le caractère, le remplacer par un P, se mettre dans un autre état, disons β, qui fait avancer jusqu'au caractère M. Là il faut remplacer le M par le caractère lu, aller vers la droite et écrire un M, aller vers la gauche jusqu'à la rencontre du P pour le remplacer par le caractère lu originalement et finalement aller vers la droite dans un état prêt à écrire un P et recommencer le tout. Il faut faire ces opérations pour chacun des caractères possibles de l'alphabet – sauf, naturellement, les caractères délimitateurs de chaînes : ici ce sont les caractères P, Q et M.

Donc, voici le programme turingule correspondant :

```
q1 P P R q2
q2 0 P R q3        on a trouvé un caractère 0
q3 0 0 R q3          on avance jusqu'au M
q3 1 1 R q3
q3 Q Q R q3
```

2. Ceci est au moins le cas dans la machine de Turing originale. Mais entre temps nous savons qu'une machine de Turing avec une bande infinie vers un côté seulement est tout à fait équivalente. Vous devinez pourquoi ?

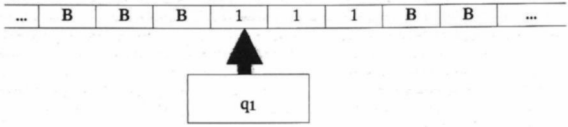
```
q3 M 0 R q4        on remplace le M par 0
q4 0 M L q5        la case à droite sera le M suivant
q4 1 M L q5
q5 0 0 L q5        et le retour jusqu'au P
q5 1 1 L q5
q5 Q Q L q5
q5 P 0 R q2

q2 1 P R q6        maintenant la même chose pour 1
q6 0 0 R q6        on avance jusqu'au M
q6 1 1 R q6
q6 Q Q R q6
q6 M 1 R q7        on remplace le M par 1
q7 0 M L q8        la case à droite sera le M suivant
q7 1 M L q8
q8 0 0 L q8        et le retour jusqu'au P
q8 1 1 L q8
q8 Q Q L q8
q8 P 1 R q2

q2 Q Q R q9        c'est la fin, plus de caractères à copier
```

Maintenant, après avoir vu comment ces machines de Turing fonctionnent et comment il faut les programmer, pensons à l'écriture d'un simulateur d'une telle machine. Par *simulateur* nous entendons un programme qui se comporte comme une machine de Turing : si nous donnons au simulateur une bande, une position initiale de la tête de lecture et un état initial, le programme nous livre la bande finale que la machine de Turing nous aurait livré pour ce même programme turingule, ce même état initial et cette même position de la tête de lecture.

Allons-y pas à pas ! La première question à laquelle nous devons répondre est *comment représenter quelque chose d'infini ?*. Bien entendu, même si nous avons une très grande machine, nous ne pouvons jamais construire quelque chose de réellement infini. Nous allons nous limiter donc à cette partie de la bande qui contient des informations intéressantes pour le programme en cours. Cette partie sera toujours limitée. Notre programme turingule d'incrémentation d'un nombre traitera – même si le nombre est infini – seulement une bande infinie à droite : le nombre doit bien débuter quelque part, d'après notre positionnement du problème. Naturellement, nous n'essayerons pas d'incrémenter un nombre infini. D'ailleurs, est-ce que vous avez une idée comment reconnaître si l'on a affaire à un nombre infini ?

Le premier problème étant réglé par la décision de ne pas traiter pas des structures infinies, il reste les problèmes du *comment représenter la bande ?*, *comment représenter la tête de lecture ?*, *comment représenter un programme turingule ?* et *comment représenter la CPU et ses états ?*, bref : comment représenter les données de notre problème ?

La bande peut bien sûr être représentée comme une liste. Ainsi, l'extrait de bande que nous avons vu dans notre exemple de programme d'incrémentation :



peut être naturellement représenté par la liste :

(B B B 1 1 1 B B)

et, pour indiquer la position de la tête de lecture, nous pouvons insérer un élément supplémentaire juste avant l'élément correspondant à la case pointée par cette tête de lecture/écriture. Pour notre exemple, cela donne la liste ci-dessous, si nous prenons le caractère **T** pour représenter la Tête de lecture/écriture :

(B B B T 1 1 1 B B)

L'activité d'un programme turingle sera alors de changer cette liste, soit en déplaçant le caractère **T** d'une position vers la gauche ou vers la droite, soit en changeant le caractère à droite du signe **T**, soit les deux à la fois. Bien entendu, si nous avons besoin d'un extrait de bande plus long, nous pouvons toujours ajouter une case blanche sur le côté où il manque une case.

Un programme turingle peut également être représenté sous forme d'une liste. Une représentation clairement compréhensible serait une liste contenant des sous-listes où chaque sous-liste est une instruction, donc une liste de 5 éléments de la forme :

(Q1 1 1 R Q1)

qui serait la même chose que l'instruction :

q_1 1 1 R q_1

Notre programme d'incrémentation devient alors la liste suivante :

((Q1 1 1 R Q1)
(Q1 B 1 L Q2)
(Q2 1 1 L Q2)
(Q2 B B R Q3))

Finalement, pour répondre à notre dernière question, la CPU de la machine de Turing sera représentée par le programme que nous allons écrire. L'état de la machine à un certain instant sera la valeur d'une variable particulière de ce programme.

Ce raisonnement sur la représentation des données, sur la question du *quoi* représenter *comment* (et même plus général : que représenter par une donnée sur laquelle on agit, que représenter par un programme qui agit sur ces données) se présente dans tout projet de programmation et est probablement le problème le plus crucial dans toute programmation. Ne suivez *jamais* la première intuition, déterminée complètement par l'énoncé du problème. Les problèmes ne sont que rarement présentés d'une manière proche des ordinateurs, proche de la programmation. Faites plusieurs plans, des variations, des raffinements, etc, il y a (presque) *toujours* une meilleure solution que celle qu'on a prévue ou construite.

Ce chapitre existe principalement pour vous montrer, à travers un exemple, toute cette problématique.

Maintenant, il devient vraiment temps que nous commençions la construction du programme. Regardons d'abord un appel possible du programme **TURING** :

```
(TURING
        '(T 0 0 1 1 B)      ; la bande initiale, T indique la position de la
                            ; tête de lecture/écriture. Le caractère B désigne
                            ; une case blanche, la machine peut en introduire
                            ; au début ou à la fin de la bande si nécessaire.
                            ; Ainsi nous gardons le caractère virtuellement
                            ; infini de la bande
        '((Q1 B B R Q1)     ; ensuite la liste d'instructions (le programme)
          (Q1 0 0 R Q1)     ; ceci est le programme qui incrémente le nombre
          (Q1 1 1 R Q2)     ; rencontré sur la bande
          (Q2 1 1 R Q2)
          (Q2 B 1 R Q3)     ; tous les états q_n qui ne se trouvent pas en
          (Q2 0 1 R Q3))    ; position initiale d'un quintuplet, sont considérés
                            ; comme des états finaux
        'Q1)                ; l'état initial
```

Notre fonction principale, **TURING** a donc trois arguments : une description de l'extrait de la bande qui nous intéresse, un programme et un état initial. Prenons la convention que si la bande ne contient pas de tête de lecture/écriture, le symbole **T**, la machine suppose que celui-ci se trouve en tête de la bande donnée en premier argument. Voilà, nous pouvons commencer à écrire la fonction :

```
(DEFUN TURING (BANDE PROGRAMME ETAT)
  (OR (MEMQ T BANDE)(SETQ BANDE (CONS T BANDE)))
    (LET
      ((INSTR
          (FIND_INSTRUCTION (CARACTERE_ACTUEL BANDE)
                BANDE PROGRAMME ETAT)))
    (TURING (EXECUTE INSTR BANDE) PROGRAMME (CAR (LAST INSTR)))))
```

La première ligne de la fonction regarde si une position de la tête de lecture/écriture est donnée, si ce n'est pas le cas, nous l'introduisons en tête de la liste. Ainsi, nous sommes assurés dans la suite qu'il y a toujours une indication du lieu où se trouve la tête de lecture/écriture. Ensuite, avec la fonction **CARACTERE_ACTUEL** que nous allons écrire par la suite, nous cherchons le caractère sur lequel pointe la tête de lecture. Ce caractère, la bande, le programme et l'état

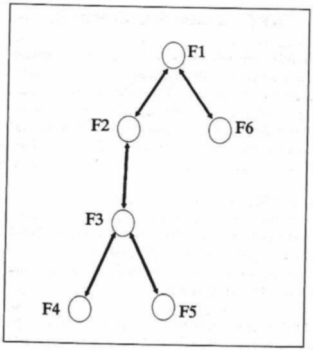
donné en argument seront ensuite envoyés à la fonction **FIND_INSTRUCTION** (une autre fonction qui reste à écrire) qui, elle, doit trouver l'instruction – le quintuplet – qui s'applique dans la situation actuelle. Une fois trouvée la bonne instruction, elle sera gardée dans la variable **INSTR**, nous l'envoyons à la fonction **EXECUTE** (encore une fonction à développer par la suite !) qui doit interpréter cette instruction et donc livrer en résultat la bande modifiée. Rappelons que le dernier élément d'une instruction donne l'état suivant, l'état dans lequel la machine se trouvera après exécution de la commande. C'est pourquoi nous donnons ce dernier élément de l'instruction comme dernier argument à l'appel récursif de **TURING**. Le premier argument pour cet appel sera la bande résultant de l'exécution de l'instruction et le deuxième argument sera, bien entendu, le programme.

16.1 LES ECHAPPEMENTS

Rappelons qu'une machine de Turing s'arrête si elle se trouve dans un état qui soit ne se trouve pas en première position d'un quintuplet, soit ne sait pas traiter le caractère pointé actuellement. Comment faire arrêter notre simulateur ? Pour l'instant aucun arrêt n'est prévu, puisque nous rappelons la fonction **TURING** indéfiniment. La fonction qui peut décider de l'arrêt est, d'après notre analyse, la fonction **FIND_INSTRUCTION**, puisque c'est elle qui trouve ou ne trouve pas une instruction à appliquer dans un état donné. Nous pourrions alors lui faire ramener une valeur spécifique, par exemple **NIL**, que nous testerions dans la fonction **TURING** pour lancer l'appel récursif si une instruction a été trouvée, et sortir de la fonction en ramenant la bande obtenue sinon.

Esthétiquement, cette solution ne semble pas belle : pourquoi ne pas arrêter tout de suite dans la fonction **FIND_INSTRUCTION**, dès que nous avons trouvé qu'il n'y a plus rien à faire ? La réponse à cette question est simple : parce que toute fonction retourne une valeur vers l'endroit où elle a été appelée. Nous ne connaissons – pour l'instant – aucun moyen d'interrompre ce *flux de contrôle*. Jusqu'à maintenant, chaque fonction appelée calculait une valeur qu'elle retournait vers la fonction appelante, ou vers le *top-level* si elle a été appelée du premier niveau.

Heureusement, LISP a prévu ce problème et dispose d'une fonction permettant de faire des *échappements* non locaux. Afin de rendre la notion d'echappement très claire, regardez les appels de fonctions représentés graphiquement ci-dessous :



Dans cet *arbre d'appels* nous avons 6 fonctions nommées **F1**, **F2**, **F3**, **F4**, **F5** et **F6**. Les flèches vers le bas représentent des appels de fonctions et les flèches vers le haut représentent les retours de fonctions. Le retour d'une fonction suit le même chemin que l'appel d'une fonction. La fonction **F1** appelle les fonctions **F2** et **F6**. La fonction **F2** appelle la fonction **F3**, qui à son tour appelle les fonctions **F4** et **F5**. Les fonctions **F4**, **F5** et **F6** n'appellent pas d'autres fonctions utilisateurs. Ci-dessous six fonctions simples (n'essayez pas d'y mettre trop de sens : il n'y en a pas) correspondant à un tel schéma :

```
(DEFUN F1 (X)
       (IF (< X 0) (- (F6 X) 10)
              (LET ((AUX (F2 X)))
                   (IF AUX (- AUX 10) 'NO-GOOD))))

(DEFUN F6 (X) (* X X))

(DEFUN F2 (X) (LET ((AUX (F3 X)))
              (IF AUX (1+ AUX) NIL)))

(DEFUN F3 (X) (IF (> X 100) (F4 X) (F5 X)))

(DEFUN F4 (X) (IF (> X 1000) NIL (SQRT X)))

(DEFUN F5 (X) (+ X X))
```

Si l'on appelle la fonction **F1** avec un nombre supérieur à 1000 elle retourne le symbole **NO-GOOD**. Dans tous les autres cas elle retourne une valeur numérique. Si nous examinons attentivement l'ensemble des fonctions, il devient très vite clair que l'endroit déterminant si le symbole **NO-GOOD** est retourné ou non se trouve dans l'expression :

(IF (> X 1000) NIL (SQRT X))

de la fonction **F4**. La valeur **NIL** calculée dans cette expression sera retournée vers la fonction **F3**, qui la retourne immédiatement vers la fonction **F2**. Pour l'instant, nous ne faisons que remonter dans l'arbre des appels. La fonction **F2** teste la valeur retournée : si elle est égale à **NIL** elle la retourne telle quelle à la fonction **F1** sinon elle retourne le résultat de l'incrémentation de 1 et de la valeur reçue. Finalement, la fonction **F1** reteste la valeur reçue, retourne la différence entre 10 et cette valeur ou l'atome **NO-GOOD** suivant que la valeur reçue était différente ou égale à **NIL**.

En résumé : une fois la valeur **NIL** calculée dans la fonction **F4**, elle n'est que transportée vers les fonctions appelantes successives, jusqu'à ce que finalement on arrive dans la fonction principale **F1** qui ne fait qu'échanger cette valeur **NIL** contre la valeur **NO-GOOD**. Tout ce transport à travers les diverses fonctions appelantes semble superflu. Nous sommes obligés de le faire puisque c'est comme cela que les appels de fonctions se comportent : jusqu'à maintenant, nos fonctions retournent *toujours* à la fonction appelante.

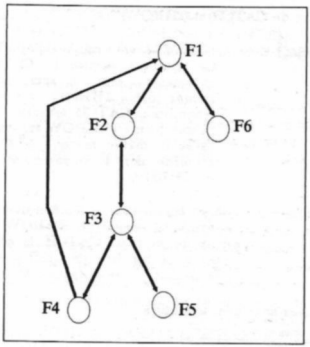
Dans des cas de ce style, des *échappements* non-locaux sont très utiles : ils permettent d'éviter des longues remontées dans les arbres d'appels et du coup, ils permettent d'éviter des tests.

Un échappement se construit avec les deux fonctions **CATCH** et **THROW**. La fonction **CATCH** indique *vers où* il faut échapper, la fonction **THROW** indique *quand* il faut échapper.[3] Avant de donner une description plus formelle de ces deux fonctions, examinons une réécriture de l'ensemble des fonctions **F1** à **F6** utilisant un échappement. Voici le nouveau programme :

```
(DEFUN F1 (X)
       (IF (< X 0) (- (F6 X) 10)
             (CATCH 'FINI (- (F2 X) 10))))

(DEFUN F6 (X) (* X X))

(DEFUN F2 (X) (1+ (F3 X)))

(DEFUN F3 (X) (IF (> X 100) (F4 X) (F5 X)))

(DEFUN F4 (X) (IF (> X 1000) (THROW 'FINI 'NO-GOOD) (SQRT X)))

(DEFUN F5 (X) (+ X X))
```

Ci-dessous, nous redonnons l'arbre d'appels et de retours pour ces fonctions. Il y a un nouveau retour de fonctions. Plus précisément, nous y avons ajouté un chemin de retour allant directement de la fonction **F4** vers la fonction **F1**. C'est un retour qui ne suit pas du tout le chemin des appels. C'est pour cela qu'un tel retour est appelé un *échappement non local*. De tels chemins sont construits avec les fonctions **CATCH** et **THROW**.

3. *Catch* et *throw* sont deux verbes anglais pour *attrape* et *lance*. *Catch* indique qui doit attraper la balle que *throw* lance. En Le_LISP **CATCH** se dit **TAG** et **THROW** se dit **EXIT**.



Examinons la nouvelle écriture des fonctions **F1** à **F6**. Les fonctions **F3**, **F5** et **F6** n'ont pas été modifiées. Dans la fonction **F2** nous avons éliminé le test sur le résultat de la fonction **F3** et dans la fonction **F1** nous avons également éliminé le test. Par contre, dans la fonction **F1** nous avons introduit l'appel de **CATCH** suivant :

```
(CATCH 'FINI (- (F2 X) 10))
```

CATCH est une fonction à au moins deux arguments : un *nom-d'échappement* et au moins une expression. Si LISP rencontre un appel de **CATCH**, LISP mémorise le *nom-d'échappement*, ici c'est le symbole **FINI**, et évalue en séquence les expressions suivantes. Si pendant l'évaluation de ces expressions LISP ne rencontre aucun appel d'une fonction **THROW** avec ce même *nom-d'échappement*, **CATCH** se comporte comme la fonction **PROGN**, c'est-à-dire que **CATCH** évalue une expression après l'autre et retourne la valeur de la dernière expression.

Par contre, si LISP rencontre pendant l'évaluation des expressions un appel de la fonction **THROW** avec le même *nom-d'échappement* comme premier argument, il évalue le deuxième argument du **THROW** et retourne la valeur au lieu directement suivant l'appel du **CATCH** correspondant.

Dans notre exemple, la fonction **F4** retourne l'atome **NO-GOOD** (si **X** est supérieur à 1000) au lieu marqué ci-dessous par un Δ

```
(DEFUN F1 (X)
       (IF (< X 0) (- (F6 X) 10)
                   (CATCH 'FINI (- (F2 X) 10)) Δ ))
```

du coup, circonvenant l'ensemble des fonctions intermédiaires. Le chemin de retour allant directement de **F4** vers **F1** dans le graphique ci-dessus correspond à l'évaluation du **THROW**.

Voici finalement la définition de **CATCH** et **THROW** :

(**CATCH** *tag* $expr_1$ ... $expr_n$) → d'abord *tag* est évalué. *tag* doit livrer un symbole. Ce symbole *nomme* le **CATCH**. Ensuite les expressions $expr_1$ à $expr_n$ sont évaluées. Si aucune forme **THROW** du même nom *tag* n'est rencontrée, **CATCH** retourne la valeur de $expr_n$. Si une forme (**THROW** *tag expr*) est rencontrée (avec le même *tag* que le **CATCH**), **CATCH** retourne alors la valeur du deuxième argument du **THROW**.

(**THROW** *tag expr*) → d'abord *tag* est évalué. *tag* doit livrer un symbole. Ce symbole *nomme* le **THROW**. retourne la valeur de *expr* au **CATCH** le plus récent du même nom *tag*. Ensuite LISP sort du **CATCH** correspondant en retournant cette même valeur.

Ces échappements non locaux sont très, très utiles.

16.2 RETOUR VERS LES MACHINES DE TURING

Utilisons alors ces deux fonctions **CATCH** et **THROW** pour programmer l'arrêt de notre programme simulant une machine de Turing. Ce qui donne :

```
(DEFUN TURING (BANDE PROGRAMME ETAT)
 (OR (MEMQ T BANDE)(SETQ BANDE (CONS T BANDE)))
 (CATCH 'EXIT
      (LET
         ((INSTR (FIND_INSTRUCTION (CARACTERE_ACTUEL BANDE)
                        BANDE PROGRAMME ETAT)))
         (TURING (EXECUTE INSTR BANDE)
              PROGRAMME
              (CAR (LAST INSTR))))))
```

Ainsi, dès que nous rencontrerons un appel (**THROW 'EXIT** *expr*) quelque part dans le programme, nous sortirons du **CATCH** de la fonction **TURING** et donc également de cette fonction.

Bien entendu, comme nous l'avons dit plus haut, ce **THROW** doit être dans la fonction **FIND_INSTRUCTION**, ou dans une fonction appelée par cette fonction. Mais écrivons d'abord la fonction **CARACTERE_ACTUEL**. Cette fonction doit ramener le caractère pointé actuellement. Sachant que la bande est de la forme :

$$c_1\ c_2 \ldots c_{i-1}\ \mathbf{T}\ c_i \ldots c_n$$

nous savons que le caractère c_i est le caractère pointé. Il suffit donc de chercher le **T** dans la bande et de retourner l'élément immédiatement suivant. Voici cette petite fonction :

```
(DEFUN CARACTERE_ACTUEL (BANDE)
  (LET ((CARACTERE (MEMBER T BANDE)))
    (WHEN (CDR CARACTERE) (CADR CARACTERE))))
```

Cette fonction au moins est simple. Le test sert dans le cas où il n'y aurait pas de caractère disponible. Etant donné que nous ne représentons pas la totalité de la bande (nous ne le pouvons pas à cause de l'infinité de cette bande), il peut arriver que la tête de lecture/écriture se trouve toute à droite du segment de la bande représenté dans la liste **BANDE**. Dans ce cas, la valeur ramenée par **CARACTERE_ACTUEL** serait **NIL**. Dans tous les autres cas, ce sera le caractère à droite du **T**.

Attaquons alors la deuxième fonction qui reste à écrire. C'est la fonction **FIND_INSTRUCTION**. Cette fonction reçoit le caractère courant, le programme et la bande. Elle commence par vérifier si un caractère est disponible et si ce n'est pas le cas, elle ajoute un **B** à droite du **T** dans la bande et prend ce **B** comme caractère courant. Ensuite, elle doit parcourir l'ensemble des quintuplets qui composent le programme pour y chercher un quintuplet qui commence par l'état actuel et qui demande de lire ce caractère actuel :

```
(DEFUN FIND_INSTRUCTION (CARACTERE BANDE PROGRAMME ETAT)
  (WHEN (NULL CARACTERE)
    ; la tête de lecture est tout à droite de la représentation de
    ; la bande, on insère un caractère blanc 'B', et on ramène
    ; ce caractère
    (NCONC BANDE '(B)) (SETQ CARACTERE 'B))
  ; ensuite on cherche l'instruction
  (FIND_INSTR PROGRAMME BANDE CARACTERE ETAT))
```

Dans cette fonction nous avons délégué la tâche principale, la recherche du *bon* quintuplet, à la fonction **FIND_INSTR** que voici :

```
(DEFUN FIND_INSTR (PROGR BANDE CARACTERE ETAT) (COND
  ((NULL PROGR)
    ; c'est la fin du programme: on se trouve dans un état
    ; qui n'est état actuel dans aucune instruction
    ; ou on a un caractère qui n'est pas prévu pour l'état actuel
    (THROW 'EXIT BANDE))
  ((AND (EQ (CAAR PROGR) ETAT) (EQ (CADAR PROGR) CARACTERE))
    ; on l'a trouvé
    (CDDR (CAR PROGR)))
  (T (FIND_INSTR (CDR PROGR) BANDE CARACTERE ETAT))))
```

Si nous n'avons pas trouvé une instruction qui peut s'appliquer à l'état actuel de la machine, nous sortons avec l'instruction :

```
(THROW 'EXIT BANDE)
```

qui nous fera sortir non seulement de la fonction **FIND_INSTR**, mais également des fonctions **FIND_INSTRUCTION** et **TURING** !

Tant que nous avons encore des quintuplets candidats au lancement, nous continuons la

recherche, et si nous avons trouvé un quintuplet, nous le retournons mais sans les deux premiers éléments. Ces deux éléments, l'état actuel et le caractère actuel, ne sont plus nécessaires dans la suite, c'est-à-dire : pendant l'exécution du quintuplet trouvé.

Ne reste finalement que la fonction simulant la CPU de la machine de Turing. Cette fonction, la fonction **EXECUTE**, devra écrire à l'endroit du caractère lu le caractère à écrire spécifié dans le quintuplet, et déplacer la tête de lecture/écriture une position vers la gauche ou une position vers la droite, dépendant du **L** ou **R** trouvé dans le quintuplet à exécuter. Ci-dessous finalement cette fonction d'exécution :

```
(DEFUN EXECUTE (INSTR BANDE)
   ; INSTR n'est plus que de la forme (S2 M qj)
   (LABELS ((MOVE-RIGHT (X INSTR)(COND
                ((EQ (CAR X) 'T)
                   (CONS (CAR INSTR)(CONS 'T (CDDR X))))
                (T (CONS (CAR X) (MOVE-RIGHT (CDR X) INSTR)))))

            (MOVE-LEFT (X INSTR)(COND
                ((EQ (CADR X) 'T)
                   (CONS 'T (CONS (CAR X)
                                  (CONS (CAR INSTR)(CDDDR X)))))
                (T (CONS (CAR X) (MOVE-LEFT (CDR X) INSTR))))))

      (COND
         ; mouvement vers la droite
         ((EQ (CADR INSTR) 'R) (MOVE-RIGHT BANDE INSTR))
         ; mouvement vers la gauche
         ((EQ (CADR INSTR) 'L)
            (IF (EQ (CAR BANDE) T)
               ; cas spécial: on se trouve au début de la bande
               ; il faut alors coller un B en tête
               (CONS T (CONS 'B (CONS (CAR INSTR) (CDDR BANDE))))
               ; sinon, comme plus haut
               (MOVE-LEFT BANDE INSTR))))))
```

Nous avons fini notre simulateur des machines de Turing. Regardons quelques appels exemples, pour voir si ça marche :

```
? (TURING '(T 0 0 1 1 B)
      '((Q1 B B R Q1)
        (Q1 0 0 R Q1)
        (Q1 1 1 R Q2)
        (Q2 1 1 R Q2)
        (Q2 0 1 R Q3)
        (Q2 B 1 R Q3))
      'Q1)

=(0 0 1 1 1 T B)
```

et la même chose avec retour de la tête de lecture/écriture

```
? (TURING '(T 0 0 1 1 B)
          '((Q1 B B R Q1)
            (Q1 0 0 R Q1)
            (Q1 1 1 R Q2)
            (Q2 1 1 R Q2)
            (Q2 0 1 L Q3)
            (Q2 B 1 L Q3)
            (Q3 1 1 L Q3)
            (Q3 0 0 R Q4)
            (Q3 B B R Q4))
          'Q1)

=(0 0 T 1 1 1)
```

Il reste un problème : le programme **TURING** que nous venons d'écrire est extrêmement inefficace. Tout ce que peut faire une instruction d'une telle machine c'est de :

1. lire le caractère de la case sur laquelle pointe la tête de lecture/écriture
2. écrire un caractère dans la case sur laquelle pointe la tête de lecture/écriture
3. déplacer la tête de lecture d'une case vers la gauche ou d'une case vers la droite.

C'est tout ! Ce que nous faisons dans l'interprétation de chaque instruction c'est de :

1. chercher l'endroit de la tête de lecture pour savoir sur quel caractère elle pointe,
2. chercher à l'intérieur du programme le quintuplet qui s'applique à cette situation,
3. rebalayer le bande pour retrouver la tête de lecture afin de pouvoir réécrire le contenu de la case pointée,
4. écrire le nouveau caractère, en copiant toute la partie précédente la tête de lecture.
5. retourner la nouvelle copie de la bande, pour recommencer le processus avec elle.

Il doit y avoir une possibilité de simuler plus précisément le comportement d'une machine de Turing. Une telle machine n'a aucun besoin de chercher la tête de lecture : elle *sait* toujours où celle-ci se trouve, sur quelle case elle pointe.

Nous pouvons faire la même chose si nous coupons la bande en deux parties : une bande infinie à gauche et une autre infinie à droite. Les deux ensembles nous donnent toujours une bande infinie à gauche et à droite. Le point de coupure serait, bien entendu, la case pointée par la tête de lecture/écriture. Ainsi, par exemple, une partie contiendrait en tête la case pointée et le reste serait la partie de l'extrait de la bande représenté qui se trouve à droite de cette tête de lecture. La partie gauche contiendrait donc cette partie de l'extrait de la bande représenté qui se trouve à gauche de la tête de lecture/écriture.

Avec une telle représentation :

$$\textit{partie gauche} = c_1\ c_2 \ldots c_{i-1}$$
$$\textit{partie droite} = c_i\ c_{i+1} \ldots c_n$$

le premier caractère de la *partie droite* correspondrait à la case pointée par la tête de lecture/écriture, les caractères c_{i+1} à c_n correspondraient aux caractères à droite de la tête de lecture,[4] et toute la partie gauche correspondrait à la suite de caractères qui précédent la tête de lecture. Avec cette représentation nous pourrions économiser :

1. la fonction **CARACTERE_ACTUEL**, puisque le caractère actuel serait disponible en permanence en tête de la liste correspondant à la *partie-droite*, et

2. nous n'aurions aucun besoin de chercher répétitivement là où se trouve la tête de lecture : la tête de lecture pointe, à tout instant de l'exécution d'un programme, sur le caractère en tête de la liste *partie-droite*.

3. De plus, un mouvement de la tête à gauche ou à droite n'impliquerait plus une copie de la bande, mais juste un transfert du premier caractère de la partie droite vers la queue de la partie gauche (si c'est un mouvement à gauche) ou un transfert du dernier caractère de la partie gauche en tête de la partie droite (si c'est un mouvement à droite).

Cette nouvelle conception de la bande peut encore être améliorée – en vue de la programmation du simulateur – si nous inversons la *partie gauche*. Pourquoi ? Tout simplement, puisque LISP sait très bien accéder aux têtes des listes (avec la fonction **CAR**) mais beaucoup plus mal à la queue d'une liste. Si nous inversons la partie gauche, le premier élément de la partie gauche contiendra à tout instant le caractère à gauche de la tête de lecture et le premier caractère de la partie droite contiendra toujours le caractère pointé. Avec ce schéma, la lecture correspond à une instruction **CAR** de la partie gauche, le mouvement à droite sera un **CDR** sur la partie droite et un **CONS** sur la partie gauche, et, finalement, le mouvement vers la gauche sera un **CDR** sur la partie gauche et un **CONS** sur la partie droite. Nous n'utiliserons que des fonctions standard de base de LISP.

En réalité, cette représentation n'est rien d'autre que la représentation d'une machine de Turing dans une machine finie à deux piles.[5]

Afin de programmer cette deuxième version nous allons mettre les parties de la bande qui correspondent aux parties gauche et droite dans des variables globales nommées respectivement **PARTIE_GAUCHE** et **PARTIE_DROITE**.

En CommonLISP, il est de bon style de *déclarer* les variables globales. Ceci pour deux raisons :

4. De temps à autre nous écrivons seulement *tête de lecture*. C'est une commodité d'écriture et veut, naturellement, dire *tête de lecture/écriture*.

5. Si vous ne savez pas ce qu'est une machine finie, lisez le livre de Minsky sur les *Finite and Infinite Machines* mentionné au début du chapitre.

1. de cette façon le compilateur LISP *sait* qu'il faut réserver de la place au niveau global, i.e.: accessible à toutes les fonctions,

2. dans une déclaration de variable globale on peut ajouter un commentaire qui indique le rôle de cette variable. Ce commentaire est accessible à tout instant par la fonction **DOCUMENTATION**.

Une déclaration de variable globale se fait avec la fonction **DEFVAR**, pour *define variable*. Cette fonction a trois arguments : le nom de la variable globale, sa valeur initiale et une chaîne de caractères qui contient le commentaire associé à cette variable. Nous déclarons alors nos deux variables :

```
(DEFVAR PARTIE_GAUCHE NIL
        "contiendra à tout instant la liste qui représente l'inverse de
        la bande qui se trouve à gauche de la tête de lecture Ainsi un
        mouvement vers la gauche n'implique qu'un enlèvement du premier
        élément de cette liste et son rajout en tête de PARTIE_DROITE")

(DEFVAR PARTIE_DROITE NIL
        "contient à tout instant la liste qui représente la bande se
        trouvant à droite de la tête de lecture. Ainsi un
        mouvement vers la droite n'implique qu'un enlèvement du premier
        élément de cette liste et son rajout en tête de PARTIE_GAUCHE")
```

Voici alors le programme **TURING** modifié de la manière indiquée. Essayez de le comprendre, implémentez-le sur votre machine et testez-le. Peut-être même que vous trouverez des améliorations supplémentaires à faire.

```
(DEFUN TURING (BANDE PROGRAMME ETAT)
    (OR (MEMQ T BANDE)(SETQ BANDE (CONS T BANDE)))
    (SETQ PARTIE_GAUCHE (DIVISE_LA_BANDE_EN_DEUX BANDE)
          PARTIE_DROITE (CDR (MEMQ 'T BANDE)))
    (TURING_AUX PROGRAMME ETAT))

(DEFUN DIVISE_LA_BANDE_EN_DEUX (L &OPTIONAL PG)(COND
    ((NULL L) NIL)
    ((EQ (CAR L) T) PG)
    (T (DIVISE_LA_BANDE_EN_DEUX (CDR L) (CONS (CAR L) PG)))))

(DEFUN TURING_AUX (PROGRAMME ETAT)
    (CATCH 'EXIT
      (LET ((INSTR
             (FIND_INSTRUCTION (CAR PARTIE_DROITE)
             ETAT PROGRAMME)))
        (EXECUTE INSTR)
        (TURING_AUX PROGRAMME (CAR (LAST INSTR))))))
```

```
        ; la bande est representée par deux listes:
        ; partie_droite: (ci ci+ 1 ... cn) , la partie à droite de la tête de lecture
        ; partie_gauche: (ci-1 ci-2 ... c1) , la partie à gauche de la tête de lecture
        ; en réalité, c'est la représentation d'une machine de Turing dans une
        ; machine finie à deux piles

        (DEFUN FIND_INSTRUCTION (CARACTERE ETAT PROGRAMME)
            (WHEN (NULL CARACTERE)
              ; la tête de lecture est tout à droite de la représentation de
              ; la bande, on insère un caractère blanc 'B', et on ramène
              ; ce caractère
              (SETQ PARTIE_DROITE '(B))(SETQ CARACTERE 'B))
            ; ensuite on cherche l'instruction
            (FIND_INSTR PROGRAMME ETAT CARACTERE))

; les instructions sont de la forme: (qi S1 S2 M qj)
; avec qi correspondant à l'état actuel, S1 au caractère actuel, S2 au caractère à écrire
; M au mouvement (soit R soit L), et qj à l'état suivant

(DEFUN FIND_INSTR (PROGR ETAT CARACTERE) (COND
    ((NULL PROGR)
        ; c'est la fin du programme: on se trouve dans un état
        ; qui n'est l'état actuel d'aucune instruction
        (THROW 'EXIT
          (APPEND (REVERSE PARTIE_GAUCHE) (CONS T PARTIE_DROITE))))
    ((AND (EQ (CAAR PROGR) ETAT) (EQ (CADAR PROGR) CARACTERE))
        ; on l'a trouvé
        (CDDR (CAR PROGR)))
    (T (FIND_INSTR (CDR PROGR) ETAT CARACTERE))))

; reste l'exécution.
(DEFUN EXECUTE (INSTR)
    ; instr n'est plus que de la forme (S2 M qj)
    (COND
        ; mouvement vers la droite
        ((EQ (CADR INSTR) 'R)
            (SETQ PARTIE_GAUCHE (CONS (CAR INSTR) PARTIE_GAUCHE))
            (NEXTL PARTIE_DROITE))
        ; mouvement vers la gauche
        ((EQ (CADR INSTR) 'L)
            (IF (NULL PARTIE_GAUCHE)
                ; cas spécial: on se trouve au début de la bande
                ; il faut alors coller un B en tête
                (SETQ PARTIE_DROITE (CONS 'B
                        (CONS (CAR INSTR)
                          (CDR PARTIE_DROITE))))
                ; sinon, comme plus haut
                (SETQ PARTIE_DROITE
                    (CONS (NEXTL PARTIE_GAUCHE)
                          (CONS (CAR INSTR) (CDR PARTIE_DROITE))))))))
```

Avez-vous vu comment le programme s'est simplifié grâce à ce changement de représentation ?

17. LES MACRO-FONCTIONS

Pendant la construction de grands programmes, il arrive souvent que l'on construise des fonctions relativement simples juste pour augmenter la lisibilité des programmes. Ainsi, si vous cherchez très souvent dans un programme le quatrième élément d'une liste, au lieu d'écrire à chaque fois :

(CADR (CDDR *la-liste*))

vous allez probablement définir une fonction nommée **4-IEME**, définie comme suit :

(DEFUN 4-IEME (L) (CADR (CDDR L)))

et remplacer toutes les occurrences des instructions ci-dessus par :

(4-IEME *la-liste*)

rendant ainsi le programme plus lisible.

Le problème avec cette manière de programmer est que l'exécution du programme devient de plus en plus lente : chaque appel d'une fonction utilisateur lance les processus de sauvegarde, liaison et restauration des valeurs des variables. A cause de cette dichotomie entre la lisibilité et le temps d'exécution d'un programme, LISP permet – en plus des fonctions **DE** et **DF** – un type de fonctions supplémentaire : ce sont les *macro-fonctions*, ou – plus simplement – les *macros*. Ce chapitre introduit un aspect de leur utilisation.

Les macro-fonctions se définissent avec la fonction **DEFMACRO** (pour *define macro*-fonction). Les macro-fonctions ont, syntaxiquement, la même forme que toutes les autres définitions de fonctions : un nom, une liste de variables et un corps :

(DEFMACRO *nom liste-de-variables corps*) → *nom*
en plus : définit une fonction de type *macro*.

La particularité des macro-fonctions est :

1. à l'appel, les variables sont liées aux arguments correspondants *non évalués*. En Common-LISP, où nous n'avons pas la fonction **DF**, cette fonction est la seule permettant de définir des fonctions qui n'évaluent pas leurs arguments.

2. ensuite le corps est évalué et *le résultat de cette évaluation est encore une fois évalué.* L'évaluation d'une macro se fait donc en deux temps.

L'originalité des macro-fonctions réside dans ce mécanisme de double évaluation, ce qui implique que le résultat de la première évaluation doit être une *forme* LISP,, c'est-à-dire : un bout de code LISP exécutable.

Afin d'illustrer ce dernier point, commençons par un exemple très simple : la macro-fonction **EXEMPLE1** qui répond à chaque appel par la valeur **3**. La voici :

```
(DEFMACRO EXEMPLE1 () '(1+ 2))
```

A la première évaluation, cette macro, sans arguments, retourne la valeur de l'expression constituant son corps :

```
(QUOTE (1+ 2))
```

donc la liste **(1+ 2)**. Cette liste ou, plus précisément, cette forme LISP est alors évaluée, dans une deuxième étape, livrant le résultat numérique **3** pour tout appel de la macro.

Mais revenons vers notre fonction **4-IEME** et définissons une macro qui résout le problème. Voici la première version :

```
(DEFMACRO 4-IEME (L) (LIST 'CADR (LIST 'CDDR L)))
```

Le corps de cette macro construit une expression à évaluer, qui exprime le calcul du **CADR** du **CDDR** de l'argument donné. Evidemment, en utilisant le macro-caractère *backquote* (‘), nous pouvons écrire la macro plus simplement comme :

```
(DEFMACRO 4-IEME (L) '(CADR (CDDR ,L)))
```

Si nous appelons cette macro comme suit :

```
(4-IEME '(A B C D E F))
```

le résultat de la première évaluation de cet appel est l'expression :

```
(CADR (CDDR (QUOTE (A B C D E F))))
```

et l'évaluation supplémentaire de cette expression livre l'atome **D**, qui sera alors le résultat de l'appel de notre macro **4-IEME**.

Naturellement, on peut construire des macro-fonctions récursives. Par exemple, la macro **MCONS** ci-dessous s'appelle récursivement jusqu'à ce qu'elle aboutisse à la fin de la liste passée en argument :

```
(DEFMACRO MCONS L
   (IF (NULL (CDR L)) (CAR L)
      '(CONS ,(CAR L) (MCONS ,@(CDR L)))))
```

Remarquez la liste de variables dégénérée dans cette définition de macro. Ce n'est pas une liste du tout, c'est juste une variable. Nous avons déjà rencontré des listes de variables similaires en Le_LISP. En CommonLISP, la fonction **DEFMACRO** est la seule forme qui définit des fonctions qui permettent des listes de variables *destructurant*. Ce type de liste de variables est nommé déstructurant, parce que la liaison des variables est guidée par la structure de la liste de variables. Autrement dit, la liaison se fait comme un filtrage entre l'arbre qui représente la liste des variables et celui qui représente la suite d'arguments.

Si, comme c'est le cas ici, la liste de variables est réduite à une simple variable, cette variable sera alors liée à la liste de tous les arguments de l'appel de la macro. Par exemple, si nous appelons **MCONS** comme :

```
(MCONS A B C D E F)
```

la variable **L** sera liée à la liste :

```
(A B C D E F)
```

Puisque **MCONS** est une macro, aucun des arguments n'est évalué. Naturellement, nous aurions pu définir **MCONS** aussi bien comme

```
(DEFMACRO MCONS (&REST L) ... )[1]
```

Ci-dessous nous examinerons un exemple supplémentaire de telles listes de variables déstructurantes. Considérons une définition de macro comme :

```
(DEFMACRO FOO ((X Y Z) ((FTN1 ARG1)(FTN2 ARG2) . RESTE) THE-LAST) ...
```

et l'appel :

```
(FOO (1 2 3) (((4 VIER FOUR QUATRE) 5)(6 7) 8 9 10) 11)
```

Cet appel crée les liaisons suivantes :

variable	*value*
X	1
Y	2
Z	3
FTN1	(4 VIER FOUR QUATRE)
ARG1	5

1. Quelques implémentations de CommonLISP n'acceptent que cette dernière forme. Avant de construire de longs programmes pleins de macro-fonctions, vérifiez donc d'abord les formes de listes de variables acceptées par votre système CommonLISP.

```
FTN2        6
ARG2        7
RESTE       (8 9 10)
THE-LAST    11
```

Ici également, la paire pointée aurait pu être exprimée avec le spécifieur **&REST**, comme en :

(DEFMACRO FOO ((X Y Z) ((FTN1 ARG1)(FTN2 ARG2) &REST RESTE) THE-LAST) ...

Si nous appelons **FOO** comme ci-dessous :

(FOO (1 2) ((4 VIER FOUR QUATRE) SIX) 10)

LISP donne un message d'erreur puisque cet appel ne filtre pas la liste de variables :

1. Il manque un argument pour le paramètre **Z**.

2. Les arguments **FOUR** et **QUATRE** n'ont pas de paramètres qui leur correspondent : l'argument **4** correspond au paramètre **FTN1**, l'argument **VIER** correspond au paramètre **ARG1**. Il n'y pas de place prévue pour des arguments supplémentaires.

3. Finalement, à la place de la sous-liste des paramètres **(FTN2 ARG2)** on trouve l'atome **SIX**. Même si la valeur de **SIX** était une liste à deux éléments, ça ne marcherait pas : LISP demande que la structure même de l'appel filtre la liste des paramètres de la macro correspondant. D'ailleurs, la liste des paramètres est souvent appelée *une lambda-liste*.

Rappelons qu'en Le_LISP *toutes* les lambda-listes sont déstructurantes; en CommonLISP, seulement les lambda-listes de **DEFMACRO**.

Mais revenons vers notre fonction **MCONS** :

```
(DEFMACRO MCONS L
  (IF (NULL (CDR L)) (CAR L)
    `(CONS ,(CAR L) (MCONS ,@ (CDR L)))))
```

Notons la forme de l'appel récursif. Si nous avions écrit :

(MCONS ,(CDR L))

nous aurions écrit un appel où **MCONS** ne recevrait qu'un seul argument, la liste qui contient tous les arguments sauf le premier. Bien entendu, ça ne livrerait pas le bon résultat. Ce que nous avons fait, c'est de construire une copie de l'appel initial de **MCONS** où nous avons enlevé le premier argument.

Voici une trace d'un appel de **MCONS** :

```
? (MCONS 1 2 3 4 5 NIL)                          ; l'appel initial ;
  ---> MCONS (1 2 3 4 5 NIL)                     ; la trace de cet appel initial ;
  <--- MCONS (CONS 1 (MCONS 2 3 4 5 NIL))        ; son résultat ;
  ---> MCONS (2 3 4 5 NIL)                       ; le premier appel récursif ;
  <--- MCONS (CONS 2 (MCONS 3 4 5 NIL))          ; son résultat
  ---> MCONS (3 4 5 NIL)
  <--- MCONS (CONS 3 (MCONS 4 5 NIL))
  ---> MCONS (4 5 NIL)
  <--- MCONS (CONS 4 (MCONS 5 NIL))
  ---> MCONS (5 NIL)
  <--- MCONS (CONS 5 (MCONS NIL))
  ---> MCONS (NIL)
  <--- MCONS NIL
=(1 2 3 4 5)
```

A chaque appel, la variable est liée à la liste des arguments *non-évalués* et son évaluation s'exécute en deux temps : chaque appel récursif est effectué *après* la sortie de l'appel précédent. Ce n'est pas réellement une exécution récursive !

Bien que nous ayons introduit les macro-fonctions en argumentant sur les temps d'exécution, nous ne pouvons pas réellement prétendre que nous avons amélioré quoi que ce soit en les introduisant : tout au contraire, les macro-fonctions sont encore plus lentes que les fonctions utilisateurs normales.

Pourtant, les macro-fonctions *sont* plus rapides ! Voici pourquoi : jusqu'à maintenant nous avons caché la caractéristique la plus importante des macro-fonctions : elles modifient les fonctions à l'intérieur desquelles elles sont appelées. *Le résultat de la première évaluation remplace l'appel de la macro-fonction. Ainsi, les appels de macro-fonctions se détruisent eux-mêmes, assurant par là l'élimination de ces appels.*

Revenons vers notre macro **4-IEME** :

```
(DEFMACRO 4-IEME (L) '(CADR (CDDR ,L)))
```

Afin de voir clairement l'effet de cette macro, construisons une fonction **FOO** qui calcule la somme du premier et du quatrième élément de sa liste argument :

```
(DEFUN FOO (&REST LISTE)(+ (CAR LISTE) (4-IEME LISTE)))
```

Quand nous appelons cette fonction, par exemple par :

```
(FOO 1 2 3 4 5 6 7)
```

le résultat sera, bien évidemment, la valeur numérique **5**. Mais, si nous regardons maintenant la fonction, elle a l'allure suivante :

```
(DEFUN FOO (&REST LISTE)
       (+ (CAR LISTE) (CADR (CDDR LISTE))))
```

Tous les appels ultérieurs de la fonction **FOO** n'impliqueront plus l'appel de la fonction **4-IEME**,

puisque le premier appel a d'abord été remplacé par le résultat de l'évaluation du corps de la macro-fonction.

La fonction **NEXTL** que nous avons définie (pour Le_LISP) dans les exercices du chapitre précédent, peut maintenant être définie en tant que macro-fonction comme suit :

```
(DEFMACRO NEXTL (VAR)
  (LET ((VAL (GENSYM)))
    `(LET ((,VAL ,VAR))
       (SETQ ,VAR (CDR ,VAL)) (CAR ,VAL))))
```

Cette définition a besoin de quelques explications : tout d'abord, nous y avons utilisé une nouvelle fonction : **GENSYM**. C'est une fonction qui *génère* de nouveaux *symboles*. Nous l'utilisons chaque fois que nous avons besoin d'un symbole dont nous sommes sûrs qu'il ne sera pas utilisé ailleurs. Voici quelques exemples d'appels :

```
?  (GENSYM)          ; un appel
=  G117          ; un premier symbole
?  (GENSYM)          ; un autre appel
=  G118          ; un symbole différent
?  (GENSYM)          ; encore un appel
=  G119       ; et encore un autre symbole
```

C'est définitivement une fonction très utile pour générer des nouveaux symboles qui n'existent pas jusqu'à maintenant. Nous utilisons ici, dans la définition de **NEXTL**, la fonction **GENSYM**, puisque nous voulons créer une variable locale **VAL**, de laquelle nous voulons être absolument assurée qu'elle n'interfère avec aucune autre variable déjà existante.

Pour rendre ce point tout à fait clair, imaginons un instant que nous ayons écrit la fonction **NEXTL** de la manière suivante :

```
(DEFMACRO NEXTL-PRESQUE-CORRECT (VAR)
  `(LET ((VAL ,VAR))
     (SETQ ,VAR (CDR VAL)) (CAR VAL))))
```

L'unique différence entre cette définition et la définition précédente est l'absence de l'appel de **GENSYM**. Regardons comment ça fonctionne. Si la variable globale **X** contient à un certain instant la liste **(A B C)**, un appel possible peut être :

```
(NEXTL-PRESQUE-CORRECT X)
```

la variable **VAR** de **NEXTL-PRESQUE-CORRECT** sera liée à l'atome **X** tel quel. Une première évaluation du corps de la macro nous livre l'expression suivante :

```
(LET ((VAL X))
     (SETQ X (CDR VAL)) (CAR VAL))))
```

et l'évaluation supplémentaire donne temporairement à **VAL** la valeur de la variable **X**, c'est-à-dire : la liste **(A B C)**, puis elle affecte à **X** le **CDR** de cette liste, i.e.: la liste **(B C)**, et ramène

finalement le **CAR** de **VAL**, donc l'atome **A**. Tout marche pour le mieux dans le meilleur des mondes.

En supposant que la variable globale **VAL** contienne la liste **(1 2 3)**, donnons maintenant un appel plus problématique :

```
(NEXTL-PRESQUE-CORRECT VAL)
```

et faisons le même exercice que ci-dessus. Le résultat de la première évaluation sera l'expression :

```
(LET ((VAL VAL))
     (SETQ VAL (CDR VAL)) (CAR VAL)))
```

et voici ce qui se passe pendant la deuxième évaluation :

1. tout d'abord, la variable locale **VAL** prendra la valeur de la variable globale **VAL**, donc la liste **(1 2 3)**;

2. ensuite la variable locale **VAL** prendra comme nouvelle valeur le **CDR** de sa valeur, i.e.: la liste **(2 3)**;

3. ensuite, le **LET** ramènera la valeur de l'expression **(CAR VAL)**, ce qui sera l'atome **2** – ce qui n'est sûrement pas le premier élément de la variable globale **VAL**, donc ici réside la première erreur;

4. ensuite, puisqu'on sortira d'une lambda-expression, les anciennes valeurs des variables globales seront restituées, ce qui donne à la variable globale **VAL** son ancienne valeur, c'est-à-dire : la liste **(1 2 3)**. La fonction **NEXTL-PRESQUE-CORRECT** ne changeait en rien la valeur de la variable **VAL** donnée en argument. C'est donc la deuxième erreur.

Ce qui se passait ici, c'est un *clash* de variables : Nous avons donné à la variable locale de **NEXTL-PRESQUE-CORRECT** un nom qui peut masquer une variable globale. D'ailleurs, dans notre fonction, la seule variable globale qui peut être masquée est – naturellement – la variable globale du même nom, donc la variable globale **VAL** (pour tous les autres arguments la fonction **NEXTL-PRESQUE-CORRECT** est tout à fait correcte). Des problèmes de masquage de variables sont connus sous le nom du problème des *funargs*. Nous en verrons plus dans le deuxième volume de cette introduction.

Juste pour montrer qu'avec la fonction **NEXTL** définie plus haut de tels problèmes ne peuvent pas surgir, voici une trace de son appel avec le même argument :

```
(NEXTL VAL)
```

Le résultat de la première évaluation sera le **LET** suivant :

```
(LET ((G108 VAL))
     (SETQ VAL (CDR G108)) (CAR G108))))
```

Dans cette expression, un masquage (un clash) de variable ne peut pas se produire : la variable **G108** a été générée par un appel de **GENSYM**, et est donc garantie unique.

Avec cette définition de **NEXTL**, la fonction Le_LISP **PREVERSE** de la fin du chapitre précédent devient, après une exécution :

```
(DF PREVERSE (L RES)
    (LETN PREV-INTERNE ((LISTE (EVAL (CAR L))))
          (IF (NULL LISTE) (SET (CAR L) RES)
              (SETQ RES
                    (CONS
                      (LET ((G120 LISTE))
                           (SETQ LISTE (CDR G120))
                           (CAR G120))
                      RES))
              (SELF LISTE))))
```

Regardons une petite interaction avec LISP utilisant cette macro **NEXTL** :

```
?  (SETQ X '(A B C D))        ; donnons une valeur à X
=  (A B C D)
?  (NEXTL X)                          ; le premier appel
=  A                                        ; le résultat
?  X                          ; quelle est la valeur de X?
=  (B C D)                             ; bien, elle a changé
?  (NEXTL X)                             ; un autre appel
=  B
?  X
=  (C D)                      ; encore, X a changé de valeur
```

Reste à examiner la fonction **MACROEXPAND** qui prend en argument une expression qui peut contenir un appel d'une macro-fonction. S'il y en a, **MACROEXPAND** applique la macro à ses arguments et ramène le résultat de la *première* évaluation de l'appel de macro-fonction correspondant. C'est-à-dire que **MACROEXPAND** montre le code généré après l'expansion de la macro. Par exemple, l'appel :

```
(MACROEXPAND '(NEXTL X))
```

retourne :

```
(LET ((G826 X)) (SETQ X (CDR G826)) (CAR G826))
```

Chaque appel de **NEXTL** produit, nous l'avons vu, un bout de code similaire.

L'utilisation des macro-fonctions s'impose chaque fois que vous voulez écrire un programme bien lisible sans pour cela diminuer son efficacité. La première exécution d'une fonction contenant une macro destructive sera un peu plus lente que l'exécution d'une fonction normale.

Par contre, toutes les exécutions suivantes seront considérablement accélérées.

17.1 EXERCICES

1. Définissez les macro-fonctions **INCR** et **DECR** qui, similaires à la macro-fonction **NEXTL**, prennent en argument le nom d'une variable, et livrent comme résultat la valeur de cette variable plus ou moins 1. Ces deux macros ont comme effet de bord d'incrémenter ou de décrémenter la valeur de la variable donnée en argument.

2. Définissez la macro-fonction **SETNTH** définie par les exemples suivants :

   (SETNTH '(A B C D) 3 'G) → (A B G D)
   (SETNTH '(1 2 3 4) 4 5) → (1 2 3 5)

   et si la variable **X** est liée à la valeur **(1 2 3 4)**, après l'appel :

   (SETNTH X 1 'A)

   la valeur de **X** est la liste **(A 2 3 4)**.

3. Définissez la macro-fonction **SETQ** qui fait exactement la même chose que la fonction **SETQ** standard mais qui *trace* tous ses appels. Cette nouvelle fonction **SETQ** livrera alors pour l'appel exemple ci-dessous :

   (SETQ A 1 B '(A B C))

   la valeur **(A B C)**, affectera – bien entendu – la valeur **1** à la variable **A** et la valeur **(A B C)** à la variable **B**, et imprimera la trace suivante :

   A = 1
   B = (A B C)

18. LES DIVERSES FORMES DE REPETITION

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

18.1 APPEL DE FONCTION IMPLICITE (OU LE BRANCHEMENT)

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

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

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

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

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

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

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

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

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

1. Cet exemple est dû à Daniel Goossens

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

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

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

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

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

18.2 LES ARGUMENTS FONCTIONNELS

Mis à part les deux exemples d'appels de fonctions que nous venons de voir, tous ceux que nous avons rencontrés jusqu'à maintenant, sont des appels où le *nom* de cette fonction apparaît explicitement, comme par exemple dans l'appel **(FOO 1)**. Autrement dit, dans la plupart des programmes que nous avons vus, on peut déterminer les fonctions appelées par une simple lecture statique.

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

Supposons que nous voulions écrire une fonction pour trier une liste de nombre dans un ordre ascendant, du plus petit nombre vers le plus grand. Une manière de faire serait :

```
(DEFUN TRIE-UNE-LISTE (L)
   (INSERE (CAR L) (TRIE-UNE-LISTE (CDR L))))

(DEFUN INSERE (ELE L) (COND
   ((NULL L) (LIST ELE))
   ((< ELE (CAR L)) (CONS ELE L))
   (T (CONS (CAR L) (INSERE ELE (CDR L))))))
```

Ci-dessous deux exemples d'appels :

2. La fonction **NCONC1** peut être définie par :

```
(DEFMACRO NCONC1 (L ELE) '(NCONC ,L (LIST ,ELE)))
```

```
(TRIE-UNE-LISTE '(12 0 3 1 -5 6))   ->   (-5 0 1 3 6 12)
(TRIE-UNE-LISTE '(1 2 3 3 2 1))     ->   (1 1 2 2 3 3)
```

Maintenant supposons que nous ayons à écrire une fonction qui trie une liste de nombres en ordre descendant. D'après ce que nous avons vu , cette deuxième fonction peut utiliser le même algorithme que celui précédemment utilisé. La seule modification nécessaire sera le changement du prédicat < en le prédicat >. Ceci donne alors la fonction suivante :

```
(DEFUN TRIE-UNE-LISTE-EN-ORDRE-DECROISSANT (L)
   (INSERE-1 (CAR L) (TRIE-UNE-LISTE-EN-ORDRE-DECROISSANT (CDR L))))

(DEFUN INSERE-1 (ELE L) (COND
   ((NULL L) (LIST ELE))
   ((> ELE (CAR L)) (CONS ELE L))
   (T (CONS (CAR L) (INSERE-1 ELE (CDR L))))))
```

Si, de plus, nous voulons écrire une fonction qui trie une liste de caractères, nous devrions écrire encore une fonction similaire, en utilisant cette fois le prédicat **CHAR<** ou **CHAR>**, selon que la liste doit être triée en ordre croissant ou décroissant.

C'est une situation plutôt pénible : puisque l'algorithme utilisé dans toutes ces fonctions est le même, et puisque seul le prédicat change, nous aimerions écrire une fonction plus générale, qui intègre toutes ces diverses versions. Donc, nous aimerions donner le prédicat comme un argument à la fonction. De cette manière, nous pourrions appeler **TRIE-UNE-LISTE** avec deux arguments : une liste à trier et un prédicat spécifiant suivant quelle règle il faut trier. Par exemple, l'appel :

(TRIE-UNE-LISTE *liste-de-nombres* **#'<)**

pour trier une liste de nombres en ordre croissant, et l'appel :

(TRIE-UNE-LISTE *liste-de-caractères* **#'CHAR>)**

pour trier une liste de caractères en ordre décroissant.

Une première version de cette nouvelle version plus générale peut s'écrire comme :

```
(DEFUN TRIE-UNE-LISTE (L PREDICAT)
   (INSERE PREDICAT (CAR L) (TRIE-UNE-LISTE (CDR L))))

(DEFUN INSERE (PREDICAT ELE L) (COND
   ((NULL L) (LIST ELE))
   ((APPLY PREDICAT (LIST ELE (CAR L))) (CONS ELE L))
   (T (CONS (CAR L) (INSERE PREDICAT ELE (CDR L))))))
```

Ici nous avons bien introduit un argument supplémentaire. Cet argument, **PREDICAT**, sera lié à une fonction et à l'intérieur de la fonction **INSERE**, il sera appliqué aux arguments **ELE** et **(CAR L)**.

Ci-dessous nous donnons quelques exemples d'appels de cette fonction :

```
?  (TRIE-UNE-LISTE '(1 2 3 3 2 1 0 -1) #'>)
=  (3 3 2 2 1 1 0 -1)
?  (TRIE-UNE-LISTE '(#\d #\c #\b #\a) #'CHAR<)
=  (#\a #\b #\c #\d)
?  (TRIE-UNE-LISTE '(1 2 3 3 2 1 0 -1) #'<)
=  (-1 0 1 1 2 2 3 3)
```

Nous avons rencontré déjà préalablement, au chapitre 12, la fonction **APPLY**. Il existe une autre fonction, similaire à **APPLY**, la fonction **FUNCALL**. **FUNCALL** est une fonction standard à nombre quelconque d'arguments qui peut être définie comme suit :

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

La différence entre **FUNCALL** et **APPLY** est que la deuxième attend toujours *deux* arguments : une fonction à calculer et une liste de ses arguments, alors que la première attend comme arguments une fonction à calculer et les arguments de cet appel de fonction. Par exemple, les deux expressions suivantes calculent exactement la même chose :

```
(APPLY 'CONS '(A (B C D)))
(FUNCALL 'CONS 'A '(B C D))
```

et, si la valeur du symbole **F** est l'atome **CONS**, les deux expressions suivantes calculent encore la même valeur :

```
(APPLY F 'A '(B C D)))
(FUNCALL F 'A '(B C D))
```

On peut utiliser la possibilité de calculer la fonction pour construire des fonctions avec des arguments fonctionnels. Voici une fonction LISP qui applique la fonction **F** *deux* fois à son argument **X** :

```
(DEFUN TWICE (F X) (FUNCALL F (FUNCALL F X))
```

Voici quelques exemples d'appels :

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

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

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

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

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

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

Ce type de fonction se présente si souvent, qu'il est utile de construire une fonction instanciant ce schéma. La voici :

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

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

```
(DEFUN PLUS-UN (L) (MAPCAR #'1+ L))

(DEFUN LIST-EACH (L) (MAPCAR #'LIST L))
```

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

Prenons un moment pour admirer la puissance de cette fonction : en prenant avantage de la possibilité de passer des fonctions en argument à des fonctions, nous avons construit une fonction qui sépare complètement la structure de contrôle des calculs réellement effectués. Cette fonction peut être utilisée dans maintes applications, chacune spécifiée exclusivement par l'argument fonctionnel adéquat. Voici quelques exemples supplémentaires de cette fonction **MAPCAR** :

```
? (MAPCAR 'FACTORIAL '(1 2 3 4 5))
= (1 2 6 24 120)
? (MAPCAR 'NUMBERP '(1 A B 2 3 C))
= (T NIL NIL T T NIL)
? (MAPCAR 'SQRT '(10 20 30 40 50))
= (3.162278S0 4.472136S0 5.477226S0 6.324555S0 7.071068S0)
? (MAPCAR 'ABS '(-200 100 -3 -4 5 6))
= (200 100 3 4 5 6)
? (MAPCAR 'RANDOM '(100 200 300 400))
= (59 158 6 162)
```

En réalité, la fonction **MAPCAR** standard est plus puissante : si la fonction qu'on désire appliquer répétitivement demande plus d'un argument, nous pouvons donner plus d'une liste d'arguments à **MAPCAR**. Supposons, par exemple, que nous voulions construire une liste contenant des sous-listes construites à partir des quatre premiers nombres impairs et des quatre premiers nombres pairs, de la manière suivante :

```
((1 . 2) (3 . 4) (5 . 6) (7 . 8))
```

Nous pouvons réaliser cela, par l'appel de **MAPCAR** suivant :

```
(MAPCAR #'CONS '(1 3 5 7) '(2 4 6 8))
```

La fonction **MAPCAR** évalue chacun des arguments. Ensuite elle applique la valeur du premier argument (un objet fonctionnel) aux **CAR**s successifs des autres arguments. De cette manière, **CONS** reçoit d'abord 1 et 2, ensuite 3 et 4, et ainsi de suite jusqu'à la fin de la première liste d'arguments. **MAPCAR** collecte tous ces résultats successifs et les retourne dans une liste.

Quelques autres exemples d'appels de **MAPCAR** sont présentés ci-dessous :

```
? (MAPCAR #'+ '(1 2 3 4 5) '(10 20 30 40 50))
=(11 22 33 44 55)

? (MAPCAR #'APPEND '((A B)(C D)(E F)) '((1 2)(3 4)(5 6)))
=((A B 1 2)(C D 3 4)(E F 5 6))

? (SETQ LIST1 '((A B)(C D)(E F)) LIST2 '(1 2 3))
=(1 2 3)

? (MAPCAR #'RPLACA LIST1 LIST2)
=((1 B)(2 D)(3 F))

? LIST1
=((1 B)(2 D)(3 F))
```

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

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

Regardez bien le dernier exemple : il utilise une λ-expression. Une λ-expression est une fonction sans nom. En Le_LISP l'évaluation d'une telle expression livre cette λ-expression elle-

même.

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

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

```
(DEFMACRO LET (&REST L)
        '((LAMBDA ,(MAPCAR #'CAR (CAR L)) ,@(CDR L))
          ,@(MAPCAR #'CADR (CAR L))))
```

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

```
(DEFUN ENL-A (L)
    (MAPCAR '(LAMBDA (X) (DELETE 'A X)) L))
```

Si on appelle :

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

le résultat est la liste :

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

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

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

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

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

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

La fonction **MAPC** est une simplification de **MAPCAR**, définie par :[3]

```
(DEFUN MAPC (F L) (COND
     ((NULL L) ())
     (T (FUNCALL F (CAR L)) (MAPC F (CDR L)))))
```

L'appel :

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

livre alors :

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

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

```
(DEFUN PERMUT (L)
     (LABELS ((PERMUT1 (L RES &OPTIONAL X)
          (MAPC #'(LAMBDA (ELE)
                    (SETQ X (DELQ ELE L))
                    (IF X (PERMUT1 X (CONS ELE RES))
                          (PRINT (CONS ELE RES))))
               L)))
       (PERMUT1 L NIL)))
```

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

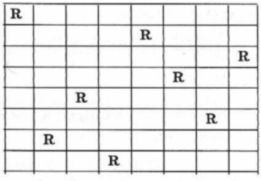
Rappelons brièvement le problème : un échiquier est un tableau de 64 cases répartis en 8 lignes et 8 colonnes. Une reine est une pièce du jeu qui peut prendre toute pièce qui se trouve sur la même ligne, sur la même colonne, ou sur la même diagonale qu'elle.

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

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

représente la position suivante :

3. La fonction LISP standard **MAPC** est plus générale et admet des arguments fonctionnels à plus d'un argument. *Dans ces cas, de même qu'avec* **MAPCAR**, *les divers arguments sont groupés dans des listes.* **MAPC** applique alors la fonction aux **CAR**s successifs des listes d'arguments.



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

```
(DEFUN REINE (LISTE &OPTIONAL (COLONNE 1) RESUL DIAG1 DIAG2 AUX)
       (MAPC #'(LAMBDA (X)
                       (SETQ AUX (DELQ X LISTE))
                       (IF (AND (NULL (MEMQ (- X COLONNE) DIAG1))
                                (NULL (MEMQ (+ X COLONNE) DIAG2)))
                           (IF (NULL AUX) (PRINT (CONS X RESUL))
                               (REINE AUX
                                      (1+ COLONNE)
                                      (CONS X RESUL)
                                      (CONS (- X COLONNE) DIAG1)
                                      (CONS (+ X COLONNE) DIAG2)))))
               LISTE))
```

Un appel se fera alors par :

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

Pour ne chercher qu'une seule des solutions possibles au problème de 8 reines, nous pouvons utiliser le même algorithme - *si* nous trouvons un moyen d'arrêter la récursion et les itérations à l'instant où nous en avons trouvé une première. C'est une bonne occasion pour utiliser les fonctions **CATCH** et **THROW**. En général, quand vous êtes dans des itérations et/ou des récursions multiples et/ou imbriquées, la manière la plus simple de sortir *d'un seul coup* de toutes ces structures répétitives est de positionner un **CATCH** à l'extérieur de ces structures et de lancer un **THROW** à l'endroit où vous voulez sortir. Ceci permet de transformer un algorithme de génération d'objets en un algorithme de recherche d'un tel objet. Voici alors la fonction **REINE** modifiée pour ne trouver qu'un seule solution et donc, pour vérifier si une solution existe :

4. Une très bonne description de ce problème se trouve dans l'excellent livre *La construction de programmes structurés* de Jacques Arsac.

```
(DEFUN REINE (LISTE)
    (LABELS ((REINE-AUX
                (LISTE &OPTIONAL (COLONNE 1)
                 RESUL DIAG1 DIAG2 AUX)
        (MAPC #'(LAMBDA (X)
                (SETQ AUX (DELQ X LISTE))
                (IF (AND (NULL (MEMQ (- X COLONNE) DIAG1))
                         (NULL (MEMQ (+ X COLONNE) DIAG2)))
                    (IF (NULL AUX) (THROW 'TROUVE (CONS X RESUL))
                        (REINE-AUX AUX3
                                (1+ COLONNE)
                                (CONS X RESUL)
                                (CONS (- X COLONNE) DIAG1)
                                (CONS (+ X COLONNE) DIAG2)))))
            LISTE)))
      (CATCH 'TROUVE (REINE-AUX LISTE))))
```

La fameuse fonction **LIT** est similaire à **MAPCAR**. La voici :

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

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

```
(DEFUN APPEND (L1 L2) (LIT L1 L2 #'CONS))
```

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

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

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

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

18.3 QUELQUES ITERATEURS SUPPLEMENTAIRES

En plus des possibilités de répétition par la récursivité, LISP dispose de quelques *itérateurs* : les fonctions **WHILE**, **UNTIL** et **DO**. (CommonLISP a, dans sa version standard, seulement l'itérateur **DO**.)

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

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

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

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

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

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

La fonction **UNTIL** est très similaire à **WHILE**. Elle est définie comme :

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

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

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

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

```
(DEFMACRO WHILE (&WHOLE ALL &REST CALL)
          `(IF (NULL ,(CAR CALL)) ()
               (PROGN ,@(CDR CALL) ,ALL)))
```

Remarquez que nous avons utilisé un nouveau mot-clef : **&WHOLE**. Le paramètre associé à ce mot-clef, ici c'est la variable **ALL**, est lié à l'appel tout entier de la macro-fonction. Par exemple, si nous appelons la macro **WHILE** avec :

```
(WHILE (> N 0) (PRINT (SETQ N (1- N))))
```

la variable **ALL** sera liée à cette expression même, c'est-à-dire : à la liste représentant l'appel de la macro. C'est cette particularité qui nous permet, dans la dernière ligne, de construire une liste circulaire - comme nous l'avons fait au début de ce chapitre. L'expression **,ALL** sera remplacée par un pointeur vers l'appel de la macro **WHILE**. Etant donné que cet appel a été modifié par la

macro, à cet endroit nous trouvons maintenant l'expression débutant par

(IF (NULL (> N 0)) () (PROGN ....

nous avons bel et bien une liste circulaire dans laquelle toute référence à la macro-fonction **WHILE** a disparu.

MACLISP, un dialecte de LISP devéloppé au Massachussets Institute of Technology, a introduit l'itérateur **DO** qui est défini par :

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

Cet itérateur se compose donc de trois parties :

1. Une liste de déclaration de variables, ou pour chaque variable on indique - comme dans un **LET** - sa valeur initiale ainsi que - c'est l'originalité de la fonction **DO** - l'expression du calcul de la nouvelle valeur pour chaque itération.

2. Une clause du style **COND**, où le **CAR** est donc un test et le reste la suite des instructions à évaluer si le test donne vrai. Ce test détermine l'arrêt de l'itération.

3. Un corps qui indique la suite des instructions à évaluer à chaque itération.

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

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

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

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

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

Voici une définition possible de la macro **DO** :

```
(DEFMACRO DO (&REST CALL)
    (LET ((INIT (CAR CALL))
          (STOPTEST (CAAR (CDR CALL)))
          (RETURN-VALUE (CDAR (CDR CALL)))
          (BODY (CDDR CALL)))
      `((LAMBDA ,(MAPCAR 'CAR INIT)
          (WHILE (NULL ,STOPTEST)
              ,@ BODY
              ,@ (MAPCAR '(LAMBDA (X)
                           `(SETQ ,(CAR X) ,(CADDR X)))
                         INIT))
          ,@ RETURN-VALUE)
        ,@ (MAPCAR 'CADR INIT))))
```

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

```
(DEFUN REVERSE (L)
  ((LAMBDA (RES L)
     (WHILE (NULL (NULL L)) (SETQ RES (CONS (CAR L) RES))
            (SETQ L (CDR L)))
     RES)
   NIL L))
```

Ici, seule la macro **DO** a été expansée. Si nous expansons également la macro **WHILE**, nous aurons quelque chose comme la fonction :

```
(DEFUN REVERSE (L)
  ((LAMBDA (RES L)
     (IF (NULL (NULL (NULL L))) NIL
         (PROGN
           (SETQ RES (CONS (CAR L) RES))
           (SETQ L (CDR L))
           (WHILE (NULL (NULL L)) (SETQ RES (CONS (CAR L) RES))
           (SETQ L (CDR L)))))
     RES)
   NIL L))
```

Deux remarques s'imposent pour cette dernière fonction :

1. Ce n'est pas la fonction la plus élégante de toutes celles que nous avons vues : le prédicat à l'air particulièrement abscons. Néanmoins, la forme originale était très lisible, et tout bon compilateur saura optimiser le double appel de **NULL**.

2. Le **WHILE** restant à l'intérieur de cette fonction est juste un moyen pour exprimer la structure circulaire. En réalité, il n'y a pas un appel de **WHILE**, mais à cet endroit il y a un pointeur de retour vers l'appel du **IF** généré par la macro **WHILE**.

18.4 EXERCICES

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

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

2. Ecrivez la macro-fonction **UNTIL** avec une liste circulaire.

3. Ecrivez la fonction **FACTORIELLE** avec l'itérateur **DO**

4. Définissez les macro-fonctions **MAPC** et **MAPCAR** de sorte qu'elles acceptent des arguments fonctionnels à plusieurs arguments.

19. LE FILTRAGE (DEUXIEME PARTIE)

Au chapitre 14 nous avons construit une fonction de filtrage, la fonction **MATCH**, qui permet de comparer une liste avec un *modèle*, ou *filtre*, de liste.

Le filtre peut contenir 3 signes particuliers :

1. le signe **$** filtre un élément quelconque,

2. le signe **?** filtre une séquence, éventuellement vide, d'éléments quelconques,

3. le signe **%** filtre un choix entre plusieurs éléments donnés.

Ainsi, le filtre **($ $ $)** correspond à toute liste comprenant exactement 3 éléments, le filtre **((%A B C) $)** à toute liste d'exactement 2 éléments commençant par l'un des trois éléments **A**, **B** ou **C**, et le filtre **(DEBUT ? FIN)** à toute liste commençant par l'élément **DEBUT** et se terminant par l'élément **FIN**.

Cette fonction de filtrage n'est pas complète : on est parfois intéressé non seulement par la structure de la liste, mais aussi par l'élément qui se situe à l'endroit d'un des signes particuliers. Ainsi, lorsqu'on compare le filtre **($ SUR A)** à un ensemble de données, on aimerait peut-être savoir ce qui se trouve à l'endroit du signe **$**.

Enrichissons alors un peu la syntaxe de nos filtres pour permettre aux opérateurs de filtrage d'être suivis du nom d'une variable, tel qu'au lieu d'écrire :

($ SUR A)

nous puissions écrire :

($X SUR A)

De plus, enrichissons notre fonction de filtrage de manière à indiquer non seulement qu'une certaine donnée correspond au filtre, mais également quel élément se trouve à l'endroit de cette variable.

Par exemple, si nous filtrons :

($X SUR A)

avec la donnée :

(D SUR A)

la fonction de filtrage ramène **T**, c'est-à-dire l'indication que la donnée correspond au filtre, mais aussi l'information qu'à l'endroit de **$X** se trouve l'élément **D**.

Evidemment, deux problèmes se posent. D'abord, il faut changer notre fonction **MATCH** de manière à ramener *deux* valeurs : l'indication de réussite ou d'échec du filtrage, plus éventuellement les *liaisons* des diverses variables du filtre. Ensuite il faut trouver un moyen d'exprimer ces liaisons. Il existe deux possibilités : soit en le faisant comme LISP, c'est-à-dire en changeant les C-valeurs des variables (avec les fonctions **SET** ou **SETQ**), soit en retournant une liste qui contienne textuellement les noms des variables suivis de leurs valeurs, donc une liste de la forme :

$$((variable_1 \,.\, valeur_1) \ldots (variable_n \,.\, valeur_n))$$

Des listes de cette forme s'appellent des *A-listes* ou des *listes d'associations*. Les toutes premières implémentations de LISP utilisaient de telles listes comme *environnement*, c'est-à-dire comme mémoire des liaisons actuelles et passées.

Ainsi, le résultat du filtrage présenté ci-dessus doit livrer le résultat :

(T ((X . D)))

où le **T** du début de la liste est l'indicateur de réussite du filtrage, et le deuxième élément de la liste est la A-liste qui exprime la liaison de **X** à **D**.[1] Donnons, en guise de définition de nos nouveaux filtres quelques exemples de filtrage :

- **(MATCH '(A B C D) '(A B C D)) → (T NIL)**
  Si le filtre ne contient aucun signe spécial le filtrage ne réussit qu'avec une donnée qui est **EQUAL** au filtre. Le résultat est une liste, contenant comme **CAR** l'indicateur de réussite du filtrage **T** et comme **CADR** la A-liste vide, puisqu'aucune liaison n'est nécessaire.

- **(MATCH '($X B C $-) '(A B C D)) → (T ((X . A)))**
  Au filtre commençant par le signe $ doit correspondre un unique élément dans la donnée. Si le signe $ est suivi d'un atome différent de -, cet atome est alors lié à l'élément correspondant dans la donnée. Le filtre spécial $- indique que l'élément correspondant doit être présent dans la donnée, mais sans que l'on procède à une liaison. Ce dernier filtre correspond alors au signe $ de notre fonction de filtrage du chapitre 14.

- **(MATCH '($X $Y $Z $X) '(A B C A)) → (T ((X . A) (Y . B) (Z . C)))**
  La répétition d'une même variable de filtre à l'intérieur d'un filtre indique des éléments égaux. Ainsi, au filtre ci-dessus correspond une donnée de 4 éléments dont le premier et le dernier élément sont identiques.

- **(MATCH '($X ?- $X) '(A B C A)) → (T ((X . A)))**
  Ce filtre décrit toute liste qui commence et qui se termine par le même élément. Le filtre **?-** correspond au filtre **?** de notre fonction **MATCH** du chapitre 14, il filtre donc un segment de longueur quelconque *sans* procéder à une liaison.

1. Si l'on désire une liaison *superficielle*, c'est-à-dire une liaison qui modifie les C-valeur des variables, il suffit de faire sur la partie A-liste du résultat :

   (MAPC '(LAMBDA (X) (SET (CAR X) (CDR X))) *A-liste*)

- (MATCH '($X ?Y $X) '(A B C A)) → (T ((X . A) (Y B C)))
  La différence entre ce filtre et le précédent est que le segment filtré est lié à la variable Y. Cette variable est donc liée à la liste (B C). (Rappelons que le CDR de chaque doublet de la A-liste contient la valeur de la variable du CAR du doublet. C'est pourquoi nous avons le doublet (Y B C) dans la A-liste résultat.)

- (MATCH '(%(A B C) ?-) '(A B C A)) → (T NIL)
  Comme préalablement, le signe spécial % indique un choix multiple. Ce filtre décrit donc toute liste de longueur quelconque commençant soit par l'atome A, soit par B, soit par C.

Modifions alors notre programme de filtrage qui était :

```
(DEFUN MATCH (FILTRE DONNEE)
  (LABELS ((CHOIX (AUX FILTRE DONNEE) (COND
           ((NULL AUX) NIL)
           ((MATCH (CAR AUX) (CAR DONNEE))
              (MATCH (CDR FILTRE)(CDR DONNEE)))
           (T (CHOIX (CDR AUX) FILTRE DONNEE)))))
    (COND
      ((ATOM FILTRE)(EQUAL FILTRE DONNEE))
      ((EQUAL (CAR FILTRE) '|?|) (COND
        ((NULL (CDR FILTRE)) T)
        ((MATCH (CDR FILTRE) DONNEE))
        (DONNEE (MATCH FILTRE (CDR DONNEE)))))
      ((ATOM DONNEE) NIL)
      ((EQUAL (CAR FILTRE) '|$|)
        (MATCH (CDR FILTRE) (CDR DONNEE)))
      ((AND (CONSP (CAR FILTRE))(EQUAL (CAAR FILTRE) '|%|))
        (LET ((AUX (CDR (CAR FILTRE))))
          (CHOIX AUX FILTRE DONNEE)))
      ((MATCH (CAR FILTRE) (CAR DONNEE))
        (MATCH (CDR FILTRE) (CDR DONNEE))))))
```

En premier lieu, il s'agit d'introduire la A-liste. Changeons donc le début de la définition de MATCH en :

```
(DE MATCH (FILTRE DONNEE)
    (LET ((ALISTE NIL)) . . .
```

La tâche sera divisée en deux : le travail proprement dit sera réalisé par une fonction auxiliaire, alors que la fonction MATCH se contente de lancer la fonction auxiliaire et d'éditer le résultat.

```
(DEFUN MATCH (FILTRE DONNEE)
  (LET ((ALISTE NIL))
    (DECLARE (SPECIAL ALISTE))
    (IF (MATCH-AUX FILTRE DONNEE) (LIST T ALISTE) NIL)))
```

Ici nous avons *déclaré* la variable ALISTE en tant qu'une variable SPECIALe. Pourquoi ? En

LISP, on n'a normalement pas besoin de déclarer une variable, sauf pour indiquer au compilateur quelques faits qu'il ne peut pas déduire lui-même. Par exemple, il est pensable qu'on veuille indiquer au compilateur qu'une certaine variable ne prend que des valeurs entières inférieures à 100. Ceci permettrait une compilation plus efficace. Mais, ici nous ne sommes pas concernés par la compilation; elle sera traîtée dans un autre volume.

La seule déclaration ayant un effet pendant l'*interprétation* des programmes LISP est la déclaration d'une variable en tant que variable *spéciale*. Etre spécial veut dire : la variable obéit à des mecanismes d'accès non-standard, donc spéciaux. Normalement, une variable réfère soit à sa définition locale (c'est-à-dire qu'elle est définie à l'intérieur du bloc, la fonction ou la λ-expression, où elle est référencée) soit elle réfère à sa définition au niveau racine de LISP : c'est alors une variable globale. Ce mécanisme d'accès s'appelle une *liaison lexicale* des variables; par abus de langage on nomme un système LISP obéissant à cette discipline d'accès un *LISP lexical*.

En CommonLISP, il est néanmoins possible d'avoir des variables qui obéissent à un mécanisme de *liaison dynamique*. Dans une *liaison dynamique*, une variable est définie non seulement dans le bloc où elle est définie (i.e.: où elle se trouve dans une λ-liste), mais également dans *toutes* les autres fonctions ou λ-expressions qui sont appelées à partir de cet endroit de la définition. Ceci implique qu'il est possible d'avoir des variables qui ne sont pas strictement *locales* à une fonction. Puisque ce type de variables n'est pas standard en CommonLISP, elles sont appelées des variables *spéciales* et elles doivent être déclarées en tant que telles.

Ce type de liaison dynamique, est le mécanisme de liaison standard de la plupart des systèmes LISP avant l'arrivée de CommonLISP. Si vous utilisez un autre LISP, vous devez tout d'abord examiner quel est le mécanisme de liaison utilisé.

Notons, finalement, que le problème de déterminer si une variable obéit à la discipline de liaison lexicale ou dynamique n'est important que pour des variables ayant une occurrence à l'intérieur de fonctions (ou expressions) où elles ne font pas partie de la liste de paramètres. De telles occurrences sont nommées des *occurrences libres*. Si c'est le cas, cette variable réfère alors soit à la variable spéciale la plus récemment définie,[2] soit – si aucune variable spéciale de ce nom n'existe – à une variable globale de ce nom.

En résumé : une variable qui n'est pas déclarée comme variable spéciale ne peut être référencée qu'à l'intérieur de la fonction ou de la λ-expression où elle est définie. Une variable spéciale peut être référencée à l'intérieur de toute fonction qui est (directement ou indirectement) appelée par la fonction où elle a été définie.

Voici un petit programme exemple illustrant cet aspect :

```
(SETQ X 'X Y 'Y Z 'Z) ; nos variables globales

(DEFUN FOO (X Y Z)
    (DECLARE (SPECIAL Z))
    (BAR X))
```

2. *Récemment* se réfère ici à la dynamique de l'exécution du programme.

```
(DEFUN BAR (X)
    (FORMAT T "~%X = ~A Y = ~A Z = ~A" X Y Z))
```

Ce programme utilise trois variables globales : **X**, **Y** et **Z**. Toutes ces variables globales sont initialisées à leurs noms respectifs.

La fonction **FOO** définit deux variables *lexicales* : **X** et **Y**, et une variable *spéciale* : **Z**.

La fonction **BAR** définit une variable *lexicale* : **X**. Elle contient une occurrence *libre* des variables **Y** et **Z**.

Appelons alors la fonction **FOO** :

```
? (FOO 1 2 3)
X = 1 Y = Y Z = 3
```

Du résultat de cet appel, nous pouvons déduire qu'à l'intérieur de la fonction **BAR** l'occurrence libre de **Y** réfère à la variable *globale* de ce nom et l'occurrence libre de **Z** réfère à sa définition *spéciale* la plus récente.

Retournons alors vers notre fonction de filtrage. Dans la nouvelle version de la fonction **MATCH**, nous déclarons la variable **ALISTE** comme variable **spéciale** puisque nous voulons que *cette* variable soit connue de toutes les autres fonction appelées par **MATCH**.

Ensuite réécrivons l'ancienne fonction **MATCH** qui s'appelle maintenant **MATCH-AUX** : dans les deux premières clauses il n'y a rien à changer. Les modifications doivent se situer dans le traitement des signes spéciaux **$** et **?**.

Résolvons d'abord la difficulté qui consiste à séparer le signe (**$** ou **?**) du nom de la variable qui suit. Si nous utilisons la fonction **EXPLODECH** pour tester le premier caractère d'un atome, nous devons le faire avec *tous* les atomes contenus dans le filtre. C'est trop coûteux en temps. Définissons donc les caractères **$**, **?** et **%** comme des macro-caractères. Ainsi, dès la lecture, les signes spéciaux seront séparés des noms des variables qui suivent. Voici les définitions de ces trois caractères, d'abord en Le_LISP :

```
(DMC |$| () (CONS '|$| (READ)))

(DMC |?| () (CONS '|?| (READ)))

(DMC |%| () (CONS '|%| (READ)))
```

et ensuite en CommonLISP :

```
(SET-MACRO-CHARACTER #\$
    #'(LAMBDA (FLUX CARACTERE) (CONS '|$| (READ FLUX T NIL T))))

(SET-MACRO-CHARACTER #\?
    #'(LAMBDA (FLUX CARACTERE) (CONS '|?| (READ FLUX T NIL T))))
```

```
(SET-MACRO-CHARACTER #\%
    #'(LAMBDA (FLUX CARACTERE) (CONS '|%| (READ FLUX T NIL T))))
```

Ainsi, le filtre :

```
($X ?Y %(A B C))
```

sera traduit en :

```
(($ . X) (? . Y) (% A B C))
```

et les tests de présence de signes spéciaux n'ont lieu que si le filtre débute par une sous-liste. Nous pouvons donc introduire une clause supplémentaire testant si le filtre commence par un atome. Si c'est le cas, nous savons qu'il doit s'agir d'une constante et qu'il suffit de comparer cette constante avec le premier élément de la donnée pour ensuite relancer le processus de filtrage sur le reste des deux listes. Ce qui nous donne :

```
((ATOM (CAR FILTRE))
    (AND (EQUAL (CAR FILTRE) (CAR DONNEE))
         (MATCH-AUX (CDR FILTRE) (CDR DONNEE))))
```

Dans tous les autres cas, le filtre débute par une sous-liste. Cette sous-liste peut alors correspondre à un des caractères spéciaux. Etudions alors tous les cas et commençons par le caractère $:

```
((EQUAL (CAAR FILTRE) '|$|) . . .)
```

Trois sous-cas peuvent se présenter :

1. la donnée est la liste vide. Dans ce cas le filtrage ne réussit pas et nous pouvons sortir en ramenant **NIL**.

2. le signe de dollar est suivi du signe '-'. Nous nous trouvons alors dans le même cas que dans l'ancienne fonction **MATCH** : il ne reste qu'à comparer le reste du filtre avec le reste de la donnée.

3. le signe de dollar est suivi du nom d'une variable. Deux cas sont alors à distinguer :

    1. Cette variable n'a pas encore de liaison sur la A-liste. Il faut alors lier la variable à l'élément correspondant dans la donnée et continuer à comparer le reste du filtre avec le reste de la donnée.

    2. La variable a déjà été liée sur la A-liste. Il faut alors comparer le filtre construit en remplaçant le premier élément par la valeur de la variable avec la donnée.

Construisons d'abord une petite fonction auxiliaire **ASSOC** qui cherche sur une A-liste si une variable donnée est déjà liée ou pas. Si elle est liée, **ASSOC** ramène le doublet (*variable* . *valeur*) :

```
(DE ASSOC (VARIABLE ALISTE) (COND
        ((NULL ALISTE) ())
        ((EQ (CAAR ALISTE) VARIABLE) (CAR ALISTE))
        (T (ASSOC VARIABLE (CDR ALISTE)))))
```

Cette fonction existe comme fonction standard dans la plupart des systèmes LISP.

Maintenant, nous pouvons écrire la clause pour le $:

```
((EQUAL (CAAR FILTRE) '|$|) (COND
    ((NULL DONNEE) ())
    ((EQ (CDAR FILTRE) '|-|)
        (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))
    ; regardons si la variable a une liaison ;
    ((ASSOC (CDAR FILTRE) ALISTE)
        (MATCH-AUX
            (CONS (CDR (ASSOC (CDAR FILTRE) ALISTE)) (CDR FILTRE))
            DONNEE))
    ; il n'y a pas de liaison ;
    (T (SETQ ALIST (CONS (CONS (CDAR FILTRE) (CAR DONNEE)) ALISTE))
        (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))))
```

Visiblement, cette clause reprend un à un chacun des cas présentés ci-dessus. Il subsiste une seule maladresse : le double calcul de la liaison de la variable. Il est possible d'attribuer une variable de plus à notre fonction **MATCH-AUX**, que nous nommerons **AUX** pour 'auxiliaire', et d'utiliser cette variable pour sauvegarder temporairement la liaison. Ceci implique que nous modifions la fonction **MATCH-AUX** en :

```
(DE MATCH-AUX (FILTRE DONNEE)
        (LET ((AUX NIL))
            (DECLARE (SPECIAL AUX))
            ....
```

et récrivons l'avant dernière clause :

```
((SETQ AUX (ASSOC (CDAR FILTRE) ALISTE))
    (MATCH-AUX (CONS (CDR AUX) (CDR FILTRE)) DONNEE))
```

Prenons ensuite le cas du filtre **%** de choix multiple explicite. La différence avec l'ancienne fonction réside dans la sauvegarde de l'état de la A-liste pendant les essais successifs; ce qui nous permet d'introduire des filtres à l'intérieur de la liste des choix. Ainsi, le filtre :

```
(%(DO RE (MI $-) (FA $- $-)) ?-)
```

décrit toute liste commençant soit par l'atome **DO**, soit par l'atome **RE**, soit par une liste de deux éléments commençant avec **MI**, soit par une liste de trois éléments et débutant avec l'atome **FA**.

Voici donc la clause correspondant au signe **%** :

```
((EQUAL (CAAR FILTRE) '|%|)
        (CHOIX (CDAR FILTRE) FILTRE DONNEE ALISTE))
```

et la fonction **CHOIX**, locale à **MATCH-AUX**, qui procède au choix effectif :

```
(LABELS ((CHOIX (AUX FILTRE DONNEE ALISTE2) (COND
          ((NULL AUX) (SETQ ALISTE ALISTE2) NIL)
          ((MATCH-AUX (CONS (CAR AUX) (CDR FILTRE)) DONNEE) T)
          (T (CHOIX (CDR AUX) FILTRE DONNEE ALISTE2)))))
```

Vient ensuite le cas difficile des variables de segment, c.a.d.: des variables précédées du signe spécial '**?**'. Nous pouvons distinguer trois cas :

1. La variable est déjà liée. Il faut alors, dans le filtre, remplacer la variable par sa valeur et comparer le filtre ainsi obtenu avec la donnée.

2. Le reste du filtre est vide. Il faut alors lier la variable (sauf si c'est le signe spécial '-') à la donnée et le filtrage est terminé avec succès.

3. Sinon, il faut lier la variable (avec la même exception que ci-dessus) au segment vide **NIL**, regarder si le reste du filtre se superpose à la donnée, et si ça ne marche pas, il faut ajouter le premier élément de la donnée à la valeur de la variable, comparer le reste du filtre avec le reste de la donnée, et ainsi de suite jusqu'à obtenir un succès ou un échec définitif.

E'crivons cette clause :

```
((EQUAL (CAAR FILTRE) '|?|) (COND
    ; existe-t-il une valeur ?
    ((SETQ AUX (ASSOC (CDAR FILTRE) ALISTE))
      ; oui, compare alors la valeur
      (MATCH-AUX (APPEND (CDR AUX)(CDR FILTRE)) DONNEE))
    ; est-ce la fin du filtre ?
    ((NULL (CDR FILTRE))
      ; oui, c'est la fin
      (IF (EQUAL (CDAR FILTRE) '|-|) T
        (SETQ ALISTE
          (CONS (CONS (CDAR FILTRE) DONNEE) ALISTE))) T)
    (T (IF (SETQ AUX (EQUAL (CDAR FILTRE) '|-|)) NIL
        (SETQ ALISTE (CONS (LIST (CDAR FILTRE)) ALISTE)))
      (SEQUENCES ALISTE AUX DONNEE))))
```

Ici nous avons délégué le cas difficile des comparaisons successives à une fonction auxiliaire **SEQUENCES**. Cette fonction sera également locale à **MATCH-AUX**; elle sera donc définie à l'interieur du **LABELS** de ci-dessus. Voici la fonction :

```
(SEQUENCES (ALISTE2 AUX DONNEE) (COND
   (DONNEE (IF
      ; faisons les comparaisons successives
      (MATCH-AUX (CDR FILTRE) DONNEE) T
      (PROGN (SETQ ALISTE ALISTE2)
         (IF AUX (NEXTL DONNEE)
            ; la liaison de la variable s'agrandit
            (NCONC (CAR ALISTE)(LIST (NEXTL DONNEE))))
         (SEQUENCES ALISTE AUX DONNEE))))
   ; échec total : restaurons la A-liste
   (T (IF AUX (NEXTL ALISTE)) NIL))))
```

Il ne reste que le dernier cas : le filtre commence par une sous-liste ordinaire. Comme dans l'ancienne version, il faut alors comparer le premier élément du filtre avec le premier élément de la donnée, et – si ça marche – il faut encore comparer le reste du filtre avec le reste de la donnée.

Voici un ensemble de tests de cette nouvelle fonction **MATCH** :

```
(MATCH '(A B C) '(A B C))                    →  (T NIL)
(MATCH '($- B C) '(A B C))                   →  (T NIL)
(MATCH '($- B C $-) '(A B C))                →  NIL
(MATCH '($X B C) '(A B C))                   →  (T ((X . A)))
(MATCH '($X B $Y) '(A B C))                  →  (T ((Y . C) (X . A)))
(MATCH '($X B $X) '(A B C))                  →  NIL
(MATCH '($X B $X) '(A B A))                  →  (T ((X . A)))
(MATCH '(?- B C) '(A B C))                   →  (T NIL)
(MATCH '(?-) '(A B C))                       →  (T NIL)
(MATCH '(?X %(1 2 3) ?Y) '(A B C 1 A B C D))  →  (T ((Y A B C D) (X A B C)))
(MATCH '(%(A (A $X) (A $X $-)) $X) '((A B C) B))  →  (T ((X . B)))
(MATCH '(?X 1 ?X) '(A B C 1 A B C D))        →  NIL
(MATCH '(?X ?X ?X) '(A B C A B C A B C))     →  (T ((X A B C)))
(MATCH '($A ?- ($- ?B (?C) ?D)) '(1 2 3 (4 5 (6 7))))  →  (T ((D) (C 6 7) (B . 5) (A . 1)))
```

Prenons comme exemple d'un programme utilisant cet algorithme de filtrage, une fonction testant si un mot donné est un palindrome ou non. Rappelons qu'un palindrome est un mot qui, lu à l'envers, livre le même mot. Par exemple, le mot *otto* est un palindrome, de même le mot *aha*, *rotor* ou la suite de mots *emile nu a une lime*.[3]

Voici le début du programme :

```
(DEFMACRO PALINDROME (MOT) '(PAL (EXPLODECH ,MOT)))
```

Ce programme ne fait qu'éclater le mot dans une liste de caractères pour que la fonction auxiliaire **PAL** puisse ensuite comparer le premier et le dernier caractère, et s'ils sont identiques comparer l'avant dernier caractère avec le deuxième, et ainsi de suite, jusqu'à ce qu'on rencontre deux caractères différents ou un seul caractère, ou bien plus rien.

3. Georges Perec a écrit une petite histoire de plusieurs pages sous forme de palindrome, cf. *Oulipo, la littérature potentielle*, pp. 101-106, idées, Gallimard, 1973.

Pour la comparaison du premier et du dernier caractère on peut utiliser le filtre :

```
($X ?MILIEU $X)
```

De plus, ce filtre récupère la suite de caractères qui reste si on enlève les caractères du début et de la fin :

```
(DEFUN PAL (LISTE-DE-CARACTERES)
    (IF (NULL (CDR LISTE-DE-CARACTERES)) 'OUI
        (LET ((AUX (MATCH '($X ?MILIEU $X) LISTE-DE-CARACTERES)))
            (IF (CAR AUX) ; que faire ??? ;
                'NON))))
```

Le problème est : "que faire si le filtrage a réussi ?". A cet instant nous avons dans la variable **AUX** une liste de deux éléments : le premier élément est l'indicateur de réussite du filtrage et le deuxième élément est la A-liste résultant du filtrage. C'est cette A-liste qui nous indique, dans le couple **(MILIEU . *valeur*)**, la suite de caractères qui doit encore être testée. Il suffirait alors d'appeler récursivement **PAL** avec la valeur de **MILIEU**.

Nous pouvons le faire avec l'appel :

```
(PAL (CDR (ASSOC 'MILIEU (CADR AUX))))
```

ou, de manière plus élégante, en mettant l'appel récursif **(PAL MILIEU)** dans un environnement où les liaisons de la A-liste sont temporairement valides.

La fonction suivante construit l'environnement adéquat :

```
(DEFMACRO LETALL (&REST CALL)
  (LET ((ALISTE (EVAL (CAR CALL))))
      `((LAMBDA ,(MAPCAR 'CAR ALISTE) ,@(CDR CALL))
        ,@(MAPCAR '(LAMBDA (X) (LIST 'QUOTE (CDR X))) ALISTE))))
```

Cette macro crée à partir d'une A-liste et d'une suite de commandes une expression ou la suite de commandes est évaluée avec les liaisons déterminées par la A-liste.

Prenons un exemple, l'appel :

```
(LETALL '((X A B C)(Y . 1) (FORMAT T "~A ~A" X Y) (CONS Y X))
```

donne d'abord l'impression **(A B C) 1**, et livre ensuite la liste **(1 A B C)**. Ceci nous permet de compléter notre fonction **PAL** en insérant après **(IF (CAR AUX)** la commande :

```
(EVAL `(LETALL ',(CADR AUX) ',(PAL MILIEU))
```

Ce qui donne finalement :

```
(DEFUN PAL (LISTE-DE-CARACTERES)
    (IF (NULL (CDR LISTE-DE-CARACTERES)) 'OUI
        (LET ((AUX (MATCH '($X ?MILIEU $X) LISTE-DE-CARACTERES)))
            (IF (CAR AUX) (EVAL '(LETALL ',(CADR AUX) ',(PAL MILIEU)))
                'NON))))
```

Tout ce qui reste à faire pour pouvoir exécuter ce programme, c'est de ne pas oublier de déclarer la variable **MILIEU** en tant que variable globale par l'instruction :

```
(DEFVAR MILIEU NIL "variable auxiliaire pour garder le résultat du filtrage")
```

Le filtrage, qui constitue la base de la plupart des langages de programmation issus de LISP, ouvre la voie à une programmation par règles. De tels systèmes interprètent un ensemble de couples <situation, actions>. en les comparant à une situation donnée. L'interprète évalue la partie action du premier couple dont la description de situation est comparable à celle donnée.

Pour rapidement exposer cette technique, reconstruisons un petit programme Eliza, un des premiers programmes d'Intelligence Artificielle. Ce programme simule un psychiatre Rogerien. Le programme est constitué d'un ensemble de couples <filtre, action>. Après chaque énoncé de l'utilisateur (le patient), il compare la suite des filtres avec l'énoncé, et dès qu'un des filtres correspond à la donnée, il imprime la partie action en y reprenant des parties de l'enoncé du patient. Voici un ensemble de telles couples :

filtre	*action*
(au revoir)	()
(stop)	()
(non)	'(vous êtes bien négatif)
(?- lisp ?-)	'(alors vous savez programmer)
(?- ans ?-)	'(remontez plus loin dans vos souvenirs)
(?- annees ?-)	'(cela fait bien longtemps)
($-)	'(c est un peu court)
(?- je ne suis pas ?l)	'(qu est-ce que vous fait pensez que vous n'êtes pas ,@ l)
(?- suis ?l)	'(pourquoi pensez-vous que vous êtes ,@ l)
(?- je ne ?-)	'(vous en êtes sûr)
(?- non ?-)	'(vous êtes très négatif)
(?- vous ?-)	'(concentrez vous sur votre problème, svp)
(?- moi ?-)	'(racontez-moi plus sur votre famille)
(?- je ?-)	'(comment cela se manifeste-t-il)
(?- j ai ?-)	'(depuis combien de temps)
(?- combien ?-)	'(ce n est point la quantité qui compte)
($- $- $-)	'(hmmm)
(?-)	'(je ne vous suis pas très bien)

Grâce à ce petit ensemble de règles, nous pouvons déjà faire des dialogues comme celui-ci (les impressions du programme sont en caractères **gras**, celles de l'utilisateur en caractères normaux) :

(depuis trois années je vis avec Marie)

(cela fait bien longtemps)
(pas du tout)
(hmmm)
(elle ne m'aime plus)
(vous en êtes sûr)
(oui)
(c est un peu court)
(je ne suis pas court)
(qu est-ce que vous fait pensez que vous n'êtes pas court)
(vous vous moquez de moi)
(concentrez-vous sur votre problème, svp)
etc . . .

Bien que ce début de dialogue ait déjà l'air bien intéressant, il est nécessaire, pour le rendre vraiment stimulant, d'accroitre le nombre de couples <filtre, action> et d'améliorer le programme qui les utilise. Voici un exemple, à vous de l'implémenter, de le tester et de l'améliorer :

```
(DEFUN ELIZA ()
    (FORMAT T "~%bonjour, quel est votre problème ? ")
    (DO
      ((SENTENCE (REPOND (ANALYSE (READ))) (REPOND (ANALYSE (READ)))))
      ((NULL SENTENCE) "au revoir")))

(DEFUN ANALYSE (PROBLEME)
  (LABELS ((INTERN-ANALYSE (PAT-REP MATCH-RESULT)
    (IF (CAR (SETQ MATCH-RESULT (MATCH (CAAR PAT-REP) PROBLEME)))
    (REFLECHIT (CADR MATCH-RESULT) (CDAR PAT-REP))
    (IF PAT-REP (INTERN-ANALYSE (CDR PAT-REP) NIL)))))
  (INTERN-ANALYSE LIST-PATTERN-REPONSE NIL)))

(DEFUN REFLECHIT (ENV MODELE) (EVAL '(LETALL ',ENV ,MODELE)))

(DEFUN REPOND (REPONSE)
    (IF REPONSE (FORMAT T "~%PSY: ~A " REPONSE)) REPONSE)
```

La seule chose qui manque encore est de mettre la liste des filtres et des réponses dans la variable globale **LIST-PATTERN-REPONSE** sous la forme :

```
(DEFVAR LIST-PATTERN-REPONSE
      '(  ((AU REVOIR) . ())
          ((STOP) . ())
          ((NON) . '(VOUS ETES BIEN NEGATIF))
          ((?- LISP ?-) . '(ALORS VOUS SAVEZ PROGRAMMER))
          ((?- ANS ?-) . '(REMONTEZ PLU LOIN DANS VOS SOUVENIR))
          ((?- ANNEES ?-) . '(CELA FAIT BIEN LONGTEMPS))
          (($-) . '(C EST UN PEU COURT))
          ; attention, ici il faut un backquote !;
          ((?- JE NE SUIS PAS ?L) .
```

```
                    '(QUE VOUS FAIT PENSER QUE VOUS N ETES PAS ,@L))
          ((?- SUIS ?L) . '(POURQUOI PENSEZ VOUS QUE VOUS ETES ,@L))

          ; et ainsi de suite tous les couples donnés précédemment ;

     ))
```

Mais revenons à notre filtrage.

Si nous pouvions accrocher des restrictions sur nos variables de filtre, l'écriture d'un programme comme **PAL** pourrait se simplifier encore énormément. Reprenons le palindrome : une autre manière de définir ce qu'est un palindrome est de dire que c'est une suite de lettres suivie de la même suite de lettres à l'envers, éventuellement séparée par une unique lettre. Ce qui pourrait s'exprimer par :

(?X ?X-*à-l'envers*) ou (?X $Y ?X-*à-l'envers*)

ou, de manière plus propre à LISP :

(OR (?X ?(Y (EQUAL X (REVERSE Y)))) (?X $M ?(Y (EQUAL X (REVERSE Y)))))

ou, de manière plus compacte :

(?X ?(Z (<= (LENGTH Z) 1)) ?(Y (EQUAL X (REVERSE Y))))

Pour introduire cette modification supplémentaire, il suffit d'ajouter un module qui traduit les variables des contraintes en leur valeur et évalue la contrainte à l'aide de la macro-fonction **LETALL**. Il est donc nécessaire d'introduire ce test de validité des contraintes dans la clause traitant les variables "élément" et celle des variables "segment".

```
((EQUAL (CAAR FILTRE) '|$|) (COND
      ((NULL DONNEE) NIL)
      ((EQ (CDAR FILTRE) '|-|)
           (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))
      ; regardons si la variable a une liaison ;
      ((SETQ AUX (ASSOC (CDAR FILTRE) ALISTE))
           (MATCH-AUX (CONS (CDR AUX) (CDR FILTRE)) DONNEE))
      ; il n'y a pas de liaison ;
      (T (SETQ ALIST
           (CONS (CONS (VAR (CDAR FILTRE)) (CAR DONNEE)) ALISTE))
          (IF (OR (ATOM (CDAR FILTRE)) (NULL (CDDAR FILTRE))
                  (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE))))
              (MATCH-AUX (CDR FILTRE) (CDR DONNEE))
              (PROGN (NEXTL ALIST) NIL)))))
```

La fonction **VAR** est nécessaire pour trouver la partie variable dans le filtre, étant donné que celle-ci peut être maintenant en position **CDR** (s'il n'y a pas des contraintes) ou **CADR** (s'il y a des contraintes) :

filtre	*donnée*	*résultat*
($X A B $(Y (EQ Y (+ 5 X))))	(5 A B 10)	(T ((Y . 10) (X . 5)))
($X A B $(Y (EQ Y (+ 5 X))))	(5 A B 5)	NIL
(?X A B ?(Y (EQUAL Y (REVERSE X))))	(1 2 A B 2 1)	(T ((Y 2 1) (X 1 2)))
(?X ?(Y (EQUAL Y (REVERSE X))))	(A B C C B A)	(T ((Y C B A) (X A B C)))

Un seul problème subsiste dans notre fonction de filtrage. Regardez l'exemple suivant :

```
(MATCH '((?X + ?Y) * (?X - ?Y)) '((A + B + C) * (A + B - C)))
```

D'après ce que nous savons, le résultat du filtrage doit être :

```
(T ((X A + B) (Y C)))
```

Néanmoins, notre fonction de filtrage échoue. Pourquoi ? Le problème vient de ce que nous utilisons la récursivité et le mécanisme de retour, généralement appelé *backtracking*. Ce backtracking, nous le faisons de manière explicite dans la clause concernant les variables de segment. Là, nous pouvons revenir vers l'arrière (et donc défaire des liaisons sur la A-liste) jusqu'à ce que nous trouvions quelque chose qui marche, ou jusqu'à ce que nous ayons épuisé toutes les possibilités.

Malheureusement, dans la dernière clause de **MATCH-AUX** :

```
((MATCH-AUX (CAR FILTRE) (CAR DONNEE))
        (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))))))
```

nous interdisons les retours vers l'arrière : une fois terminée la comparaison du premier élément du filtre, ici (**$X + $Y**), avec le premier élément de la donnée, ici (**A + B + C**), nous ne pouvons plus défaire la liaison résultante, même quand nous constatons par la suite, dans la comparaison des **CDR**s respectifs, qu'elle est incorrecte.

Manquant de structures de contrôle plus sophistiquées, la seule solution qui nous reste, consiste à traduire les filtres et les données à l'entrée de la fonction **MATCH** en listes linéaires puis de retraduire les résultats à la sortie.

Ainsi, le filtre :

```
((?X + ?Y) * (?X - ?Y))
```

peut être linéarisé de la manière suivante :

```
(<< ?X + ?Y >> * << ?X - ?Y >>)
```

la donnée ((A + B + C) * (A + B - C)) peut être linéarisée en :

```
(<< A + B + C >> * << A + B - C >>)
```

3. Nous examinerons de telles structures de contrôle dans un deuxième livre sur LISP.

et de cette façon, la dernière clause de notre fonction de filtrage, celle qui inhibe le retour en arrière, peut être éliminée, puisqu'elle ne sera plus nécessaire : tous les filtres et toutes les données seront linéaires !

Le programme de filtrage complet intégrant toutes nos modifications a maintenant la forme suivante :

```
(DEFUN MATCH (FILTRE DONNEE) ; la fonction principale
  (LET ((ALISTE NIL))       ; la déclaration de la A-liste
       (DECLARE (SPECIAL ALISTE))
       (IF (MATCH-AUX
                ; n'oublions pas de linéariser les arguments
                (LINEAR FILTRE) (LINEAR DONNEE))
           (LIST T ALISTE)  ; le résultat, si ça marchait
           NIL)))           ; le résultat si échec

(DEFUN MATCH-AUX (FILTRE DONNEE)  ; la fonction qui fait le travail
  (DECLARE (SPECIAL DONNEE))      ; DONNEE devient spécial puisque
                                  ; nous l'utiliserons dans DELINEAR-DONNEE
  (LET ((AUX NIL)(AUX1 NIL))
   (DECLARE (SPECIAL AUX AUX1))   ; des spéciales auxiliaires
   (LABELS (

      ; d'abord la fonction traitant les choix multiples
      (CHOIX (AUX FILTRE DONNEE ALISTE2) (COND
          ((NULL AUX) (SETQ ALISTE ALISTE2) NIL) ; pas de succès
          ((MATCH-AUX (CONS (CAR AUX) (CDR FILTRE)) DONNEE) T) ; succès
          (T (CHOIX (CDR AUX) FILTRE DONNEE ALISTE2)))) ; essayons encore

      ; la fonctions traitant les séquences
      (SEQUENCES (ALISTE2 AUX FILTRE DONNEE) (COND
      (DONNEE
         ; le test des contraintes
         (IF (OR (ATOM (CDAR FILTRE))
                 (NULL (CDDAR FILTRE))
                 (EVAL '(LETALL ',ALISTE ,@(CDDAR FILTRE))))
                 ; faisons les comparaisons successives
                 (SETQ AUX1 (MATCH-AUX (CDR FILTRE) DONNEE)))
         (IF AUX1 T
            (PROGN (SETQ ALISTE ALISTE2)
                   (IF AUX (NEXTL-DONNEE)
                       ; ajoutons des liaisons
                       (NCONC (CAR ALISTE)(LIST (NEXTL DONNEE))))
                   (SEQUENCES ALISTE AUX DONNEE))))
      ; échec total : restaurons la A-liste
      (T (IF AUX (NEXTL ALISTE)) NIL))))
```

```
; le corps de la fonction MATCH-AUX
(COND
   ; deux tests d'arrêt de la récursion
   ((NULL FILTRE) (NULL DONNEE))
   ((ATOM FILTRE)(EQUAL FILTRE DONNEE))

   ; est-ce une constante ?
   ((ATOM (CAR FILTRE))
      (AND (EQUAL (CAR FILTRE)(CAR DONNEE)) ; si égale
         (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))) ; continuons

   ; une variable de séquence
   ((EQUAL (CAAR FILTRE) '|?|) (COND
      ; y a-t-il déjà une valeur ?
      ((SETQ AUX (ASSOC (CDAR FILTRE) ALISTE))
        ; oui, comparons la avec la donnée
        (MATCH-AUX (APPEND (LINEAR (CDR AUX))(CDR FILTRE)) DONNEE))
      ; est-ce la fin du filtre ?
      ((NULL (CDR FILTRE))
        ; succès, alors faut-il ou non faire les liaisons
        (IF (EQUAL (CDAR FILTRE) '|-|) T
            (PROGN ; procédons au rajout de la liaison
              (SETQ ALISTE
                (CONS
                  (CONS (VAR (CDAR FILTRE))
                        (DELINEAR-DONNEE))
                  ALISTE))
              (IF (OR (ATOM (CDAR FILTRE))
                      (NULL (CDDAR FILTRE))
                      (EVAL `(LETALL ',ALISTE ,@(CDDAR FILTRE))))
                  T ; tout va bien
                  ; les contraintes ne sont pas satisfaites : échec
                  (PROGN (NEXTL ALISTE) NIL)))))
      (T (IF (SETQ AUX (EQUAL (CDAR FILTRE) '|-|)) NIL
             ; commençant par la séquence vide
             (SETQ ALISTE (CONS (LIST (VAR (CDAR FILTRE))) ALISTE)))
         ; SEQUENCE se chargera du reste
         (SEQUENCES ALISTE AUX DONNEE))))

   ; une donnée atomique ? si oui, c'est un échec
   ((AND DONNEE (ATOM DONNEE)) ())

   ; une variable d'élément
   ((EQUAL (CAAR FILTRE) '|$|) (COND
      ((NULL DONNEE) NIL) ; plus de donnée : échec
      ((EQUAL (CDAR FILTRE) '|-|) ; une variable anonyme ?
         ; oui, alors continuons à comparer le reste
         (MATCH-AUX (CDR FILTRE) (PROGN (NEXTL-DONNEE) DONNEE)))
```

```
            ; regardons si la variable a une liaison
            ((SETQ AUX (ASSOC (VAR (CDAR FILTRE)) ALISTE))
                ; oui, alors comparons la valeur avec la donnée
                (MATCH-AUX
                    (IF (ATOM (CDR AUX))
                        (CONS (CDR AUX)(CDR FILTRE))
                        (APPEND (LINEAR (LIST (CDR AUX)))(CDR FILTRE)))
                    DONNEE))
            ; s'il n'y a pas encore de liaison
            (T (SETQ ALISTE
                    ; il faut la construire
                    (CONS (CONS (VAR (CDAR FILTRE)) (NEXTL-DONNEE)) ALISTE))
               ; est-ce qu'il y a des contraintes ?
               (IF (OR (ATOM (CDAR FILTRE)) (NULL (CDDAR FILTRE))
                       ; et sont-elles satisfaites ?
                       (EVAL '(LETALL ',ALISTE ,@(CDDAR FILTRE))))
                   ; oui, continuons alors le filtrage
                   (MATCH-AUX (CDR FILTRE) DONNEE)
                   ; non, restaurons la A-liste et retournons l'échec
                   (PROGN (NEXTL ALISTE) NIL)))))

        ; un filtre de choix multiple
        ((EQUAL (CAAR FILTRE) '|%|)
                ; déléguons le travail à la fonction CHOIX
                (CHOIX (CDAR FILTRE) FILTRE DONNEE ALISTE))

        ; le cas du filtre commençant par une sous-liste
        ; ne peut plus se produire (à cause de la linéarisation)
        ; nous avons donc examiné tous les cas possibles.
    ))))

(DEFUN VAR (FILTRE)      ; pour trouver la variable d'un filtre
    (IF (ATOM FILTRE) FILTRE
        (IF (ATOM (CDR FILTRE)) (CDR FILTRE) (CAR FILTRE))))

(DEFUN LINEAR (DONNEE) (COND   ; pour linéariser les données et les filtres
   ((ATOM DONNEE) DONNEE)      ; rien à linéariser
   ((ATOM (CAR DONNEE))        ; toujours rien à linéariser
     (CONS (CAR DONNEE) (LINEAR (CDR DONNEE))))  ; regardons dans la suite
   ((MEMBER (CAAR DONNEE) '(|$| |?| |%|))
     ; les filtres ne doivent pas être linéarisés (ce sont des programmes)
     (CONS (CAR DONNEE) (LINEAR (CDR DONNEE))))
   ; enfin, remplaçons la sous-liste "(A B)" par la suite "<< A B >> "
   (T (APPEND (CONS '|<<| (LINEAR (CAR DONNEE))) ; et récursivement de même
       (CONS '|>>| (LINEAR (CDR DONNEE)))))))


(DEFUN DELINEAR-DONNEE () (COND ; pour reconstruire des vraies listes
```

```
  ((ATOM DONNEE) DONNEE)      ; rien à faire
  ((EQUAL (CAR DONNEE) '|<<|) ; aha, une sous-liste
     (NEXTL DONNEE)           ; éliminons ce signe
     ; et construisons une nouvelle sous-liste
     (CONS (DELINEAR-DONNEE)(DELINEAR-DONNEE)))
  ((EQUAL (CAR DONNEE) '|>>|) ; fin de la sous-liste
     (NEXTL DONNEE) NIL)      ; il suffit d'enlever ce signe
  (T (CONS (NEXTL DONNEE) (DELINEAR-DONNEE)))))

(DEFUN NEXTL-DONNEE () (COND ; pour bien trouver l'élément suivant
  ((EQUAL (CAR DONNEE) '|<<|) (NEXTL DONNEE) (DELINEAR-DONNEE))
  (T (NEXTL DONNEE))))

; les macro-caractères

; variable d'élément
(SET-MACRO-CHARACTER #\$
    #'(LAMBDA (FLUX CARACTERE) (CONS '|$| (READ FLUX T NIL T))))

; variable de segment
(SET-MACRO-CHARACTER #\?
    #'(LAMBDA (FLUX CARACTERE) (CONS '|?| (READ FLUX T NIL T))))

; choix multiples
(SET-MACRO-CHARACTER #\%
    #'(LAMBDA (FLUX CARACTERE) (CONS '|%| (READ FLUX T NIL T))))

; finalement, pour mémoire, encore une fois la fonction LETALL
(DEFMACRO LETALL (&REST CALL)
    (LET ((ALISTE (EVAL (CAR CALL))))
        `((LAMBDA ,(MAPCAR 'CAR ALISTE) ,@(CDR CALL))
          ,@(MAPCAR '(LAMBDA (X) (LIST 'QUOTE (CDR X))) ALISTE))))
```

Pour conclure ce chapitre, construisons un autre interprète de règles utilisant cet algorithme de filtrage. Ce programme utilise un ensemble de règles de la forme :

(*<filtre> <action>*)

où la partie *filtre* détermine *quand* il faut lancer une activité, et la partie *action* détermine *que faire* avec la donnée qui correspond au *filtre*.

Si nous appliquons notre interprète à la tâche de simplifier des expressions algébriques, un exemple de filtre peut être :

($X + 0)

et la partie action correspondant peut alors être **X**, c'est une règle disant que **0** est élément neutre à droite pour l'addition.

L'activité de l'interprète se réduit à appliquer l'ensemble de règles jusqu'à ce que le résultat de cette application soit identique à l'expression originale, c.a.d.: jusqu'à ce que plus aucune règle ne puisse s'appliquer.

Voici la boucle top-level de notre interprète :

```
(DEFUN INTERPRETE (EXPR &OPTIONAL EXP1 X) (COND
        ((EQUAL EXP1 EXPR) EXPR)
        (T (SETQ X (SIMPL EXPR REGLES))
           (INTERPRETE X EXPR))))
```

La première clause de cette fonction est la clause d'arrêt : l'interprète cesse d'appliquer l'ensemble des règles de réécritures quand l'expression calculée par la fonction auxiliaire SIMPL est égale à l'expression donnée à cette fonction.

La variable globale **REGLES** contient l'ensemble des règles. Si nous prenons l'exemple d'un simplificateur algébrique simple (i.e.: ne fonctionnant que sur des expressions algébriques complètement parenthèsées), l'ensemble des règles peut alors être la liste suivante :

```
(DEFVAR REGLES '(
        (($(X (NUMBERP X)) + $(Y (NUMBERP Y))) (+ X Y))
        (($(X (NUMBERP X)) * $(Y (NUMBERP Y))) (* X Y))
        (($(X (NUMBERP X)) - $(Y (NUMBERP Y))) (- X Y))
        (($(X (NUMBERP X)) / $(Y (NUMBERP Y))) (/ X Y))
        (($X + 0)  X)
        (($X - 0)  X)
        ((0 + $X)  X)
        ((0 - $X)  (LIST '- X))
        (($X * 1)  X)
        ((0 * $X)  0)
        (($X * 0)  0)
        ((1 * $X)  X)
        (($X / 1)  X)
        (($X + $X) (LIST 2 '* X))
        (($X / $X) 1)
))
```

Mais il nous manque encore la fonction **SIMPL** qui doit effectivement balayer *la liste des règles et, si une s'applique, livrer la réécriture correspondante* et relancer l'ensemble des règles sur cette réécriture :

```
(DEFUN SIMPL (EXPR REGLES-NON-ESSAYEES &OPTIONAL ALISTE) (COND
      ((NULL EXPR) NIL)
      ((ATOM EXPR) EXPR)
      ((CAR (SETQ ALISTE (MATCH (CAAR REGLES-NON-ESSAYEES) EXPR)))
         (SIMPL
            (EVAL '(LETALL ',(CADR ALISTE) ,(CADAR REGLES-NON-ESSAYEES)))
            REGLES))
      ((AND REGLES-NON-ESSAYEES (SIMPL EXPR (CDR REGLES-NON-ESSAYEES))))
      (T (CONS (SIMPL (CAR EXPR) REGLES)
               (SIMPL (CDR EXPR) REGLES)))))
```

C'est tout !

Voici une interaction avec ce mini-simplificateur :

```
? (INTERPRETE '(4 * (5 + 2)))
=28
? (INTERPRETE '(4 * (A * 1)))
=(4 * A)
? (INTERPRETE '((A + 0) * ((B / B) * 1))))
=A
? (INTERPRETE '((A * (3 * 3)) - ((B + B) / (B - 0))))
=((A * 9) - ((2 * B) / B))
```

Si nous ajoutons à notre ensemble de règles les deux règles suivantes :

```
(((=X * =Y) / =Y) X)
(((=Y * =X) / =Y) X)
```

le dernier test nous livre alors :

```
? (INTERPRETE '((A * (3 * 3)) - ((B + B) / (B - 0))))
=((A * 9) - 2)
```

Ainsi, l'extension de cet interprète se fait simplement par adjonction de nouvelles règles. Bien sûr, notre programme d'interprétation de règles n'est pas très efficace, néanmoins il est très général et peut s'appliquer à tout algorithme qui s'exprime par un ensemble de règles de réécriture : il suffit de changer l'ensemble des règles. Les améliorations possibles consistent à :

1. adjoindre la possibilité de grouper des règles afin de ne pas balayer l'ensemble de toutes les règles à chaque itération,

2. adjoindre la possibilité de modifier l'ordre des règles suivant les données pour accélerer la recherche d'une règle qui s'applique.

3. et, finalement, associer à chaque règle plusieurs filtres et plusieurs actions.

20. UN MINI-PROJET

Dans ce dernier chapitre nous allons développer un programme générant des théorèmes sur des programmes. Pendant l'écriture de ce programme, nous utiliserons l'ensemble des méthodes de programmation LISP rencontrées dans ce livre.

Commençons par une description du projet, par une spécification des activités que nous attendons de notre programme.

20.1 QUELQUES REMARQUES SUR LA CORRECTION DE PROGRAMMES

La programmation est, comme chacun sait, une activité très difficile et sujette à beaucoup d'erreurs. Quelques programmeurs sont convaincus que la partie la plus difficile et la plus coûteuse en temps pendant le développement d'un programme, sont ces longue periodes de recherche d'erreurs et de correction des erreurs. Une étude dans l'armée américaine montrait dans les années 70, qu'approximativements 90% du temps des programmeurs était consacré à ces activités, et seulement 10% du temps était nécessaire pour le développement initial d'un programme. Même si l'on pensait que les programmeurs de l'armée américaine étaient particulièrement mauvais – ce qui n'est pas le cas – ce rapport entre temps de développement et temps de correction d'un programme se confirme également dans d'autres contextes de production professionelle de logiciels.

Une fois les programmes *corrigés*, ils contiennent toujours des erreurs. En général, une bonne hypothèse semble être de penser que *tout* programme est erroné. Pour illustrer ce propos, voici un exemple particulièrement coûteux : une des sondes Voyager, prévue pour explorer le système solaire, sortit hors du système solaire dans l'espace interstallaire seulement à cause d'une erreur mineure (un *bug* comme disent les informaticiens) dans le programme de guidage.

Bien entendu, c'est une situation très désagréable pour tout commanditaire de programme : il paye beaucoup d'argent pour le développement d'un programme et, finalement, ce programme contient des erreurs. Ce n'est évidemment pas bon ! Tout un ensemble de systèmes de correction d'erreurs, de vérification de programme, de méthodologie de programmation a été développé pour remédier à ce problème, pour l'instant toujours avec un succès très limité. Aujourd'hui encore, un des domaines de recherche en informatique des plus actifs est celui des *environnements de programmation* et des *environnements de génie logiciel*, qui ont pour buts de concevoir et d'implémenter des systèmes facilitant la programmation (la recherche d'erreurs, la vérification de programmes, la ré-utilisation correcte de programmes, la mise au point de programmes, etc.) tout en assurant une plus grande correction des programmes produits.

La logique de Hoare[1] s'inscrit dans cette lignée de recherche. Elle propose une méthode pour

faire correspondre des expressions d'une logique formelle avec des fragments de programmes et de combiner ces expressions pour arriver à des expressions plus complexes, jusqu'à arriver à une expression logique décrivant le comportement d'un programme en entier. Etant donné que cette expression logique doit traduire la *sémantique* du programme, une vérification ou une falsification de cette expression logique (en utilisant des méthodes de la logique formelle) vérifierait ou falsifierait le programme correspondant. Naturellement, dans la réalité de la programmation quotidienne, cette méthode s'est révélée d'une utilité plutôt limitée. Néanmoins, elle indique une voie possible et peut toujours servir au moins pour des programmes qui ont été générés automatiquement.

Ce que nous allons faire dans ce chapitre, c'est automatiser une petite partie d'un tel système de correspondance entre logique et programme. Plus précisément, nous allons écrire un programme qui génère des expressions logiques, donc des théorèmes logiques, à partir d'un programme donné.

20.2 LA LOGIQUE DE HOARE

Regardons donc comment fonctionne cette logique de programme. Tout d'abord notons que pour Hoare un programme se compose de *trois* parties :

1. un programme *prog*,
2. une *assertion d'entrée* **P**, et
3. une *assertion de sortie* **Q**.

L'énoncé d'un programme sera donc toujours de la forme :

$$\mathbf{P}\ \{prog\}\ \mathbf{Q}$$

L'*assertion d'entrée* **P** exprime les conditions qui doivent être satisfaites *avant* le lancement du programme *prog*. Sans la satisfaction de ces conditions il n'y a pas de sens à lancer le programme. Un exemple d'une assertion d'entrée informelle pour la fonction **REVERSE** peut être, par exemple, la condition que *l'argument doit être une séquence*. Bien entendu, il n'y a aucun sens à demander le **REVERSE** d'un nombre. L'assertion d'entrée est une *hypothèse* qui doit être satisfaite avant le lancement du programme correspondant. Le programme *prog* est, naturellement, un programme dans un langage de programmation donné. L'*assertion de sortie* exprime, en terme d'arguments du programme, l'activité de celui-ci. Elle exprime les transformations que le programme apporte aux données initiales.

Pour Hoare, la correction d'un programme s'exprime donc, pour

1. Cette logique est due à C. A. R. Hoare, Professeur d'Informatique à l'Université de Belfast en Irlande. Il est connu pour ces travaux avec Nicolaus Wirth et Edsgar Dijkstra autour du langage Pascal, pour ses travaux autour de la sémantique axiomatique de programmes – dont nous donnons un court aperçu ici – et pour ses travaux concernant la programmation parallèle. La logique de programmation traîtée ici est très bien exposée dans [Hoare69].

P {*prog*} **Q**

de la manière suivante : sous l'hypothèse que la condition **P** soit satisfaite, le programme *prog* transforme les données initiales de manière telle que la condition **Q** soit satisfaite.

Pour n'en prendre qu'un petit exemple, supposons que nous ayons le programme *prog* suivant :

{(SETQ Z A)(SETQ A B)(SETQ B Z)}

C'est une suite d'instructions qui échange les valeurs des variables **A** et **B** en utilisant une variable auxiliaire **Z**. Une assertion d'entrée **P** possible est :

(A = m *et* B = n)

où *et* est le *et logique* et **m** et **n** sont des expressions désignant des objets arbitraires. Cette assertion se lit : supposons que **A** ait la valeur **m** et **B** ait la valeur **n** avant le lancement du programme. Une assertion de sortie **Q** pour ce programme et cette assertion d'entrée **P** est :

(A = n *et* B = m)

disant que la variabe **A** a la valeur **n** et **B** a la valeur **m** après exécution du programme correspondant. Bien entendu, ces assertions ne font rien d'autre qu'exprimer que le programme *prog* échange le contenu des deux variables **A** et **B**. Mais, elles l'expriment d'une manière logique formelle, donc d'une manière qui est – comme nous le verrons par la suite – manipulable par programme.

Pour l'instant nous savons ce qu'est un programme : une assertion d'entrée, le code d'un programme et une assertion de sortie. Examinons maintenant comment on manipule cette chose.

Initialement, une des idées de Hoare était de créer un langage de programmation *sûr*, c'est-à-dire : un langage permettant une vérification aisée de tout programme écrit dans ce langage. Ceci implique que le langage en question ne devait comporter que des instructions aisément vérifiable. A chacune des instructions de base devait alors être associé un *axiome* exprimant, d'une manière logiquement correcte, l'activité de cette instruction. L'activité d'une instruction était bien sûr exprimée en terme d'assertions d'entrée et de sortie. Hoare lui-même utilisait un sous-ensemble du langage Pascal comme langage de programmation. Nous allons créer un pseudo-langage (ressemblant à LISP) intégrant quelques instructions que Hoare prenait en considération.

Tous les axiomes de Hoare sont exprimés comme un ensemble de règles indiquant comment générer un théorème (ou un ensemble de théorèmes) dont la vérification implique la correction du programme correspondant. Ces règles ont (presque) toujours la forme générale suivante :

$$\frac{\textbf{P}\ \{\textit{programme après simplification}\}\ \textbf{Q'}}{\textbf{P}\ \{\textit{programme actuel}\}\ \textbf{Q}}$$

où **P** est une assertion d'entrée, **Q** une assertion de sortie, *programme actuel* est le programme

qu'on veut vérifier, *programme après simplification* est le programme obtenu après éliminiation de l'instruction qui a engendré l'application de la règle, et **Q'** est l'assertion de sortie obtenue après application de la règle.

20.2.1 La règle d'affectation

Tout cela deviendra plus clair après avoir vu notre première illustration d'une règle de Hoare : (notre adaptation de) la *règle d'affectation*. Traditionnellement, elle s'écrit de la manière suivante :

règle d'affectation :

$$\frac{\mathbf{P}\ \{\mathbf{A}\}\ \mathbf{Q(e)}}{\mathbf{P}\ \{\mathbf{A}\ (\mathbf{SETQ\ x\ e})\}\ \mathbf{Q(x)}}$$

Cette règle se lit de la manière suivante : si nous avons une assertion d'entrée P pour un programme dont la dernière instruction est une affectation, et une assertion de sortie Q exprimée en termes de x, la variable qui change de valeur dans cette dernière instruction, alors cette même assertion P et l'assertion de sortie Q dont on a remplacé toutes les occurrences de la variable x par l'expression e (la nouvelle valeur) doit toujours être valide pour le programme obtenu en enlevant la dernière instruction (l'instruction d'affectation). Reprenons alors l'exemple ci-dessus, le programme échangeant le contenu de deux variabbles :

{(SETQ Z A)(SETQ A B)(SETQ B Z)}

et soit l'assertion d'entrée P égale à :

(A = m *et* B = n)

et l'assertion de sortie égale à :

(A = n *et* B = m)

A ce programme s'applique bien la *règle d'affectation*, ce qui donne le programme suivant :

{(SETQ Z A)(SETQ A B)}

l'assertion d'entrée P est toujours égale à :

(A = m *et* B = n)

et l'assertion de sortie est maintenant égale à

(A = n *et* Z = m)

Si après exécution du programme initial la variable **B** doit contenir la valeur **m**, bien entendu, après exécution du programme où on a éliminé la dernière instruction, qui affecte la valeur de **Z**

à **B**, ce doit bien être **Z** qui contient la valeur **m**.

D'ailleurs, nous pouvons continuer à appliquer cette même règle, puisque la nouvelle dernière instruction du programme est encore une affectation. Ce qui donne le nouveau programme :

$$\{(\mathbf{SETQ\ Z\ A})\}$$

toujours la même assertion d'entrée P :

$$(\mathbf{A = m}\ \textit{et}\ \mathbf{B = n})$$

et la nouvelle assertion de sortie :

$$(\mathbf{B = n}\ \textit{et}\ \mathbf{Z = m})$$

Etant donné que nous avons toujours un programme dont la dernière instruction est une affectation, cette même règle d'affectation peut encore s'appliquer, et nous obtenons alors le programme :

$$\{\}$$

c'est-à-dire : le programme vide, toujours la même assertion d'entrée P :

$$(\mathbf{A = m}\ \textit{et}\ \mathbf{B = n})$$

et la nouvelle assertion de sortie :

$$(\mathbf{B = n}\ \textit{et}\ \mathbf{A = m})$$

20.2.2 la première règle de conséquence

A ce nouveau programme nous ne pouvons plus appliquer la règle d'affectation. Par contre, nous pouvons y appliquer une autre règle, la *première règle de conséquence* qui est la suivante :

règle de conséquence (1) :

$$\frac{\mathbf{P \Rightarrow Q}}{\mathbf{P\ \{NULL\}\ Q}}$$

qui dit : si l'on a une assertion d'entrée P et une assertion de sortie Q qui s'appliquent à un programme vide, alors, pour vérifier que le programme est correct, il suffit de vérifier que

$$P \Rightarrow Q$$

Le signe $\Rightarrow$ représente l'implication logique. Si nous appliquons cette règle à l'état actuel de notre programme, nous obtenons l'expression logique :

$$(A = m \textit{ et } B = n) => (B = n \textit{ et } A = m)$$

qui, sachant que *et* est une opération commutative, est une tautologie.[2]

Arrêtons-nous un instant et examinons ce que nous avons fait. Nous sommes partis d'un programme échangeant le contenu de deux variables, **A** et **B**, accompagné d'une assertion d'entrée et d'une assertion de sortie. Par des applications répétitives de la règle d'affectation et une application de la première (il y en a d'autres !) règle de conséquence, nous sommes arrivés à une expression logique indépendante de tout programme. D'une certaine manière tout se passe comme si nous avions *interprété* le programme sur des données abstraites (ou logiques) spécifiées par les assertions. Le résultat de cette interprétation est une suite de transformations de l'assertion de sortie initiale vers l'assertion d'entrée.

Si nous n'avions pas interprété le programme en commençant par la fin, mais par le début, nous aurions dû changer nos règles de manière telle que l'assertion d'entrée aurait été transformée en une forme équivalente de l'assertion de sortie.

Dans tous les cas, ce que nous avons fait ici, c'est d'avoir tourné à la main le programme que nous avons l'intention de décrire dans ce chapitre. On voit déjà apparaitre la structure du programme :

> Le programme doit appliquer des règles tant que c'est possible, c'est-à-dire : jusqu'à ce que le programme donné initialement soit transformé en programme vide. A l'instant où nous avons un programme vide, nous avons généré un théorème dont la vérification implique la correction du programme initial *par rapport aux assertions initiales.*
>
> Chaque règle est composée de deux parties : l'une décrivant la structure du programme sur laquelle cette règle peut s'appliquer et l'autre décrivant les actions à entreprendre.

Nous avons déjà rencontré une telle structure de programme : c'était notre simplificateur algébrique avec lequel nous avons terminé le chapitre précédent. A priori, l'écriture du programme ne devrait pas être trop difficile.

Mais avant de commencer l'écriture réelle du programme continuons d'abord avec l'énumération des règles de Hoare.

20.2.3 La règle du conditionnel

Examinons une règle légèrement plus complexe, la *règle du conditionnel* :

2. Une tautologie est une expression logique qui est toujours vraie.

règle du conditionnel :

$$\frac{\mathbf{P}\ \{\mathbf{A}\ \mathbf{Q\text{-}if}\ \mathbf{B}\}\ \mathbf{R},\ \mathbf{P}\ \{\mathbf{A}\ (\mathbf{non}\ \mathbf{Q\text{-}if})\ \mathbf{C}\}\ \mathbf{R}}{\mathbf{P}\ \{\mathbf{A}\ (\mathbf{IF}\ \mathbf{Q}\ \mathbf{B}\ \mathbf{C})\}\ \mathbf{R}}$$

Cette règle dit que si vous avez un programme qui se termine par un appel de la fonction **IF** avec un test **Q** une action si-vraie **B** et une action si-fausse **C**, ainsi qu'une assertion d'entrée **P** et une assertion de sortie **R**, afin de le vérifier il suffit de vérifier :

1. que l'assertion **P** implique l'assertion **R**, après exécution de la partie du programme qui précède le **IF** et, en supposant que le test du **IF** soit vrai, après exécution de l'instruction **B**, et

2. que l'assertion **P** implique l'assertion **R** également après exécution de la partie du programme qui précède le **IF** et, en supposant que le test du **IF** soit faux, après exécution de l'instruction **C**.

C'est une règle qui va générer deux théorèmes, correspondant aux deux parcours possibles du **IF** : un parcours pour la branche vraie et un parcours pour la branche fausse.

20.2.4 La deuxième règle de conséquence

Cette règle engendre une autre règle, la *deuxième règle de conséquence*, qui ne sert qu'à indiquer ce qu'il faut faire avec l'expression **Q-if** :

règle de conséquence (2) :

$$\frac{\mathbf{P}\ \{\mathbf{A}\}\ \mathbf{Q} \Longrightarrow \mathbf{R}}{\mathbf{P}\ \{\mathbf{A}\ \mathbf{Q\text{-}if}\}\ \mathbf{R}}$$

Prenons un programme et regardons ce que font ces règles. Voici le programme et ses assertions :

$$\{(\mathbf{IF}\ (< \mathbf{X}\ \mathbf{0})\ (\mathbf{SETQ}\ \mathbf{Y}\ (\text{-}\ \mathbf{0}\ \mathbf{X}))\ (\mathbf{SETQ}\ \mathbf{Y}\ \mathbf{X}))\}$$

assertion d'entrée :
(entier(X))

assertion de sortie :
$(\mathbf{Y} \geq \mathbf{0})$

La première règle qui s'applique est la règle du conditionnel. Son application nous génère les deux continuations suivantes possibles (les assertions étant inchangées, nous ne les reproduisons pas) :

1) $\{(< \mathbf{X}\ \mathbf{0})_{\text{-if}}\ (\mathbf{SETQ}\ \mathbf{Y}\ (\text{-}\ \mathbf{0}\ \mathbf{X}))\}$

2) $\{(\mathbf{non}\ (< \mathbf{X}\ \mathbf{0}))_{\text{-if}}\ (\mathbf{SETQ}\ \mathbf{Y}\ \mathbf{X})\}$

Développons d'abord le premier programme. Il se termine par une instruction d'affectation, c'est donc la règle d'affectation qui s'applique. Ce qui donne :

$$\{(< X\ 0)_{-if}\}$$

assertion d'entrée :
(entier(X))

assertion de sortie :
$((-\ 0\ X) \geq 0)$

et l'application de la deuxième règle de conséquence transforme ceci en :

$$\{\}$$

assertion d'entrée :
(entier(X))

assertion de sortie :
$(< X\ 0) \Longrightarrow ((-\ 0\ X) \geq 0)$

qui engendrera, après application de la première règle de conséquence, notre premier théorème :

$$\textbf{(entier(X))} \Longrightarrow ((< X\ 0) \Longrightarrow ((-\ 0\ X) \geq 0))$$

La deuxième continuation engendrée par la règle du conditionnel était :

$$\{(\textbf{non}\ (< X\ 0))_{-if}\ \textbf{(SETQ Y X)}\}$$

Elle se termine également par un appel de SETQ, c'est donc encore la règle d'affectation qui s'applique. Ce qui donne :

$$\{(\textbf{non}\ (< X\ 0))_{-if}\}$$

assertion d'entrée :
(entier(X))

assertion de sortie :
$(X \geq 0)$

qui se transformera par l'application de la deuxième règle de conséquence en :

$$\{\}$$

assertion d'entrée :
(entier(X))

assertion de sortie :
$\textbf{(non}\ (< X\ 0)) \Longrightarrow (X \geq 0)$

qui sera finalement, par la première règle de conséquence, transformé sous la forme du

théorème que voici :

$$\mathbf{(entier(X)) \Longrightarrow ((non\ (< X\ 0)) \Longrightarrow (X \geq 0))}$$

La vérification de la conjonction de ces deux théorèmes générés pourrait nous assurer que le programme initial est correct par rapport aux assertions initiales.

Bien entendu, ces deux théorèmes ne sont pas très compliqués à vérifier : ce sont encore des tautologies.

Insistons sur le fait que ce que nous vérifions avec les règles de Hoare, c'est la correction d'un programme par rapport à ses assertions. Si nous arrivons à vérifier un programme par rapport à des assertions fausses, nous pouvons conclure que le programme est erroné. Si nous arrivons à démontrer qu'un théorème généré est faux, nous pouvons conclure que le programme est faux par rapport aux assertions qui l'accompagnent. Des problèmes surgissent dès que nous n'arrivons plus à démontrer ni la correction, ni la fausseté des théorèmes générés. Dans ces cas, nous ne pouvons rien déduire et plusieurs interprétations de la défaillance sont possibles :

1. soit le programme est erroné,
2. soit il y a une erreur dans une (ou les deux) assertion(s),
3. soit il existe une erreur dans notre démonstration,
4. soit nos capacités de démonstration sont limitées.

Néanmoins, cela ne limite pas la puissance de cette méthode. Les mêmes observations peuvent être faites sur *tout* système de vérification de programme. Je m'imagine que dans un futur pas trop lointain, les machines utiliseront de tels mécanismes automatiquement, sans que nous nous en apercevions, pour augmenter leur confiance dans nos programmes. Je pourrais même imaginer un système qui, quand une erreur se produit, nous renvoie vers des parties du programme lui semblant faibles puisqu'il ne sera pas arrivé à démontrer leur correction. A partir d'un tel système, on ne sera pas loin d'arriver à un devéloppement ultérieur qui utilisera cette vérification ou son impossibilité pour améliorer ou corriger automatiquement nos programmes.

20.2.5 La règle de l'itération

Revenons à la description des règles de Hoare. La *règle d'itération* est la suivante :

règle d'itération :

$$\frac{\mathbf{P\ \{A\}\ R,\ R}\ \textit{et}\ \mathbf{S\ \{B\}\ R,\ R}\ \textit{et}\ \mathbf{(non\ S) \Longrightarrow Q}}{\mathbf{P\ \{A\ (assert:\ R\ (WHILE\ S\ B))\}\ Q}}$$

Cette règle est un peu plus compliquée. Notons tout d'abord que **R** est une assertion supplémentaire demandée pour toute itération. C'est une assertion qui doit être vraie chaque fois qu'on parcourt la boucle. C'est pour cela qu'elle s'appelle une *assertion d'invariance* ou un

invariant de boucle.

La règle dit que si nous avons un programme terminé par une itération **WHILE** qui obéit à l'assertion d'invariance **R**,[3] afin de vérifier ce programme, par rapport à ses assertions, **P** et **Q**, il faut vérifier trois choses :

1. d'abord que l'assertion d'entrée **P**, suivie de l'exécution du bout de programme précédant l'itération, doit impliquer l'assertion d'invariance **R**,

2. ensuite que suivant l'hypothèse que l'assertion **R** soit correcte (nous venons de le démontrer) et suivant l'hypothèse que le test **S** du **WHILE** soit également vrai, l'exécution du corps **B** de la boucle doit impliquer l'assertion d'invariance **R**. Intuitivement cela devrait être le cas, puisque l'assertion **R** doit être vraie à chaque itération.

3. Finalement, nous devons vérifier la correction de l'implication :

$$\mathbf{R}\ et\ (\mathbf{non\ S}) \Longrightarrow \mathbf{Q}$$

Après un instant de reflexion, tout cela semble bien évident.[4] Regardons néanmoins un exemple. Mais cette fois-ci nous n'allons donner que les parties principales. Prenons le programme suivant :

```
{(SETQ Y 0)
 (assert: ((X + Y) = m)
    (WHILE (> X 0)
           (SETQ X (- X 1))
           (SETQ Y (+ Y 1)))))}
```

avec l'assertion d'entrée :

$$(\mathbf{entier(m)}\ et\ \mathbf{m} \geq \mathbf{0}\ et\ \mathbf{X} = \mathbf{m})$$

et l'assertion de sortie :

$$(\mathbf{Y} = \mathbf{m})$$

C'est un programme de transfert du contenu numérique de **X** dans **Y**. La première règle qui s'applique à ce programme est la *règle de l'itération* qui donne les trois branches que voici :

1. le programme :

```
{(SETQ Y 0)}
```

3. Pour des boucles d'une certaine complexité, trouver une assertion d'invariance est loin d'être facile. C'est pourquoi quelques informaticiens ont pensé automatiser également ce processus, donc générer automatiquement des invariants de boucles. Un des meilleurs systèmes de ce type a été développé par Patrick Cousot [Cousot77].

4. C'est toujours les théories qui semblent évidentes qui sont les plus intéressantes !

avec l'assertion d'entrée initiale et l'assertion de sortie :

$$((X + Y) = m)$$

2. le programme :

$$\{(SETQ\ X\ (-\ X\ 1))(SETQ\ Y\ (+\ Y\ 1))\}$$

avec l'assertion d'entrée :

$$((X + Y) = m)\ et\ (> X\ 0)$$

et l'assertion de sortie :

$$((X + Y) = m)$$

3. et finalement le théorème :

$$(((X + Y) = m)\ et\ (non\ (> X\ 0))) \Rightarrow (Y = m)$$

Ce dernier théorème semble être encore une tautologie si l'on prend en compte l'assertion d'entrée qui dit que **m** est un nombre positif ou zéro et que **X** a comme valeur **m**. A vous de le vérifier.

La première branche donne après application de la règle d'affectation et de la première règle de conséquence le théorème suivant :

$$(entier(m)\ et\ m \geq 0\ et\ X = m) \Rightarrow ((X + 0) = m)$$

Encore une évidence, c'est-à-dire : une tautologie !

Ne reste que la deuxième branche qui se compose de deux affectations :

$$\{(SETQ\ X\ (-\ X\ 1))(SETQ\ Y\ (+\ Y\ 1))\}$$

qui donnera après deux applications de la règle d'affectation et une application de la première règle de conséquence le théorème que voici :

$$((X + Y) = m)\ et\ (> X\ 0) \Rightarrow (((-\ X\ 1) + (+\ Y\ 1)) = m)$$

La vérification de la conjonction de ces trois théorèmes nous prouve la correction du programme ci-dessus par rapport à ses assertions.

20.2.6 Les règles concernant des procédures

Il nous reste encore deux règles : une qui permet de vérifier les définitions de procédures et l'autre leurs appels. Commençons par la *règle de définition de procédure*. Elle est ici :

règle de définition de procédure :

$$\frac{\mathbf{P}\ \{(\mathbf{q}\ (\mathbf{x})\ (\mathbf{v}))\}\ \mathbf{Q} \vdash \mathbf{P}\ \{\mathbf{A}\}\ \mathbf{Q}}{\mathbf{P}\ \{\mathbf{PROCEDURE}\ \mathbf{q}\ (\mathbf{x})(\mathbf{v})\ \mathbf{A}\}\ \mathbf{Q}}$$

Quelques remarques sur cette règle s'imposent. Il est clair que nous nous éloignons ici du LISP classique : premièrement, en LISP il n'y a pas de procédures mais des fonctions, deuxièmement, en LISP il n'y pas une seule construction qui permette *deux* listes de variables. Ici, toute procédure est censée avoir deux listes de variables (de paramètres). La première liste donne des paramètres habituels, ce derniers recevront des arguments évalués à l'appel que nous appellerons des *paramètres valeurs* (en référence à l'appel par valeur des langages algorithmiques). La deuxième liste de paramètres donne des paramètres où les arguments sont des *adresses* de variables, nous appellerons ces paramètres des *paramètres noms* (en référence à l'appel par nom des langages algorithmiques). Ceci provient du langage Pascal, pour lequel les règles de Hoare ont été développées initialement.

Une expression de la forme :

$$X \vdash Y$$

se lit : Y peut être vérifié en supposant X vrai.

Maintenant nous pouvons paraphraser cette règle. Elle dit : pour vérifier un appel de la procédure q avec les paramètres valeurs x et les paramètres noms $\mathbf{v}$ par rapport à l'assertion d'entrée $\mathbf{P}$ et l'assertion de sortie $\mathbf{Q}$, il suffit de supposer que pour tout appel de la procédure q avec des arguments x et v les assertions sont vraies, et sous cette hypothèse de vérifier ensuite le corps $\mathbf{A}$ de la procédure.

L'hypothèse de la vérification des assertions est nécessaire dans le cas où il y a des appels récursifs de cette procédure à l'intérieur du corps. Bien entendu, ceci correspond exactement à ce que vous faites quand vous écrivez une fonction récursive : quand vous définissez la fonction **REVERSE**, par exemple, vous dites que l'inverse de la liste vide est la liste vide, sinon, sous l'hypothèse que la procédure **REVERSE** fonctionne correctement, vous dites que l'inverse d'une liste est le résultat de l'opération qui colle ensemble la liste obtenue en inversant le CDR de l'argument et la liste obtenue en mettant une paire de parenthèses autour du CAR de l'argument. C'est le même raisonnement ici, seulement c'est un peu plus formel.

La *règle de l'appel de procédure* est comme suit :

règle de l'appel de procédure :

$$\frac{\mathbf{P}\ \{\mathbf{A}\}\ \mathbf{U(a;\ e)}\ \textit{et}\ \#\ \mathbf{a}\ (\mathbf{W(a;\ e)} \Longrightarrow \mathbf{Q})}{\mathbf{P}\ \{\mathbf{A}\ (\mathbf{q}\ (\mathbf{a})(\mathbf{e}))\}\ \mathbf{Q}} \quad \vdash \mathbf{U(x;\ v)}\ \{(\mathbf{q}\ (\mathbf{x})(\mathbf{v}))\}\ \mathbf{W(x;\ v)}$$

Elle a l'air compliquée, mais elle ne l'est pas. Voila ce que ça veut dire : par hypothèse on connaît une définition de la procédure *q* avec une assertion d'entrée **U** (éventuellement en termes de **x** et *v*) et une assertion de sortie *W* (elle aussi éventuellement en termes de **x** et *v*) si l'on rencontre un appel de cette procédure, éventuellement précédé par un fragment de programme **A**, le tout entouré par les assertions **P** et **Q**, alors pour vérifier ce programme, il suffit de vérifier que l'assertion d'entrée **P** implique bien, après exécution du fragment de programme **A**, l'assertion d'entrée de la procédure *q*. Naturellement, il faut adapter l'assertion d'entrée de la procédure à l'appel particulier, donc aux arguments **a** et **e** de l'appel. Etant donné que nous ne pouvons rencontrer un appel de procédure qu'après avoir vu sa définition, nous l'avons déjà vérifié. Nous pouvons donc utiliser cette vérification et il suffit de démontrer que l'assertion de sortie de la procédure implique l'assertion de sortie du programme actuel. N'oublions pas que l'assertion de sortie de la procédure est exprimée en termes généraux. Nous devons l'adapter d'abord aux arguments spécifiques de l'appel en question.

Si cela vous semble un peu obscur, voici un exemple dont l'application des règles devrait éclaircir ces deux dernières règles. Prenons la version de la procédure **FACTORIELLE** que voici :

```
(PROCEDURE FACTORIELLE (N) (R)
   (IF (= N 0) (SETQ R 1)
      (FACTORIELLE (1- N) R)
      (SETQ R (* R N))))
```

et accompagnons la de l'assertion d'entrée :

$$(\mathbf{entier(N)}\ \textit{et}\ (\mathbf{N} \geq \mathbf{0}))$$

et de l'assertion de sortie :

$$(\mathbf{R} = !\mathbf{N})$$

et essayons de vérifier cette procédure. Notons que c'est une procédure récursive, nous avons donc à la fois une définition de procédure et un appel de procédure. Notez également que cette procédure ne marchera pas dans un LISP traditionnel ! La première raison en est que nous avons généralisé la fonction **IF** pour quelle prenne, comme en Le_LISP, une *suite* d'actions si le test est faux, et la deuxième raison est qu'ici cela fonctionne puisque **R** est un pointeur vers une variable, et ce pointeur ne change pas dans les appels successifs. C'est donc une variable externe à la procédure qui recevra la valeur de la factorielle de **N**. Etudiez bien cela, sinon cette procédure est incompréhensible.

La première règle qui s'applique est la *règle de définition de procédure*. elle nous livre le résultat suivant :

$$\begin{array}{l}(\text{entier}(N)\ \textit{et}\ (N \geq 0))\\ \quad \{(\text{FACTORIELLE}\ (N)(R))\}\ (R = !N)\\ \quad \vdash\\ \quad\quad (\text{entier}(N)\ \textit{et}\ (N \geq 0))\\ \quad\quad\quad \{(\text{IF}\ (= N\ 0)\ (\text{SETQ}\ R\ 1)(\text{FACTORIELLE}\ (1\text{-}\ N)\ R)(\text{SETQ}\ R\ (*\ R\ N)))\}\\ \quad\quad\quad (R = !N)\end{array}$$

Afin de vérifier le corps, nous appliquons la *règle du conditionnel*, ce qui donne les deux branches suivantes (étant donné que les assertions ne sont pas modifiées par cette règle, nous ne donnons que les programmes) :

$$1)\ \{(= N\ 0)_{\text{-if}}\ (\text{SETQ}\ R\ 1)\}$$

et le deuxième programme est :

$$2)\ \{(\text{non}\ (= N\ 0))_{\text{-if}}\ (\text{FACTORIELLE}\ (1\text{-}\ N)\ R)(\text{SETQ}\ R\ (*\ R\ N))\}$$

La première branche nous donne, après une application de la règle de l'affectation, une application de la deuxième règle de conséquence et une application de la première règle de conséquence,[5] le théorème suivant :

$$(\text{entier}(N)\ \textit{et}\ (N \geq 0)) \Longrightarrow ((= N\ 0) \Longrightarrow (1 = !N))$$

La deuxième branche nous donne, après l'application de la règle d'affectation le programme :

$$\{(\text{non}\ (= N\ 0))_{\text{-if}}\ (\text{FACTORIELLE}\ (1\text{-}\ N)\ R)\}$$

et la nouvelle assertion de sortie :

$$((*\ R\ N) = !N)$$

Maintenant nous sommes confrontés à un appel de procédure en dernière position du programme. C'est donc la *règle de l'appel de procédure* qui s'applique. Ce qui donne le nouveau programme :

$$\{(\text{non}\ (= N\ 0))_{\text{-if}}\}$$

et la nouvelle assertion de sortie que voici :

$$\begin{array}{l}(\text{entier}((1\text{-}\ N))\ \textit{et}\ ((1\text{-}\ N) \geq 0))\\ \quad\quad \textit{et}\\ \quad\quad\quad \#\ N\ ((R = !(1\text{-}\ N)) \Longrightarrow ((R * N) = !N))\end{array}$$

Notez bien les substitutions que nous avons effectuées : dans l'assertion d'entrée initiale nous avons remplacé toutes les occurrences des paramètres par leurs valeurs d'appels, c'est-à-dire : que nous avons substitué **R** pour toute occurrence de **R** (ça ne changeait pas grand chose) et

5. Vérifiez ces affirmations en appliquant vous-mêmes une règle après l'autre. Entrainez-vous un peu à manipuler cette axiomatique de Hoare.

nous avons substitué l'expression **(1- N)** à toute occurrence de **N**.

Le reste est – entre temps – de la routine. Après une application de la deuxième règle de conséquence, puis une application de la première règle de conséquence nous obtenons le théorème suivant :

$$\begin{array}{l}\textbf{(entier(N)}\ \textit{et}\ \textbf{(N} \geq \textbf{0))} \Longrightarrow \\ \textbf{((non (=N 0))} \Longrightarrow \\ \quad \textbf{(entier((1- N))}\ \textit{et}\ \textbf{((1- N)} \geq \textbf{0))} \\ \qquad \textit{et} \\ \qquad\quad \textbf{\# N ((R =!(1- N))} \Longrightarrow \textbf{((R * N) =!N))))}\end{array}$$

Si nous arrivons à démontrer ce théorème ainsi que le premier théorème généré :

$$\textbf{(entier(N)}\ \textit{et}\ \textbf{(N} \geq \textbf{0))} \Longrightarrow \textbf{((=N 0)} \Longrightarrow \textbf{(1 =!N))}$$

nous pouvons avoir confiance dans la correction de l'algorithme[6] du programme FACTORIELLE. Ces deux démonstrations ne semblent pas trop difficiles non plus – si nous enrichissons notre logique par quelques connaissances arithmétiques supplémentaires.

Ici nous terminons la spécification du programme à écrire : pour l'instant nous avons vu toutes les règles qui nous intéressent. En même temps nous avons vu notre langage auquel ces règles s'appliquent : une forme de pseudo-LISP, avec la même forme extérieure, mais des constructions légèrement différentes et où, au lieu de fonctions, on définit des procédures.

Il est temps alors de voir comment implémenter un programme qui fait automatiquement tout ce que nous avons fait à la main.

20.3 LE PROGRAMME GCV

Après cette longue spécification, nous devrions avoir une idée assez précise de la structure du programme. Tel que le problème se présente, il semble évident que nous allons écrire encore un système qui applique des *règles de réécriture* à la donnée initiale. Quelles sont les données ? D'abord il y a l'ensemble des règles. Celles-ci, nous pouvons les mettre dans une variable globale , ainsi nous assurons leurs accès de tout point du programme. Ensuite il y a le programme,

6. Comme la quasi totalité des systèmes de vérification de programmes, le système de Hoare vérifie – en réalité – plutôt des algorithmes que des programmes. Par exemple, nous sommes bien capables de démontrer la procédure **FACTORIELLE** ci-dessus, mais d'une manière abstraite, éloignée de la réalité informatique quotidienne, c'est-à-dire : loin des contraintes de la machine. Même si l'algorithme est correct, le lancement effectif de la procédure peut toujours produire une erreur : aucune vérification n'a été faite concernant, par exemple, l'espace disque nécessaire ou les possibilités d'une erreur de débordement. C'est un modèle mathématique livrant la possibilité de vérifier des algorithmes (ce qui est déjà très, très utile) mais qui ne sait pas, pour l'instant, traiter des contraintes purement informatiques. Peut-être pouvez vous développer une extension de la théorie prenant en compte les contraintes d'espace mémoire, d'espace disque, la taille des mots, etc., bref les contraintes imposées par l'outil informatique ?

l'assertion d'entrée (la *précondition*) et l'assertion de sortie (la *postcondition*). Ces trois données seront livrées à la routine principale comme de vrais arguments. Nous devons penser également à un endroit où accumuler les théorèmes générés au fur et à mesure de l'application des règles. Ce sera une variable supplémentaire.

Muni de cette petite analyse, nous pouvons écrire le début de la routine principale. Nous allons l'appeler **GCV**, pour *Générateur de Conditions de Vérification*. Voici alors le début de cette procédure :[7]

(DEFUN GCV (PROG PRECONDITION POSTCONDITION &OPTIONAL VCS) ...

La variabl. **PROG** contiendra, à l'appel, le programme à vérifier. Les variables **PRECONDITION** et **POSTCONDITION** prendront, comme leurs noms l'indiquent, la précondition et la postcondition accompagnant le programme. La dernière variable, **VCS**, servira à accumuler les théorèmes au fur et au mesure de leur génération.

Avant de continuer cette fonction, écrivons d'abord les règles de reécriture. Chaque règle sera une liste où le premier élément sera un filtre decrivant le fragment de programme auquel cette règle s'applique et le deuxième élément sera une expression LISP qui décrit ce qu'il faut faire si la règle s'applique.

Commençons par la règle la plus simple : la *première règle de conséquence*. Elle dit que si le programme à analyser est vide, il faut ajouter un nouveau théorème, à la suite de ceux déjà générés, exprimant l'implication de la postcondition actuelle par la précondition actuelle. Ce qui donne la règle suivante :

(() (CONS '(,PRECONDITION => ,POSTCONDITION) VCS))

Bien entendu, il faut ajouter le nouveau théorème à la liste **VCS** qui contient l'ensemble de tout les théorèmes à démontrer pour vérifier le programme donné en argument à **GCV**.

Notez que puisque nous accédons ici dans les règles aux variables **PRECONDITION**, **POSTCONDITION** et **VCS**, il est nécessaire de les déclarer en tant que variables *spéciales*.

Prenons la règle suivante : la *règle d'affectation*. Elle est ici :

7. Ce programme a été inspiré initialement par un programme écrit par Daniel Goossens. Ce programme n'était pas écrit comme un système à règles, mais intégrait les règles directement dans l'algorithme. Il tournait sous VLISP-10, dans sa version de la PDP-10 de l'IRCAM. Goossens l'avait écrit, à l'époque, pour *comprendre* comment les règles de Hoare fonctionnaient. Je suis convaincu qu'un bon informaticien se distingue par cette capacité : écrire des programmes qui l'aident dans la compréhension du fonctionnement de l'objet modélisée dans le programme. D'ailleurs, c'est cela que l'on fait en Intelligence Artificielle.

   Le programme donné ici, est le résultat de moult adaptations, réécritures et modifications. Le programme de Daniel Goossens a été inspiré à son tour par un large système de génération de conditions de vérifications pour des programmes Pascal, écrit par le groupe de recherche autour du Professeur David Luckham de Stanford University. Une très bonne description de ce programme est donnée dans [Igarashi75].

```
( (?A (SETQ $X $E)) (GCV A PRECONDITION (SUBST E X POSTCONDITION) VCS) )
```

La première partie est un filtre, comme ceux que nous avons étudiés au chapitre précédent. Ce filtre s'applique si la liste donnée se termine par un appel de **SETQ**. Dans ce cas, la variable de filtre **A** sera associée à la séquence de toutes les expressions précédant l'appel du **SETQ**, la variable **X** sera associée à la variable donnée en premier argument à **SETQ** et, finalement, la variable **E** sera associée à l'expression qui détermine la nouvelle valeur. Visiblement, la fonction **MATCH** du chapitre précédent est très utile. C'est donc un appel de cette fonction, avec comme filtre le CAR des règles successives et comme donnée le programme, qui déterminera quelle règle il faut appliquer. Bien entendu, ce sera une boucle essayant une règle après l'autre de l'ensemble de nos règles de Hoare. Mettons cet ensemble dans la variable globale **REGLES** avec l'instruction :

```
(DEFVAR REGLES '(
    ; première règle de conséquence
    ( () (CONS '(,PRECONDITION => ,POSTCONDITION) VCS) )

    ; règle d'affectation
    ( (?A (SETQ $X $E))
          (GCV A PRECONDITION (SUBST E X POSTCONDITION) VCS) )

    ..... et ainsi de suite l'ensemble de règles
    .....) )
```

Nous n'avons pas terminé l'examen de la règle d'affectation. Regardons l'appel de **GCV** comme action à faire quand cette règle s'applique. C'est une sorte d'appel récursif *calculé*. Par ce mécanisme nous nous assurons que le programme itère (ou récurse) jusqu'à ce qu'on tombe sur une règle qui ne contient pas d'appel récursif de **GCV** *sans* que nous le spécifions dans le programme. C'est une manière très élégante de faire déterminer le flux de contrôle de l'exécution du programme directement par les données. En informatique, cette manière d'appeler des fonctions s'appelle *le lancement de procédure dirigé par les données.*[8]

Avez-vous remarqué que quelques-uns des arguments de l'appel récursif de **GCV**, dans la partie action de la règle, sont des variables du filtre. Ceci implique que nous devons exécuter cet appel dans un environnement où les variables, qui ont été associées à des valeurs dans la A-liste résultant de l'appel de **MATCH**, sont effectivement liées à ces valeurs. Nous avons rencontré également ce problème dans le chapitre précédent, et nous l'avons résolu avec la macro-fonction **LETALL** (si vous l'avez oublié, retournez examiner cette fonction !). Bien entendu, ceci implique que nous devons déclarer toutes les variables du filtre en tant que variables spéciales.

Revenons maintenant vers notre programme principal **GCV** et complétons le :

8. Dans la littérature anglophone ce processus s'appelle *data driven function invocation.*

```
(DEFUN GCV (PROG PRE POST &OPTIONAL VCS)
   (LABELS ((GCV1 (PROG REGL
                        PRECONDITION
                        POSTCONDITION
                        VCS REGG
                        &OPTIONAL AUX)
                  (DECLARE (SPECIAL AUX PRECONDITION POSTCONDITION VCS)
                  (COND
                     ((NULL REGL)(FORMAT T "erreur dans le programme~%"))
                     ((SETQ AUX (MATCH (CAAR REGL) PROG))
                      ; le CAR de la règle est le filtre ;
                      (EVAL
                        `(LETALL ',(CADR AUX)
                           (P-P-P ',PROG ',REGG)
                           ; le CADR de la règle est l'action
                           (EVAL ',(CADAR REGL))))
                     (T (GCV1 PROG (CDR REGL)
                              PRECONDITION
                              POSTCONDITION
                              VCS
                              (CDR REGG))) )))
    (GCV1 PROG REGLES PRE POST VCS REGL1)))
```

Nous avons introduit une boucle, la fonction **GCV1** qui ne sert qu'à chercher la bonne règle parmi l'ensemble des règles. Nous faisons cela en comparant l'un après l'autre les filtres des règles avec le programme, par des appels successifs de la fonction **MATCH** (définie dans le chapitre précédent). Nous avons également déclaré les variables utilisées dans les règles en tant que variables spéciales. La variable **AUX** est déclarée spéciale à cause de son occurrence dans l'appel de la macro fonction **LETALL**. Elle ne sera évaluée qu'à l'intérieur de cette macro, donc dans une position non-locale.

Nous avons introduit un argument supplémentaire, le paramètre **REGG**. Cette variable n'existe que pour la beauté du programme : elle sera liée à une autre liste de règles de Hoare, la liste **REG1**. Cette deuxième liste de règles contiendra des formes lisibles des règles de Hoare, telles que celles que nous avons données plus haut. Nous nous en servons pour faire des *traces* de l'exécution du programme : ainsi nous pourrons voir, au fur et à mesure de l'exécution, les règles qui s'appliquent. Dans les appels itératifs de **GCV1** nous avançons en parallèle dans cette liste et dans la liste des *vraies* règles, nous assurant ainsi que nous pointons toujours sur la même règle dans **REGG** et dans **REGL** – si nous donnons les règles dans le même ordre dans les deux listes.

Remarquez que l'évaluation de la partie action de la règle choisie se fait à l'intérieur d'un appel de **LETALL**. Ceci nous assure que les diverses variables du filtre sont effectivement liées à la valeur filtrée.

Maintenant construisons d'abord notre liste **REG1**.

```
(DEFVAR REG1 '(

("première règle de conséquence :"
"          P ==> Q
           ----------
          P {NULL} Q" )

("règle de l'affectation:"
"            P {A} Q(e)
           ----------------
          P {A (SETQ x e)} Q(x)")

("règle du conditionnel:"
"          P {A Q-if B} R, P {A (non Q-if) C} R
          ------------------------------------
                P {A (IF Q B C)} R")

("deuxième règle de conséquence :"
"          P {A} Q ==> R
          ----------
          P {A; Q-if} R")

("règle de l'itération:"
"          P {A} R, R et S {B} R, R et (non S) ==> Q
          ----------------------------------------
               P {A (assert: R (WHILE S B))} Q")


("règle de définition de procédure:"
"      P {q (x) (v)} Q ⊢ P {A} Q
      ------------------------
      P {PROCEDURE q (x) (v) A} Q"

("règle de l'appel de procédure:"
"      U (x; v) {q (x) (v)} W (x; v)
            P {A} U (a; e) et # a (W (a; e) ==> Q)
      ⊢    ------------------------------------
                  P {A (q (a) (e))} Q")
) "la forme lisible des règles")
```

Dans la règle d'affectation nous appelons une fonction **SUBST** à trois arguments. La tâche de cette fonction est de remplacer à l'intérieur du troisième argument toutes les occurrences du deuxième argument par le premier. **SUBST** est une fonction LISP standard, qui peut être définie comme suit :

```
(DEFUN SUBST (VAL VAR BOD)(COND
    ((EQUAL VAR BOD) VAL)
    ((ATOM BOD) BOD)
    (T (CONS (SUBST VAL VAR (CAR BOD))(SUBST VAL VAR (CDR BOD))))))
```

Continuons avec l'écriture de nos règles de Hoare. Voici la *règle du conditionnel* :

```
( (?A (IF $R1 $B ?C))
    (GCV (APPEND A '((ASSERT-IF: ,R1) ,B)) PRECONDITION POSTCONDITION
         (GCV (APPEND A '((ASSERT-IF: (NON ,R1)) , C))
              PRECONDITION POSTCONDITION VCS) ))
```

Dans cette règle nous devons développer deux branches : la branche vraie et la branche fausse. Chacune des branches génèrera un théorème. Nous réaliserons cela en donnant un appel récursif – celui pour la branche fausse – à la place de l'argument **VCS** de l'appel récursif extérieur. Ainsi, nous nous assurerons que le théorème correspondant sera bien accumulé dans la variable **VCS**.

Cette règle nécessite la *deuxième règle de conséquence* que voici :

```
( (?A (ASSERT-IF: $B)) (GCV A PRECONDITION '(,B => ,POSTCONDITION) VCS) )
```

Ci-dessous la *règle de l'itération* dans une forme utilisable par le programme :

```
( (?A (ASSERT: $R (WHILE $S ?B)))
    (GCV A PRECONDITION R
         (GCV B '(,R ET ,S) R
              (CONS '(,R ET (NON ,S) => ,POSTCONDITION) VCS))))
```

Cette règle génère trois théorèmes. L'un d'eux est déjà donné tel quel, et nous l'ajoutons directement à **VCS**, les deux autres seront calculés par les deux appels récursifs de **GCV**.

La *règle de définition de procédure* prend la forme suivante :

```
( (PROCEDURE $F $AS $ES ?K)
    (PROGN
         (SETQ REC
               '((,PRECONDITION ,F ,AS ,POSTCONDITION ,ES) , REC))
         (GCV K PRECONDITION POSTCONDITION VCS)))
```

Cette règle fait deux choses : d'abord elle garde dans la liste **REC** l'hypothèse que les assertions sont correctes pour cette procédure (c'est ici que nous allons chercher les assertions associées à une procédure, si nous en rencontrons un appel) et ensuite elle vérifie que le corps satisfait effectivement ces assertions.

Finissons alors avec la *règle de l'appel de procédure* que voici :

```
( (?A ($F ?VS))
    (IF (FIND F REC)
        (GCV A PRECONDITION '(,U ET |#|,AS (,W => ,POSTCONDITION)) VCS)
        ()))
```

Cette règle est complémentaire à la règle de définition de procédure : pendant la définition d'une procédure nous gardons les assertions, les paramètres et le nom de la procédure dans la structure **REC**; ici nous cherchons, avec la fonction **FIND**, si **REC** contient effectivement une référence à la procédure appelée.

Dans la partie action de cette règle nous trouvons des références à des variables **U**, **AS** et **W** qui ne sont pas des variables du filtre. D'où viennent-elles ? Si nous regardons notre version lisible de cette règle, nous voyons que **U** réfère à la précondition de la procédure, **W** réfère à la postcondition de cette même procédure et **AS** réfère aux paramètres valeurs de la procédure. Ce sera la responsabilité de la fonction **FIND** d'affecter ces variables. Ce seront encore quelques variables globales.

Ecrivons donc la fonction **FIND**. **FIND** a besoin de deux arguments, le nom de la fonction et la liste **REC** contenant la description de toutes les procédures déjà définies – dans l'ordre inverse de leur apparition : la dernière procédure définie est en tête de cette liste. Nous devons chercher à l'intérieur de cette liste la description de la procédure appelée. Une fois que nous l'avons trouvée, nous devons prendre les paramètres valeurs et leur substituer les valeurs de l'appel dans l'assertion d'entrée et dans l'assertion de sortie associées à cette fonction. Nous mettrons les résultats de ces deux opérations dans les variables globales **U** et **W** respectivement. Ceci est suffisant pour des appels récursifs d'une fonction.

Si, par contre, nous appelons une autre fonction définie préalablement (c'est donc une fonction dont la description ne se trouve pas en tête de la liste **REC**) nous devons, à l'intérieur des assertions d'entrée et de sortie contenues dans les variables **U** et **W**, substituer les paramètres noms de l'appel pour les paramètres noms de la définition. Ceci est nécessaire pour obtenir un théorème qui exprime des choses en termes des paramètres actuels.

Voici la traduction de cette spécification sous une forme plus LISP :

```
(DEFUN FIND (F L)
  (CATCH 'OUT
    (MAPC
      '(LAMBDA (X)(IF (EQ F (CADR X))
        (THROW 'OUT (PROGN
              (SETQ AS (CADDR X)
                 U (SUBS VS (CAR X) AS)
                 W (SUBS VS (CADDDR X) AS))
              (IF (EQ F (CADAR REC)) NIL
                 (SETQ
                   U (SUBS (CADDDDAR REC) U (CADDDDR X))
                   W (SUBS (CADDDDAR REC) W (CADDDDR X))
                   AS (CADDAR REC)))
              T))))
      L)))
```

Regardez l'utilisation des fonctions **CATCH** et **THROW** : nous plaçons le **CATCH** à l'extérieur du **MAPC**, puisque nous n'utilisons la fonction **MAPC** qu'en tant que procédure de recherche d'un élément. Dès que nous avons trouvé l'élément cherché, nous activons la fonction donnée comme premier argument de **MAPC** et sortons de la fonction **FIND** (avec la fonction **THROW**) *sans* terminer la fonction **MAPC**.

Si les deux fonctions **CADDDDAR** et **CADDDDR** n'existent pas, elles sont aisément définies comme des macro-fonctions :

```
(DEFMACRO CADDDDR (X) '(CADDR (CDDR ,X)))

(DEFMACRO CADDDDAR (X) '(CADDDR (CDAR ,X)))
```

Reste alors la fonction **SUBS** qui doit organiser la substitution à l'intérieur des assertions. La voici :

```
(DEFUN SUBS (APPEL CONDITION VARS)
  (IF (NULL VARS) CONDITION
      (SUBST (CAR APPEL)(CAR VARS)
             (SUBS (CDR APPEL) CONDITION (CDR VARS)))))
```

Et, juste pour la beauté de la chose nous allons légèrement modifier notre fonction **SUBST** afin qu'elle ne fasse pas de substitution pour la variable quantifiée universellement. Ce qui donne :

```
(DEFUN SUBST (VAL VAR BOD)(COND
    ((EQUAL VAR BOD) VAL)
    ((ATOM BOD) BOD)
    ((EQUAL (CAR BOD) '|#|)
        (CONS (CAR BOD) (CONS (CADR BOD)
                (SUBST VAL VAR (CDDR BOD)))))
    (T (CONS (SUBST VAL VAR (CAR BOD))(SUBST VAL VAR (CDR BOD))))))
```

A cet endroit, nous avons presque terminé le noyau de notre programme. Manquent encore les déclarations des variables globales et quelques fonctions qui améliorent l'interaction entre ce programme et son utilisateur.

Donnons la définition des variables globales :

```
(DEFVAR A () "le reste du programme, i.e. tout ce qui précède le filtre")
(DEFVAR F () "la fonction filtrée")
(DEFVAR X () "la variable filtrée, dans un SETQ")
(DEFVAR R1 () "le test d'un IF")
(DEFVAR AS () "les paramètres valeurs de la procédure")
(DEFVAR ES () "les paramètres noms")
(DEFVAR VS () "les arguments d'un appel de fonction")
(DEFVAR K () "le corps d'une procédure")
(DEFVAR E () "une expression filtrée")
(DEFVAR B () "le corps du WHILE et l'action-vraie du IF")
(DEFVAR R () "l'assertion d'invariance de boucle")
(DEFVAR S () "le test d'une itération")
(DEFVAR C () "les actions si-faux d'un IF")
(DEFVAR U () "l'assertion d'entrée d'une procédure")
(DEFVAR W () "l'assertion de sortie d'une procédure")
```

Il nous reste à écrire la fonction **P-P-P** qui doit imprimer les traces et que nous appelons dans la fonction **GCV1** et une routine organisant l'interaction avec l'utilisateur. Ecrivons d'abord la fonction **VC** organisant l'interface utilisateur :

```
(DEFUN VC (&OPTIONAL REC PROG PRECONDITION POSTCONDITION -TRACE-)
    (DECLARE (SPECIAL REC -TRACE-))
    (TERPRI)
    (TERPRI)
    (FORMAT T "donnez le programme s.v.p.")
    (SETQ PROG (READ))
    (IF (NULL PROG) "ok"
      (PROGN (FORMAT T "donnez l'assertion d'entree s.v.p. ")
             (SETQ PRECONDITION (READ))
             (FORMAT T "donnez l'assertion de sortie s.v.p. ")
             (SETQ POSTCONDITION (READ))
             (FORMAT T "voulez-vous une trace? (si oui, tapez 'y') ")
             (SETQ -TRACE-
                   (IF (MEMBER (READ) '(#\y #\Y Y)) T NIL))
             (PR (GCV PROG PRECONDITION POSTCONDITION))
             (FORMAT T "<RETURN> pour continuer")
             (READ-CHAR)
             (VC REC))))
```

Pour interagir avec le programme que nous venons d'écrire, l'utilisateur lance cette fonction **VC**, qui lui demande interactivement les diverses données. Elle lui demande également s'il veut une trace ou non. Ainsi, les impressions de **P-P-P** auront lieu seulement si l'utilisateur veut vraiment voir le déroulement du programme. Ce programme boucle jusqu'à ce qu'on lui donne le programme vide à vérifier. La fonction **PR** doit imprimer les théorèmes générés. Elle est définie ci-dessous :

```
(DEFUN PR (L)
  (LABELS ((PR-AUX (L C)(COND
            ((NULL L)(FORMAT T "ca y est~%"))
            (T (TERPRI)
               (FORMAT T "theoreme" ~S : ~%    ~S~%" (CAR L))
               (PR-AUX (CDR L) (1+ C))))))
    (PR-AUX L 1)))
```

Finalement la fonction **P-P-P** qui imprime les traces :

```
(DEFUN P-P-P (PROG REGG)
    (WHEN -TRACE-
          (TERPRI)
          (FORMAT T "~%au programme")
          (FORMAT T "~% ~S" PROG)
          (TERPRI)
          (FORMAT T "~%avec la precondition ~S" PRECONDITION)
          (FORMAT T "~%et la postcondition ~S" POSTCONDITION)
          (FORMAT T "~%s'applique la règle ~S" (CAAR REGG))
          (FORMAT T "~% ~S" (CADAR REGG))))
```

Le programme de génération de condition de vérification est fini. Avant de terminer ce chapitre, et ce livre, donnons une trace de l'exécution de ce programme :

```
? (VC)
donnez le programme s.v.p.
            ((setq x a)(setq a b)(setq b x))
donnez l'assertion d'entrée s.v.p. (a = n et b = m)
donnez l'assertion de sortie s.v.p. (a = m et b = n)
voulez-vous une trace? (si oui, tapez 'y') y


au programme ((SETQ X A) (SETQ A B) (SETQ B X))

avec la précondition (A = N ET B = M)
et la postcondition(A = M ET B = N)
s'applique la règle "règle de l'affectation:"
"          P {A} Q(e)
          ----------------
         P {A; x <- e} Q(x)"

au programme ((SETQ X A) (SETQ A B))

avec la précondition (A = N ET B = M)
et la postcondition(A = M ET X = N)
s'applique la règle "règle de l'affectation:"
"          P {A} Q(e)
          ----------------
         P {A; x <- e} Q(x)"

au programme ((SETQ X A))

avec la précondition (A = N ET B = M)
et la postcondition(B = M ET X = N)
s'applique la règle "règle de l'affectation:"
"          P {A} Q(e)
          ----------------
         P {A; x <- e} Q(x)"

au programme NIL

avec la précondition (A = N ET B = M)
et la postcondition(B = M ET A = N)
s'applique la règle "règle de consequence (1):"
"          P => Q
          ---------
         P {NULL} Q"
théorème 1 :
     ((A = N ET B = M) "=>" (B = M ET A = N))
ça y est
<RETURN> pour continuer
```

; et voici un exemple plus complexe :

```
donnez le programme s.v.p.
(procédure factorielle (n) (r)
 (if (= n 0)(setq r 1)
    (factorielle (- n 1) r)
    (setq r (* r n))))
donnez l'assertion d'entrée s.v.p. (entier (n) et (n >= 0))
donnez l'assertion de sortie s.v.p. (r = ! n)
voulez-vous une trace? (si oui, tapez 'y') y


au programme (PROCEDURE FACTORIELLE (N) (R)
              (IF (= N 0) (SETQ R 1)
                          (FACTORIELLE (- N 1) R)
                          (SETQ R (* R N))))

avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition(R = ! N)
s'applique la règle "règle de définition de procédure:"
"     P {q (x) (v)} Q ⊢ P {A} Q
      ------------------------
      P {PROCEDURE q (x) (v) A} Q"

au programme ((IF (= N 0) (SETQ R 1)
                          (FACTORIELLE (- N 1) R)
                          (SETQ R (* R N))))

avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition(R = ! N)
s'applique la règle "règle du conditionnel:"
"          P {A Q-if B} R, P {A (non Q-if) C} R
           ------------------------------------
                P {A (IF Q B C)} R"

au programme ((assert-if:|
               (NON (= N 0))) (FACTORIELLE (- N 1) R) (SETQ R (* R N)))

avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition(R = ! N)
s'applique la règle "règle de l'affectation:"
"          P {A} Q(e)
           ----------
          P {A; x <- e} Q(x)"

au programme ((assert-if:|(NON (= N 0))) (FACTORIELLE (- N 1) R))

avec la précondition (ENTIER (N) ET (N >= 0))
```

```
et la postcondition((* R N) =! N)
s'applique la règle "règle de l'appel de procédure:"
"      U (x; v) {q (x) (v)} W (x; v)
          P {A} U (a; e) et # a (W (a; e) ==> Q)
       ⊢  ------------------------------------
                  P {A (q (a) (e))} Q"

au programme ((|assert-if:|(NON (= N 0))))

avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition((ENTIER ((- N 1)) ET ((- N 1) >= 0))
                        ET |#|(N) ((R =! (- N 1)) ==> ((* R N) =! N)))
s'applique la règle "deuxième règle de conséquence :"
"       P {A} Q ==> R
        ------------
        P {A; Q-if} R"

au programme NIL

avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition((NON (= N 0)) "==>"
              ((ENTIER ((- N 1)) ET ((- N 1) >= 0))
                    ET |#|(N) ((R =! (- N 1)) ==> ((* R N) =! N))))
s'applique la règle "règle de conséquence (1):"
"          P ==> Q
           ----------
           P {NULL} Q"

au programme ((|assert-if:|(= N 0)) (SETQ R 1))

avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition(R =! N)
s'applique la règle "règle de l'affectation:"
"          P {A} Q(e)
           -----------------
           P {A; x <- e} Q(x)"

au programme ((|assert-if:|(= N 0)))

avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition(1 =! N)
s'applique la règle "deuxième règle de conséquence :"
"       P {A} Q ==> R
        ------------
        P {A; Q-if} R"

au programme NIL
```

```
avec la précondition (ENTIER (N) ET (N >= 0))
et la postcondition((= N 0) "=>" (1 =! N))
s' applique la règle "règle de conséquence (1):"
"          P => Q
           ---------
          P {NULL} Q"
théorème 1 :
    ((ENTIER (N) ET (N >= 0)) "=>" ((= N 0) "=>" (1 =! N)))

théorème 2 :
    ((ENTIER (N) ET (N >= 0)) "=>"
        ((NON (= N 0)) "=>"
            ((ENTIER ((- N 1)) ET ((- N 1) >= 0))
                ET |#|(N) ((R =! (- N 1)) => ((* R N) =! N)))))
ça y est
<RETURN> pour continuer

donnez le programme s.v.p. ()
"ok"
```

Etant donné que **MATCH** est une fonction très gourmande en calcul, procédons à une optimisation du programme, afin d'accélérer la recherche de la bonne règle. Regardez, tous les filtres des différentes règles imposent une structure particulière *à la fin* de la liste formant le programme. C'est pourquoi, la plupart des filtres sont de la forme

(↑**A** *le filtre*)

Ils sont composés d'une variable de séquence, qui s'appelle *toujours* **A**, suivie du filtre correspondant à la règle tout à la fin de liste. Rappelons-nous, que les variables de séquence sont très coûteuses en temps : chaque utilisation d'une telle variable implique, dès que la séquence correspondant a au moins un élément, beaucoup de *backtracking*. Nous pouvons éviter ce backtracking en inversant le filtre et le programme. Ainsi, la variable de séquence se trouve à la fin du filtre, un cas très simple pour la fonction **MATCH**, et tout ce qui reste à faire après le filtrage, c'est de re-inverser la liste associée à la variable **A**. Ceci accélère considérablement la recherche de la règle s'appliquant au bout du programme considéré.

Pour réaliser cela, nous faisons un tout petit changement et un rajout dans la fonction **GCV**. Dans le listing de la nouvelle fonction **GCV** ci-dessous, les modifications sont données en caractères *italiques* :

```
(DEFUN GCV (PROG PRE POST &OPTIONAL VCS)
    (LABELS ((GCV1 (PROG REGL PRECONDITION
                         POSTCONDITION
                         VCS
                         REGG
                         &OPTIONAL AUX)
                   (DECLARE (SPECIAL AUX PRECONDITION POSTCONDITION VCS)
                   (COND
                       ((NULL REGL)(FORMAT T "erreur dans le programme"))
                       ((SETQ AUX (MATCH (REVERSE (CAAR REGL)) (REVERSE PROG)))
                        ; le CAR de la règle est le filtre ;
                        (LETALL (CADR AUX)
                                (SETQ A (REVERSE A))
                                (P-P-P PROG REGG)
                                ; le CADR de la règle est l'action
                                (EVAL (CADAR REGL))))
                       (T (GCV1 PROG (CDR REGL)
                                     PRECONDITION
                                     POSTCONDITION
                                     VCS
                                     (CDR REGG)) )))
     (GCV1 PROG REGLES PRE POST VCS REGL1)))
```

Bien sûr, nous pourrions optimiser encore plus en donnant les filtres à l'envers, c'est-à-dire : avec la variable de segment à la fin; ainsi nous n'aurions pas à inverser les filtres à chaque appel de **MATCH**. De même, nous pourrions inverser le programme dans l'appel initial de la fonction **GCV1**, ce qui nous économiserait les inversements de la liste représentant le programme dans les appels de **MATCH**.

A vous de procéder à ces optimisations supplémentaires !

Tous les programmes un tant soit peu intéressants ne sont – en réalité – jamais terminés. Ce programme n'est pas une exception. Vous pouvez continuez à le développer. Diverses directions sont ouvertes : vous pouvez adjoindre un interprète pour ce LISP très particulier que le programme sait vérifier. Vous pouvez adjoindre des règles supplémentaires pour d'autres fonctions de base. Vous pouvez écrire un démonstrateur de théorèmes, qui prend les théorèmes que ce programme génère et essaie de les démontrer (ou de les falsifier). etc.

Notre livre d'introduction à la programmation en LISP s'achève ici. Nous n'avons vu qu'une minime partie de ce qui est possible et faisable dans la programmation en général, et dans le langage LISP en particulier. Seules les bases absolument nécessaires ont été exposées. Maintenant il s'agit de pratiquer ce langage : écrire ses propres programmes, lire et modifier des programmes écrits par d'autres.

Ce livre a remplit son rôle s'il vous a donné envie de programmer : à l'instar de toute technique, ce n'est pas la lecture d'un livre, aussi bon soit-il, qui permette l'apprentissage de la programmation : seule votre propre activité de programmation, l'observation de vos erreurs ainsi que leurs corrections et votre interaction avec l'ordinateur vous permettront de devenir un programmeur expert.

21. SOLUTIONS AUX EXERCICES

21.1 CHAPITRE 1

1. Détermination du type :

123	atome numérique
(EIN (SCHOENES) BUCH)	liste
(AHA (ETONNANT . . . . !))	objet impossible à cause des 'points'
(((((1) 2) 3) 4) 5)	liste
-3Aiii	atome alphanumérique
T	atome alphanumérique
(((((ARRRGH)))))	liste

2. Détermination du nombre d'éléments :

(EIN (SCHOENES) BUCH)	3 éléments : **EIN**, **(SCHOENES)** et **BUCH**
(((((1) 2) 3) 4) 5)	2 éléments : **((((1) 2) 3) 4)** et **5**
(((((ARRRGH)))))	1 élément : **((((ARRRGH))))**

(UNIX (IS A) TRADEMARK (OF) BELL LABS)
6 éléments : **UNIX**, **(IS A)**, **TRADEMARK**, **(OF)**, **BELL**, **LABS**

3. Détermination du nombre de la profondeur maximum :

$({}_1$**EIN** $({}_2$**SCHOENES**$)_2$ **BUCH**$)_1$	**SCHOENES** à la profondeur 2
$({}_1({}_2({}_3({}_4({}_5$**1**$)_5$ **2**$)_4$ **3**$)_3$ **4**$)_2$ **5**$)_1$	**1** à la profondeur 5
$({}_1({}_2({}_3({}_4({}_5$**ARRRGH**$)_5)_4)_3)_2)_1$	**ARRRGH** à la profondeur 5
$({}_1$**UNIX** $({}_2$**IS A**$)_2$ **TRADEMARK** $({}_2$**OF**$)_2$ **BELL LABS**$)_1$	**IS**, **A** et **OF** à la profondeur 2

21.2 CHAPITRE 2

1. Les combinaisons de **CAR**, **CDR** et **CONS** :

   a. **(A (B C))**
   b. **(D (E F))**
   c. **(B C)**
   d. **(E F)**
   e. **(NOBODY IS PERFECT)**

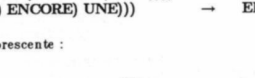
f. ((CAR A) (CONS A B))

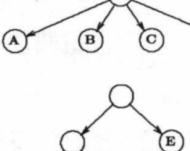
21.3 CHAPITRE 3

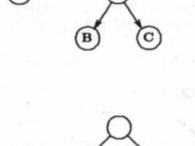
1. Combinaisons de CAR et CDR :

(CAR (CDR (CDR (CDR '(A B C D)))))	→	D
(CAR (CDR (CADR (CAR '((A (B C)) E)))))	→	C
(CAR (CAR (CAR '(((DIEU) ENCORE) UNE))))	→	DIEU
(CADR (CAR '(((DIEU) ENCORE) UNE)))	→	ENCORE

2. Traduction en forme arborescente :





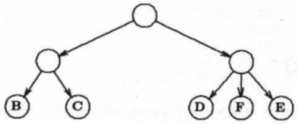


3. SOL, ((C D)), ((HELLO) HOW ARE YOU), (JE JE JE BALBUTIE), (C EST SIMPLE)

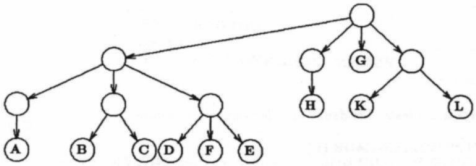
4. Calcul d'opérations sur des arbres :

(A)

((B C) (D F E))



et finalement **(((A) (B C) (D F E)) (H) G (K L))**



5. Traduction des appels de fonction de l'exercice précédent sous forme de liste :

```
(CAR '((A) (B C) (D F E)))
(CDR '((A) (B C) (D F E)))
(CONS '((A) (B C) (D F E)) '((H) G (K L)))
```

21.4 CHAPITRE 4

1. Résultats des appels de la fonction **TRUC1** :

```
((A B C) 1 2 3)
((JA ES GEHT) OUI CA VA)
((A CAR) UNE VOITURE)
```

2. Combinaisons de fonctions utilisateurs :

```
(MI MI MI)
((UN BRAVO LA TABLE) (SUR BRAVO BRAVO) CUBE ROUGE)
```

3. Une fonction de salutation :

```
(DEFUN BONJOUR ()
   '(BONJOUR))
```

4. Quatre répétitions du même élément :

```
(DEFUN 4FOIS (ARGUMENT)
   (CONS ARGUMENT
      (CONS ARGUMENT
         (CONS ARGUMENT
            (CONS ARGUMENT ())))))
```

5. Une fonction qui livre la liste des 3 arguments dans l'ordre inverse :

```
(DEFUN REVERSE3 (ARG1 ARG2 ARG3)
   (CONS ARG3 (CONS ARG2 (CONS ARG1 NIL))))
```

21.5 CHAPITRE 5

21.5.1 chapitre 5.1.1

1. Une fonction qui teste si le deuxième élément est un nombre :

```
(DEFUN NUMBERP-CADR (L)
   (NUMBERP (CADR L)))
```

2. Une fonction qui teste si le premier élément est une liste :

```
(DEFUN LISTE-CAR? (L)
   (CONS 'EST (CONS (CONSP (CAR L)) ())))
```

ou :

```
(DEFUN LISTE-CAR? (L)
   (CONS 'EST (CONS (NULL (ATOM (CAR L))) NIL)))
```

3. Telle que la fonction **NUMBERP-CADR** est définie, elle teste si le *premier* élément est un nombre. Les résultats des appels sont donc : T, NIL, T, T

21.5.2 **chapitre 5.2.1**

1. Une fonction qui teste si les 3 premiers éléments sont des nombres :

```
(DEFUN 3NOMBRES (L)
  (IF (NUMBERP (CAR L))
      (IF (NUMBERP (CADR L))
          (IF (NUMBERP (CADDR L)) 'BRAVO 'PERDANT)
          'PERDANT)
      'PERDANT))
```

2. Une fonction qui inverse une liste de 1, 2 ou 3 éléments :

```
(DEFUN REV (L)
  (IF (NULL (CDR L)) L
      (IF (NULL (CDR (CDR L))) (CONS (CADR L) (CONS (CAR L) ()))
          (CONS (CADDR L) (CONS (CADR L) (CONS (CAR L) NIL))))))
```

3. Voici les résultats des appels de la fonction **BIZARRE** :

```
(ALORS, BEN, QUOI)
(1 2 3)
(1 2 3)
(TIENS C-EST-TOI)
(NOMBRE)
(DO N EST PAS UNE LISTE, MSIEUR)
```

21.6 **CHAPITRE 6**

1. La nouvelle fonction **DELETE**, qui enlève toutes les occurrences du premier argument à des profondeurs quelconques, se différencie de celle examinée par l'appel récursif sur le **CAR** de la liste (dans le cas où le premier élément est une liste) et par l'utilisation de la fonction auxiliaire **COLLE**.

   Cette fonction **COLLE** sert quand une sous-liste ne contient que des éléments à éliminer, comme dans l'appel

```
(DELETE 'A '(1 (A A) 2 3))
```

   Le résultat doit être : **(1 2 3)**, donc une liste où toute cette sous-liste a disparu. Si nous avions utilisé la fonction **CONS** au lieu de cette nouvelle fonction, le résultat aurait été **(1 NIL 2 3)**, ce qui est moins élégant !

```
(DEFUN DELETE (ELE LISTE)
    (IF (NULL LISTE) ()
      (IF (ATOM LISTE)
          (IF (EQ ELE LISTE) () LISTE)
          (IF (CONSP (CAR LISTE))
              (COLLE (DELETE ELE (CAR LISTE)) (DELETE ELE (CDR LISTE)))
              (IF (EQ (CAR LISTE) ELE) (DELETE ELE (CDR LISTE))
                  (CONS (CAR LISTE) (DELETE ELE (CDR LISTE))))))))
```

avec la fonction **COLLE** que voici :

```
(DEFUN COLLE (X LISTE)
    (IF (NULL X) LISTE
        (CONS X LISTE)))
```

2. Une fonction qui double tous les éléments d'une liste :

```
(DEFUN DOUBLE (L)
  (IF (NULL L) ()
      (CONS (CAR L) (CONS (CAR L) (DOUBLE (CDR L))))))
```

3. Et voici une fonction qui double tous les atomes d'une liste :

```
(DEFUN DOUBLE (L)
  (IF (NULL L) ()
      (IF (CONSP (CAR L))
          (CONS (DOUBLE (CAR L)) (DOUBLE (CDR L)))
          (CONS (CAR L) (CONS (CAR L) (DOUBLE (CDR L)))))))
```

4. Voici la fonction **MEMQ** qui cherche l'occurrence d'un élément à l'intérieur d'une liste :

```
(DEFUN MEMQ (ELE LISTE) (COND
    ((NULL LISTE) NIL)
    ((EQ (CAR LISTE) ELE) LISTE)
    (T (MEMQ ELE (CDR LISTE)))))
```

5. La fonction **MEMBER** se distingue de la fonction **MEMQ** par

   - la recherche d'un élément de type quelconque : la modification nécessaire est de remplacer l'appel de la fonction **EQ** par un appel de la fonction **EQUAL**

   - la recherche de l'élément à un niveau arbitraire : comme préalablement avec les fonctions **DELETE** et **DOUBLE**, la solution est d'appliquer la fonction non seulement sur les **CDR**s successifs, mais également sur les **CAR**s, chaque fois que le **CAR** est une liste.

   Voici cette nouvelle fonction :

```
(DEFUN MEMBER (ELE LISTE) (COND
    ((ATOM LISTE) NIL)
    ((EQUAL (CAR LISTE) ELE) LISTE)
    (T (MEMB1 ELE (MEMBER ELE (CAR LISTE)) (CDR LISTE)))))

(DEFUN MEMB1 (ELE AUX LISTE)
    (IF AUX AUX (MEMBER ELE LISTE)))
```

6. Une fonction qui groupe les éléments successifs de deux listes :

```
(DEFUN GR (L1 L2)
  (IF (NULL L1) L2
    (IF (NULL L2) L1
      (CONS
        (CONS (CAR L1) (CONS (CAR L2) ()))
        (GR (CDR L1) (CDR L2))))))
```

7. D'abord la fonction auxiliaire **FOOBAR** : elle construit une liste avec autant d'occurrences du premier argument qu'il y a d'éléments dans la liste qu'est le deuxième argument. Voici quelques exemples d'appels :

```
(FOOBAR '(X Y Z) '(1 2 3))     →   ((X Y Z) (X Y Z) (X Y Z))
(FOOBAR '(1 2 3) '(X Y Z))     →   ((1 2 3) (1 2 3) (1 2 3))
(FOOBAR 1 '(A A A A A A))      →   (1 1 1 1 1 1)
```

Ensuite la fonction **FOO** (à prononcer comme *fouh*) : elle construit une liste contenant les résultats des appels de **FOOBAR** de la liste donnée en argument, et de tous les **CDR**s successifs de cette liste.

Si on livre à **FOO** une liste de *x* éléments, la liste résultat sera donc une liste commençant par *x* répétitions de la liste même, suivie par *x-1* occurrences du **CDR** de cette liste, suivie par *x-2* occurrences du **CDR** du **CDR** de cette liste, et ainsi de suite, jusqu'à une unique occurrence de la liste constituée du dernier élément de la liste donnée en argument.

Par exemple, si l'on appelle **FOO** comme :

```
(FOO '(X Y Z))
```

le résultat sera :

```
((X Y Z) (X Y Z) (X Y Z) (Y Z) (Y Z) (Z))
```

8. Le charme de la fonction **F** se situe dans la permutation des arguments à chaque appel récursif !

Cette fonction construit une liste où les éléments du premier argument et ceux du deuxième argument sont, en préservant l'ordre, mélangés. Voici trois appels de cette fonction :

```
(F '(A B C D) '(1 2 3 4))    →   (A 1 B 2 C 3 D 4)
(F '(A B C D) '(1 2))        →   (A 1 B 2 C D)
(F '(A B C) '(1 2 3 4 5))    →   (A 1 B 2 C 3 4 5)
```

9. Cette surprenante fonction **BAR** (quatre appels récursifs !) calcule l'inverse de la liste donnée en argument. Dans son effet, cette fonction est donc identique à la fonction **REVERSE** avec le deuxième argument égal à **NIL**.

   Afin de bien comprendre comment elle fonctionne, faites-la donc tourner à la main – mais ne prenez pas une liste trop longue !

21.7 CHAPITRE 7

1. Pour la fonction **NTH** nous supposons que l'argument numérique donné sera ≥ 1, autrement la fonction n'aura pas beaucoup de sens : quel sera le '-3ième' élément d'une liste ?

```
(DEFUN NTH (NOMBRE LISTE) (COND
   ((NULL LISTE) NIL)
   ((<= N 1) (CAR LISTE))
   (T (NTH (1- NOMBRE) (CDR LISTE)))))
```

2. L'écriture d'une fonction qui transforme des nombres décimaux en nombres octaux et hexadécimaux sera grandement facilitée si nous divisons cette tâche en plusieurs sous-tâches : d'abord nous allons écrire une fonction **DEC-OCT** qui traduit des nombres décimaux en nombres octaux, ensuite nous alllons écrire une fonction **DEC-HEX** qui traduit les nombres décimaux en nombres hexadécimaux, et, finalement, nous allons écrire la fonction principale **DEC-OCT-HEX** qui ne fait rien d'autre que construire une liste avec les résultats des appels des deux fonctions précédentes.

```
(DEFUN DEC-OCT (N)
  (IF (> N 0)
     (APPEND (DEC-OCT (FLOOR N 8)) (APPEND (REM N 8) NIL)) NIL))

(DEFUN DEC-HEX (N)
  (IF (> N 0)
     (APPEND
        (TRAD (DEC-HEXA (FLOOR N 16)))
        (APPEND (REM N 16) NIL))
     NIL))
```

   La fonction **TRAD** sert à traduire les valeurs entre 10 et 15 dans leurs représentations correspondantes hexadécimales (A à F).

```
(DEFUN TRAD (L)
  (IF (NULL L) ()
     (CONS
        (NTH (1+ (CAR L)) '(0 1 2 3 4 5 6 7 8 9 A B C D E F)))
        (TRAD (CDR L)))))
```

```
(DEFUN DEC-OCT-HEX (NOMBRE)
  (CONS (DEC-HEX NOMBRE) (CONS (DEC-OCT NOMBRE) ())))
```

3. La fonction **OCT-DEC** fait la conversion des nombres octaux en nombres décimaux, la fonction **HEX-DEC** celle des nombres hexadécimaux en nombres décimaux.

```
(DEFUN OCT-DEC (N) (OCT-DEC1 N 0))

(DEFUN OCT-DEC1 (N RES)
    (IF N (OCT-DEC1 (CDR N) (+ (* RES 8) (CAR N))) RES))

(DEFUN HEX-DEC (N) (HEX-DEC1 N 0))

(DEFUN HEX-DEC1 (N RES)
    (IF N (HEX-DEC1 (CDR N) (+ (* RES 16)
                    (IF (NUMBERP (CAR N)) (CAR N)
                     (LET ((M (CAR N))) (COND
                      ((EQ M 'A) 10)
                      ((EQ M 'B) 11)
                      ((EQ M 'C) 12)
                      ((EQ M 'D) 13)
                      ((EQ M 'E) 14)
                      ((EQ M 'F) 15))))))
        RES))
```

4. Et voici la fonction **FIBONACCI** :

```
(DEFUN FIBONACCI (N)
  (IF (<= N 1) 1
      (+ (FIBONACCI (1- N)) (FIBONACCI (- N 2)))))
```

Ci-dessous une version récursive terminale de cette même fonction :

```
(DEFUN FIBONACCI (N)
   (IF (<= N 1) 1 (FIB 2 '(1 1) N)))

(DEFUN FIB (COMPTEUR LAST N)
   (IF (= COMPTEUR N) (+ (CAR LAST)(CADR LAST))
       (FIB (1+ COMPTEUR)
            (CONS (CADR LAST)(CONS (+ (CAR LAST) (CADR LAST)) NIL))
            N)))
```

ou, de manière équivalente, mais sans la fonction **CONS** :

```
(DEFUN FIBONACCI (N)
   (IF (<= N 1) 1 (FIB 2 1 1 N)))

(DEFUN FIB (COMPTEUR AVANT-DERNIER DERNIER N)
   (IF (= COMPTEUR N) (+ AVANT-DERNIER DERNIER)
       (FIB (1+ COMPTEUR) DERNIER (+ AVANT-DERNIER DERNIER) N)))
```

5. La fonction **QUOI** inverse les chiffres d'un nombre, c'est-à-dire, si l'on appelle **QUOI** avec un nombre de la forme $c_1c_2...c_n$, le résultat sera un nombre de la forme $c_nc_{n-1}...c_1$. Par exemple, le résultat de l'appel **(QUOI 1234)** sera le nombre **4321**.

6. Voici la fonction de Monsieur Ackermann :

```
(DEFUN ACK (M N) (COND
   ((ZEROP M) (1+ N))
   ((ZEROP N) (ACK (1- M) 1))
   (T (ACK (1- M) (ACK M (1- N))))))
```

7. Voici deux versions d'une fonction qui calcule la longueur d'une liste. Seule la première version est récursive terminale.

```
(DEFUN LENGTH (L)
   (LABELS ((LENGTH-INTERNE (L N)
            (IF (NULL L) N
                (LENGTH-INTERNE (CDR L) (1+ N)))))
     (LENGTH-INTERNE L 0)))

(DEFUN LENGTH (L)
   (IF (NULL L) 0 (1+ (LENGTH (CDR L)))))
```

La fonction **NBATOM** est identique à la fonction **LENGTH**, sauf qu'elle entre dans toutes les sous-listes. Là aussi, nous allons donner deux versions :

```
(DEFUN NBATOM (L)
   (LABELS ((NBATOM-INTERNE (L N)
            (IF (NULL L) N
                (IF (ATOM (CAR L)) (NBATOM-INTERNE (CDR L) (1+ N))
                (NBATOM-INTERNE (CDR L) (NBATOM-INTERNE (CAR L) N))))))
     (NBATOM-INTERNE L 0)))

(DEFUN NBATOM (L)
   (IF (NULL L) 0
     (IF (ATOM L) 1
         (+ (NBATOM (CAR L)) (NBATOM (CDR L))))))
```

21.8 CHAPITRE 8

1. Bien évidemment, afin de représenter ces diverses relations, il suffit de prendre les relations comme indicateurs sur les P-listes :

   a. **(PUTPROP 'GUY 'PERE 'PIERRE)**
      ou **(PUTPROP 'PIERRE 'ENFANT '(GUY))**
   b. **(PUTPROP 'MARIE 'PERE 'PIERRE)**
      ou
      (PUTPROP 'PIERRE 'ENFANT

```
                (CONS 'MARIE (GET 'PIERRE ENFANT)))
```

c. **(PUTPROP 'JACQUES 'PERE 'PIERRE)**
   ou
   (PUTPROP 'PIERRE 'ENFANT
   (CONS 'JACQUES (GET 'PIERRE ENFANT)))

d. **(PUTPROP 'PIERRE 'SEXE 'MASCULIN)**
e. **(PUTPROP 'MARIE 'SEXE 'FEMININ)**
f. **(PUTPROP 'GUY 'SEXE 'MASCULIN)**
   (PUTPROP 'JACQUES 'SEXE 'MASCULIN)
g. **(PUTPROP 'JUDY 'SEXE (GET 'MARIE 'SEXE))**
h. **(PUTPROP 'JUDY 'AGE 22)**
i. **(PUTPROP 'ANNE 'AGE 40)**
j. **(PUTPROP 'SARAH 'AGE (+ (GET 'JUDY 'AGE) (GET 'ANNE 'AGE)))**

2. Ceci est notre premier exercice d'écriture d'un *vrai* programme : le premier exercice qui ne se réduit pas à l'écriture d'une seule fonction, mais qui suppose une analyse préalable du problème, la décomposition du problème en sous-problèmes, la reconnaissance de sous-problèmes communs à différentes tâches, etc.

   Nous ne donnerons pas la solution complète de cet exercice : seules des *pistes* qu'il faudrait poursuivre et développer sont énoncées :

   Chaque livre sera représenté par un atome dont le nom correspond au code de ce livre. Ainsi, par exemple, **A1**, **A2**, **A3** etc., seront des livres. La bibliothèque entière sera donnée par une fonction, nommons-la **BIBLIO**, qui sera définie comme :

   (DEFUN BIBLIO () '(A1 A2 A3 . . . An))

   Ainsi un appel de :

   (BIBLIO)

   nous livrera la liste :

   (A1 A2 A3 . . . An)

   Ensuite, nous devons à chaque livre attacher des propriétés : son titre, son auteur et la liste des mots clefs. Ces propriétés ont leur place naturelle sur la P-liste des atomes-livres. Par exemple. pour les deux livres de Karl Kraus, nous aurons :

```
(PUTPROP 'A1 'AUTEUR '(KARL KRAUS))
(PUTPROP 'A1 'TITRE '(DIE LETZTEN TAGE DER MENSCHHEIT))
(PUTPROP 'A2 'AUTEUR '(KARL KRAUS))
(PUTPROP 'A2 'TITRE '(VON PEST UND PRESSE))
```

   Ce qui reste à faire ensuite, après avoir mis à jour la bibliothèque, c'est d'écrire un ensemble de fonctions permettant de la consulter.

   Construisons d'abord une fonction auxiliaire qui cherche à l'intérieur de la bibliothèque toutes les occurrences d'une certaine propriété :

```
(DEFUN CHERCHEALL (PROP VAL) (CHERCHE PROP (BIBLIO) VAL))

(DEFUN CHERCHE (PROP BASE VAL) (COND
   ((NULL BASE) NIL)
   ((EQUAL (GET (CAR BASE) PROP) VAL)
          (CONS (CAR BASE) (CHERCHE PROP (CDR BASE) VAL)))
   ((MEMBER VAL (GET (CAR BASE) PROP))
          (CONS (CAR BASE) (CHERCHE PROP (CDR BASE) VAL)))
   (T (CHERCHE PROP (CDR BASE) VAL))))
```

Cette fonction permet, par exemple, de trouver tous les livres d'un certain auteur. Ainsi, si les livres **A1**, **A2** et **A23** sont des livres de Karl Kraus, et le livre **A456**, un livre de Karl Kraus et Ludwig Wittgenstein, l'appel :

```
(CHERCHEALL 'AUTEUR '(KARL KRAUS))
```

nous donne la liste **(A1 A2 A23 A456)**.

Nous avons besoin également d'une fonction, nommons-la **FORALL**, qui nous donne pour chacun de ces livres la propriété cherchée. La voici :

```
(DEFUN FORALL (LISTE PROPRIETE)
    (IF (NULL LISTE) NIL
       (LET ((AUX (GET (CAR LISTE) PROPRIETE)))
         (IF AUX (CONS AUX (FORALL (CDR LISTE) PROPRIETE))
             (FORALL (CDR LISTE) PROPRIETE)))))
```

Maintenant, nous pouvons commencer à écrire les fonctions d'interrogation de cette base de données. D'abord une fonction permettant de trouver, à partir du titre d'un livre, son auteur :

```
(DEFUN AUTEUR (TITRE)
       (FORALL (CHERCHALL 'TITRE TITRE) 'AUTEUR))
```

Remarquez que cette fonction prévoit même le cas où deux auteurs ont écrit un livre de même titre.

Ensuite une fonction qui donne tous les livres d'un auteur :

```
(DEFUN ALLTITRE (AUTEUR)
       (FORALL (CHERCHALL 'AUTEUR AUTEUR) 'TITRE))
```

Pour terminer, voici une fonction qui nous ramène tous les livres se référant à un sujet donné dans la liste de mots-clefs, qui se trouve sous la propriété **MOT-CLEF**.

```
(DEFUN ALLLIVRE (MOT-CLEF)
       (FORALL (CHERCHALL 'MOT-CLEF MOT-CLEF) 'TITRE))
```

Si l'on veut connaître tous les auteurs qui ont écrit sur un sujet donné, il suffit de remplacer, dans cette dernière fonction, la propriété **TITRE** par la propriété **AUTEUR**.

Après ces quelques idées, arrêtons le développement d'un programme d'interrogation d'une base de données bibliographique. A vous de continuer !

3. Donnons d'abord les bonnes propriétés aux divers atomes :

```
(PUTPROP 'ALBERT 'MERE 'MARTINE)
(PUTPROP 'ALBERT 'PERE 'CLAUDE)
(PUTPROP 'PIERRE 'MERE 'MARTINE)
(PUTPROP 'PIERRE 'PERE 'JEAN)
(PUTPROP 'KLAUS 'MERE 'EVE)
(PUTPROP 'KLAUS 'PERE 'ALBERT)
(PUTPROP 'GERARD 'MERE 'JULIE)
(PUTPROP 'GERARD 'PERE 'PIERRE)
(PUTPROP 'GUY 'MERE 'JULIE)
(PUTPROP 'GUY 'PERE 'PIERRE)
(PUTPROP 'MARIE 'MERE 'JULIE)
(PUTPROP 'MARIE 'PERE 'PIERRE)
(PUTPROP 'JACQUES 'MERE 'JULIE)
(PUTPROP 'JACQUES 'PERE 'PIERRE)
(PUTPROP 'ALAIN 'MERE 'ALICE)
(PUTPROP 'ALAIN 'PERE 'KLAUS)
(PUTPROP 'BRIAN 'MERE 'CATHERINE)
(PUTPROP 'BRIAN 'PERE 'ALAIN)
(PUTPROP 'ANNE 'MERE 'SARAH)
(PUTPROP 'ANNE 'PERE 'GERARD)
(PUTPROP 'CATHERINE 'MERE 'JANE)
(PUTPROP 'CATHERINE 'PERE 'JACQUES)
(PUTPROP 'JUDY 'MERE 'ANNE)
(PUTPROP 'JUDY 'PERE 'PAUL)
```

et le sexe :

```
(DEFUN PUT-SEXE (SEXE LISTE)
  (IF (NULL LISTE) NIL
      (PUTPROP (CAR LISTE) 'SEXE SEXE)
      (PUT-SEXE SEXE (CDR LISTE))))

(PUT-SEXE 'MASCULIN
          '(CLAUDE JEAN ALBERT PIERRE KLAUS GERARD GUY JACQUES
            ALAIN BRIAN PAUL))

(PUT-SEXE 'FEMININ
          '(MARTINE EVE JULIE ALICE SARAH MARIE JANE ANNE
            CATHERINE JUDY))
```

Comme dans l'exercice ci-dessus, construisons une fonction qui ramène la liste de toutes les personnes :

```
(DEFUN PERSONNES ()
    '(CLAUDE MARTINE JEAN ALBERT EVE PIERRE JULIE ALICE
      KLAUS SARAH GERARD GUY MARIE JACQUES JANE ALAIN ANNE
      CATHERINE PAUL BRIAN JUDY))
```

Ensuite, il ne reste qu'à écrire des fonctions qui déduisent de cette base de données des relations familiales. Commençons par une fonction qui trouve des frères :

```
(DEFUN FRERES (X)
  (LET ((PERE (GET X 'PERE)) (MERE (GET X 'MERE)))
    (FINDALL 'MERE MERE
      (FINDALL 'PERE PERE
        (FINDALL 'SEXE 'MASCULIN (PERSONNES))))))
```

Naturellement, la fonction **FINDALL** n'existe pas encore ! Comment suis-je tombé là-dessus ? Sachant que des frères doivent tous être de sexe masculin, qu'ils doivent tous avoir le même père et la même mère, je me suis dit qu'afin de trouver tous les frères de quelqu'un, il suffit de chercher à l'intérieur de la liste de toutes les personnes masculines connues, les personnes ayant le même père et qu'à l'intérieur de cet ensemble de personnes il faut trouver celles qui ont la même mère.

Ça fait donc trois fois que je dois chercher à l'intérieur d'une liste tous les éléments qui ont une certaine propriété identique. Au lieu d'écrire trois fois la même chose, j'ai décidé d'écrire une fonction auxiliaire qui devra faire cela. Voici la fonction **FINDALL** :

```
(DEFUN FINDALL (INDICATEUR VALEUR LISTE) (COND
    ((NULL LISTE) ())
    ((EQUAL (GET (CAR LISTE) INDICATEUR) VALEUR)
      (CONS (CAR LISTE)
        (FINDALL INDICATEUR VALEUR (CDR LISTE))))
    (T (FINDALL INDICATEUR VALEUR (CDR LISTE)))))
```

Regardez cette fonction : elle contient deux appels récursifs. Dans ces deux appels seulement *un* argument, **LISTE**, est modifié. Les deux autres arguments restent constants. Sachant que chaque appel d'une fonction implique une nouvelle liaison des variables paramètres, nous pouvons améliorer légèrement cette fonction en sortant les arguments invariants de la boucle. Voici donc la deuxième version de la fonction **FINDALL**, d'abord en Le_LISP :

```
(DE FINDALL (INDICATEUR VALEUR LISTE)
    (LETN SELF ((LISTE LISTE)) (COND
        ((NULL LISTE) ())
        ((EQUAL (GET (CAR LISTE) INDICATEUR) VALEUR)
          (CONS (CAR LISTE) (SELF (CDR LISTE))))
        (T (SELF (CDR LISTE))))))
```

et maintenant pour CommonLISP :

```
(DEFUN FINDALL (INDICATEUR VALEUR LISTE)
  (LABELS ((SELF (LISTE) (COND
                ((NULL LISTE) NIL)
                ((EQUAL (GET (CAR LISTE) INDICATEUR) VALEUR)
                      (CONS (CAR LISTE) (SELF (CDR LISTE))))
                (T (SELF (CDR LISTE))))))
    (SELF LISTE)))
```

Continuons : une fonction qui trouve les sœurs d'une personne ne sera pas difficile à écrire : il suffit de remplacer, dans la fonction **FRERES**, la propriété **MASCULIN** par la propriété **FEMININ**, ce qui donne :

```
(DEFUN SOEURS (X)
  (LET ((PERE (GET X 'PERE)) (MERE (GET X 'MERE)))
      (FINDALL 'MERE MERE
          (FINDALL 'PERE PERE
              (FINDALL 'SEXE 'FEMININ (PERSONNES))))))
```

Maintenant, nous avons deux fonctions quasiment identiques : la fonction **FRERES** et la fonction **SOEURS**. Chaque fois que vous rencontrez dans un programme des fonctions presque identiques, vous pouvez être sûr qu'il y a des améliorations à faire en construisant des fonctions auxiliaires qui captent la similitude. Ecrivons donc une fonction auxiliaire que nous nommerons **FINDALL-TROIS**, puisqu'elle doit trouver tous les éléments qui ont trois propriétés identiques :

```
(DEFUN FINDALL-TROIS (X PROP1 PROP2 PROP3 VALEUR3)
      (FINDALL PROP1 (GET X PROP1)
          (FINDALL PROP2 (GET X PROP2)
              (FINDALL PROP3 VALEUR3 (PERSONNES)))))
```

et les fonctions **FRERES** et **SOEURS** se simplifient en :

```
(DEFUN FRERES (X)
    (DELETE X (FINDALL-TROIS X 'PERE 'MERE 'SEXE 'MASCULIN)))

(DEFUN SOEURS (X)
    (DELETE X (FINDALL-TROIS X 'PERE 'MERE 'SEXE 'FEMININ)))
```

et pour avoir à la fois les frères et les sœurs, il suffit d'écrire :

```
(DEFUN SIBLINGS (X) (DELETE X (APPEND (FRERES X) (SOEURS X))))
```

A vous de compléter l'exercice en écrivant les fonctions nécessaires pour trouver les cousins, cousines, tantes, oncles, etc. Toutes ces fonctions devraient maintenant être faciles à écrire !

Pour conclure cet exercice, voici la fonction qui vous ramène tous les ancêtres d'une personne :

```
(DEFUN ANCETRES (X)
   (LET ((MERE (GET X 'MERE)) (PERE (GET X 'PERE)))
      (IF (NULL MERE)
         (IF (NULL PERE) () (CONS PERE (ANCETRES PERE)))
         (IF (NULL PERE)
            (CONS MERE (ANCETRES MERE))
            (CONS MERE (CONS PERE
                  (APPEND (ANCETRES PERE)
                        (ANCETRES MERE))))))))
```

21.9 CHAPITRE 9

1. Voici la fonction **FACTORIELLE** sous forme de mémo-fonction :

```
(DEFUN FACTORIELLE (N) (COND
   ((<= N 0) 1)
   ((GET 'FACTORIELLE N))
   (T (PUTPROP 'FACTORIELLE N (* N (FACTORIELLE (1- N))))
      (GET 'FACTORIELLE N))))
```

2. Sachant qu'un **COND** évalue un test après l'autre, nous pouvons réécrire ce programme comme :

```
(DEFUN FOO (X) (COND
   ((< X 0) (- 0 X))
   ((ZEROP X) 0)
   (T (* X -1))))
```

   Bien évidemment, ce programme peut encore se simplifier en :

```
(DEFUN INVERSE (X) (- 0 X))
```

3. Voici un premier petit simplificateur algébrique :

```
(DEFUN SIMP (X)(COND
    ((ATOM X) X)
    ((EQ (CAR X) '+)
          (SIMPLIFY-PLUS (SIMP (CADR X)) (SIMP (CADDR X))))
    ((EQ (CAR X) '*)
          (SIMPLIFY-TIMES (SIMP (CADR X)) (SIMP (CADDR X))))
    (T X)))

(DEFUN SIMPLIFY-PLUS (X Y) (COND
     ((AND (NUMBERP X) (NUMBERP Y)) (+ X Y))
     ((ZEROP X) Y)
     ((ZEROP Y) X)
     (T (CONS '+ (CONS X (CONS Y ()))))))

(DEFUN SIMPLIFY-TIMES (X Y) (COND
     ((AND (NUMBERP X) (NUMBERP Y)) (* X Y))
     ((OR (ZEROP X) (ZEROP Y)) 0)
     ((= X 1) Y)
     ((= Y 1) X)
     (T (CONS '* (CONS X (CONS Y ()))))))
```

Dans ce programme nous avons utilisé deux nouvelles fonctions : **AND** et **OR**.

La fonction **AND** est le *et* logique. Elle prend un nombre quelconque d'arguments et ramène la valeur du dernier argument, si *tous* les arguments ont des valeurs différentes de **NIL**, et ramène **NIL** dans les autres cas.

La fonction **OR** est le *ou* logique. Elle aussi prend un nombre quelconque d'arguments et ramène la valeur du *premier* argument qui s'évalue à une valeur différente de **NIL**. Si *tous* les arguments s'évaluent à **NIL**, la valeur de la fonction **OR** est **NIL**.

Bien entendu, nous aurions pu écrire le programme sans ces deux fonctions standard nouvelles. Mais l'écriture avec **AND** et **OR** est plus lisible. Jugez vous même : voici l'écriture de **SIMPLIFY-TIMES** sans ces nouvelles fonctions :

```
(DEFUN SIMPLIFY-TIMES (X Y) (COND
     ((ZEROP X) 0)
     ((ZEROP Y) 0)
     ((= X 1) Y)
     ((= Y 1) X)
     ((NUMBERP X)
          (IF (NUMBERP Y) (* X Y)
               (CONS '* (CONS X (CONS Y ())))))
     (T (CONS '* (CONS X (CONS Y ()))))))
```

21.10 CHAPITRE 10

1. Si nous voulons généraliser ce programme de manière qu'il puisse conjuguer des verbes non seulement au présent, mais également à d'autres temps, la première chose à résoudre est : comment représenter les terminaisons supplémentaires ?

   Actuellement, les terminaisons se trouvent en plein milieu de la fonction **CONJ1**. Cacher des données à l'intérieur du code n'est que très rarement une bonne méthode. Souvent il est préférable (et plus lisible) que les données soient séparées des algorithmes. C'est ce que nous allons faire en créant deux atomes **ER** et **IR**. Sur les P-listes de ces deux atomes nous mettrons, sous des indicateurs qui correspondent au temps les listes de terminaisons. Ce qui donne :

```
(PUTPROP   'IR
           'présent
              '(is is it it issons issez issent issent)
           'imparfait
              '(issais issais issait issait issions issiez issaient issaient)
           'passé-simple
              '(is is it it îmes îtes irent irent)
           'futur
              '(irai iras ira ira irons irez iront iront)
           'conditionnel
              '(irais irais irait irait irions iriez iraient iraient)
           'subjonctif-présent
              '(isse isses isse isse issions issiez issent issent)
           'subjonctif-imparfait
              '(isse isse ît ît issions issiez issent issent))

(PUTPROP   'ER
           'présent
              '(e es e e ons ez ent ent)
           'imparfait
              '(ais ais ait ait ions iez aient aient)
           'passé-simple
              '(ai as a a âmes âtes èrent èrent)
           'futur
              '(erai eras era era erons erez eront eront)
           'conditionnel
              '(erais erais erait erait erions eriez eraient eraient)
           'subjonctif-présent
              '(e es e e ions iez ent ent)
           'subjonctif-imparfait
              '(asse asses ât ât assions assiez assent assent))
```

   Maintenant nous devons également changer la fonction **CONJ1** pour enlever les listes de terminaisons qui s'y trouvent et pour adapter le programme aux changements : nous allons donc remplacer les lignes :

```
(LET ((TERMINAISONS (COND
        ((EQ TYP 'ER) '(E ES E E ONS EZ ENT ENT))
        ((EQ TYP 'RE) '(S S T T ONS EZ ENT ENT))
        (T '(IS IS IT IT ISSONS ISSEZ ISSENT ISSENT)))))
```

par l'unique ligne :

```
(LET ((TERMINAISONS (GET TYP TEMPS)))
```

en supposant que la variable **TEMPS** ait comme valeur le temps de conjugaison demandé. **TEMPS** sera donc le nouveau deuxième argument de la fonction **CONJUGUE** :

```
(DEFUN CONJUGUE (VERBE TEMPS) . . .
```

Ensuite nous remplaçons l'appel de la fonction **PRINT**, à l'intérieur de **CONJ1**, par un appel de la fonction **VERBPRINT** que voici :

```
(DEFUN VERBPRINT (PRONOM VERBE TEMPS)
  (LET ((AUX (MEMQ TEMPS '(subjonctif-présent subjonctif-imparfait))))
    (FORMAT T "~%~A~A~A"
      (IF AUX
        (IF (MEMQ PRONOM '(il elle ils elles)) "qu'" "que ") "")
      PRONOM VERBE)))
```

et le tour est joué !

Néanmoins, comme vous avez sûrement constaté, il n'y encore rien de prévu pour des conjugaisons dans des temps composés. Afin de pouvoir conjuguer dans ces temps, il faut absolument connaître les conjugaisons du verbe *avoir*. Mettons-les sur la P-liste de l'atome **AVOIR** :

```
(PUTPROP   'AVOIR
           'passé-composé
             '(ai as a a avons avez ont ont)
           'plus-que-parfait
             '(avais avais avait avait avions aviez avaient avaient)
           'passé-antérieur
             '(eus eus eut eut eûmes eûtes eurent eurent)
           'futur-antérieur
             '(aurai auras aura aura aurons aurez auront auront)
           'conditionnel-passé1
             '(aurais aurais aurait aurait aurions auriez auraient auraient)
           'conditionnel-passé2
             '(eusse eusses eût eût eussions eussiez eussent eussent))
```

Ensuite construisons, comme précédemment pour le subjonctif, une fonction spéciale d'impression des temps composés :

```
(DEFUN PR-COMPOS (PRON PARTICIPE TEMPS RACINE)
  (LABELS ((DO-IT (AUX PRON PARTICIPE) (COND
             ((NULL AUX) "voila, c'est tout")
             (T (FORMAT T "~%~A ~A ~A"
                  (IF (EQ (CAR PRON) 'je) "j'" (CAR PRON))
                  (CAR AUX) PARTICIPE)
                (DO-IT (CDR AUX) (CDR PRON) PARTICIPE)))))
    (DO-IT (GET 'AVOIR TEMPS)
           PRON
           (CREE-MOT RACINE PARTICIPE))))
```

Il ne reste qu'à changer la fonction **CONJ1** de manière à introduire une possibilité de construire des temps composés. Rappelons-nous qu'actuellement nous avons une fonction qui sait conjuguer normalement (des temps non composés) et une autre qui sait conjuguer des temps composés. Donc il faut introduire quelque part un test permettant d'appeler soit l'une soit l'autre.

La fonction **CONJ1** ci-dessous résout le problème en restant à l'intérieur de **CONJ1** si l'évaluation de **(GET TYP TEMPS)** était différente de **NIL**, et lance la fonction **PR-COMPOS** dans le cas contraire, c'est-à-dire quand il n'y a pas, sur la P-liste de **TYP** (donc **ER** ou **IR**), une liste de terminaisons associée au temps demandé :

```
(DEFUN CONJ1 (RACINE TYP PRONOMS TEMPS)
  (LABELS ((DO-IT (PRONOMS TERMINAISONS)(COND
             ((NULL PRONOMS) "voila, c'est tout")
             ((ATOM TERMINAISONS)
                (PR-COMPOS PRONOMS
                           (IF (EQ TYP 'IR) 'I 'E)
                           TEMPS RACINE))
             (T (VERBPRINT (CAR PRONOMS)
                           (CREE-MOT RACINE (CAR TERMINAISONS))
                           TEMPS)
                (DO-IT (CDR PRONOMS) (CDR TERMINAISONS))))))
    (DO-IT PRONOMS (GET TYP TEMPS))))
```

Maintenant, il ne reste qu'à adapter l'appel de **CONJ1**. Voici alors la nouvelle version de la fonction **CONJUGUE** :

```
(DEFUN CONJUGUE (VERBE TEMPS)
  (TERPRI)
  (CONJ1 (RACINE VERBE) (TYPE VERBE)
         '(JE TU IL ELLE NOUS VOUS ILS ELLES) TEMPS))
```

2. Cet exercice est laissé entièrement à vos soins ! Notez toutefois, qu'après l'exercice précédent, il ne devrait poser aucun problème.

3. Entre temps il doit être évident qu'un problème peut être résolu (et donc programmé) de beaucoup de façons différentes. Voici *une* manière de résoudre le problème de la tour de Hanoi avec, initialement, des disques sur chacune des aiguilles :

```
(DEFUN HANOI (N S F I)
  (LABELS ((HANOI-AUX (N S F I AID)
             (WHEN (> N 0)
                   (IF AID
                       (HANOI-AUX (1- N) S I F)
                       (HANOI-AUX N I F S N))
                   (FORMAT T "~%Disque ~A de ~A vers ~A" N S F)
                   (IF (AND AID (= N AID))
                       (HAN (1- N) I S F)
                       (HAN (1- N) I F S))))

           (PRINT-HANOI (C)
             (WHEN (> C 0)
                   (FORMAT T "~%Disque ~A de ~A vers ~A" N S F)
                   (PRINT-HANOI (1- C))))

           (HAN (N S F I)
             (WHEN (> N 0)
                   (HAN (1- N) S I F)
                   (PRINT-HANOI 3)
                   (HAN (1- N) I F S))))
    (HANOI-AUX N S F I NIL)))
```

Etudiez bien cette fonction ! Elle a l'air très difficile à comprendre !

Notez que j'ai utilisé la fonction **WHEN**. Cette fonction est similaire à la fonction **IF** : le premier argument est un *test*, et si ce test s'évalue à quelque chose de différent de **NIL** la suite d'instructions qui compose le reste des arguments du **WHEN** est exécutée et la valeur de la dernière expression est celle du **WHEN**, sinon, quand l'évaluation du test donne **NIL**, on sort du **WHEN** en ramenant la valeur **NIL**.

Voici deux appels de cette fonction (si vous voulez voir le résultat avec plus de disques, implémentez, améliorez et testez vous-mêmes le programme) :

```
? (HANOI 1 'DEBUT 'ARRIVEE 'INTERMEDIAIRE)

disque 1 de INTERMEDIAIRE vers ARRIVEE
disque 1 de DEBUT vers ARRIVEE
= NIL

? (HANOI 2 'DEBUT 'ARRIVEE 'INTERMEDIAIRE)

disque 1 de ARRIVEE vers DEBUT
disque 1 de INTERMEDIAIRE vers DEBUT
disque 2 de INTERMEDIAIRE vers ARRIVEE
disque 1 de DEBUT vers INTERMEDIAIRE
disque 1 de DEBUT vers INTERMEDIAIRE
disque 1 de DEBUT vers INTERMEDIAIRE
disque 2 de DEBUT vers ARRIVEE
disque 1 de INTERMEDIAIRE vers ARRIVEE
```

```
disque 1 de INTERMEDIAIRE vers ARRIVEE
disque 1 de INTERMEDIAIRE vers ARRIVEE
= NIL
```

21.11 CHAPITRE 11

21.11.1 chapitre 11.1.3

1. La fonction **TIMES**, qui multiplie une suite de nombres, peut être définie comme suit :

```
(DE TIMES L (TTIMES L))

(DE TTIMES (L)
   (IF (NULL (CDR L)) (CAR L)
      (* (CAR L) (TTIMES (CDR L)))))
```

2. Voici une fonction qui construit une *liste des valeurs* des éléments de rang pair :

```
(DE SLIST L (SSLIST (CDR L)))

(DE SSLIST (L)
   (IF (NULL L) ()
      (CONS (CAR L) (SSLIST (CDDR L)))))
```

3. Et voici la fonction qui construit une *liste des éléments* de rang pair :

```
(DF QSLIST (L) (SSLIST (CDR L)))

(DE SSLIST (L)
   (IF (NULL L) ()
      (CONS (CAR L) (SSLIST (CDDR L)))))
```

   Les deux fonctions **SSLIST** sont identiques. La différence se situe exclusivement au niveau de la définition de la fonction **SLIST**, une NEXPR, et **QSLIST**, une FEXPR.

4. Voici la fonction **MAX** qui doit trouver le maximum dans une suite de nombres :

```
(DE MAX L (IF L (MAXIMUM (CDR L) (CAR L))))
```

   La fonction auxiliaire **MAXIMUM** prendra deux arguments : une liste de nombres et le maximum déjà trouvé. Naturellement, rien n'exclut que je considère le premier argument comme un maximum temporaire, je cherche donc dans la liste restante un nombre plus grand, et ainsi de suite. La voici :

```
(DE MAXIMUM (L N)
   (IF (NULL L) N
      (MAXIMUM (CDR L) (IF (< N (CAR L)) (CAR L) N))))
```

La fonction **MIN** sera évidemment similaire : l'unique différence sera qu'au lieu de faire un test (< N (CAR L)) nous testerons si (> N (CAR L)). Voici donc les deux fonctions nécessaires :

```
(DE MIN L (IF L (MINIMUM (CDR L) (CAR L))))

(DE MINIMUM (L N)
    (IF (NULL L) N
        (MINIMUM (CDR L) (IF (> N (CAR L)) (CAR L) N))))
```

21.11.2 **chapitre 11.2.1**

1. La fonction **TIMES** qui multiplie une suite de nombres peut être définie comme suit :

```
(DEFUN TIMES (&REST L)
    (LABELS ((TTIMES (L)
                (IF (NULL (CDR L)) (CAR L)
                    (* (CAR L) (TTIMES (CDR L))))))
        (TTIMES L)))
```

2. Voici la fonction **SLIST** qui construit une liste des valeurs des éléments de rang pair :

```
(DEFUN SLIST (&REST L)
    (LABELS ((SSLIST (L)
                (IF (NULL L) NIL
                    (CONS (CAR L) (SSLIST (CDDR L))))))
        (SSLIST L)))
```

3. Finalement les deux fonctions **MAX** et **MIN** :

```
(DEFUN MAX (&REST L)
    (LABELS ((MAXIMUM (L N)
                (IF (NULL L) N
                    (MAXIMUM (CDR L)
                        (IF (< N (CAR L)) (CAR L) N)))))
        (IF L (MAXIMUM (CDR L) (CAR L)))))

(DEFUN MIN (&REST L)
    (LABELS ((MINIMUM (L N)
                (IF (NULL L) N
                    (MINIMUM (CDR L)
                        (IF (> N (CAR L)) (CAR L) N)))))
        (IF L (MINIMUM (CDR L) (CAR L)))))
```

4. Pour réécrire la fonction **RACINE-ENTIER** sans utiliser une fonction auxiliaire, nous devons déclarer les paramètres **P** et **K** de la fonction auxiliaire comme des paramètres optionnels de la fonction principale qui prennent comme valeurs initiales les valeurs de l'appel initial de cette fonction auxiliaire. Ce qui donne :

```
(DEFUN RACINE-ENTIER (N &OPTIONAL (P 1) (K 0))
    (IF (< N 0) K
        (RACINE-ENTIER (- N (+ 2 P)) (+ P 2) (1+ K))))
```

21.12 CHAPITRE 12

1. La nouvelle version de la fonction **TIMES** :

```
(DEFUN TIMES (&REST L)
  (IF (NULL L) 1
      (* (CAR L) (APPLY 'TIMES (CDR L)))))
```

2. La nouvelle version de la fonction **SLIST** :

```
(DEFUN SLIST (&REST L)
  (IF (NULL (CDR L)) ()
      (CONS (CADR L) (APPLY 'SLIST (CDDR L)))))
```

3. La fonction **QSLIST** ne peut toujours être écrite qu'en Le_LISP. Voici alors sa nouvelle version :

```
(DF QSLIST (L)
  (IF (NULL (CDR L)) ()
      (CONS (CADR L) (EVAL (CONS 'QSLIST (CDDR L))))))
```

   Ici nous ne pouvons pas utiliser la fonction **APPLY** : elle n'admet pas de fonction de type FEXPR en position fonctionnelle.

4. Afin que notre petit traducteur mot à mot fonctionne au moins pour les quelques phrases exemples, nous avons besoin du vocabulaire suivant :

```
(DEFUN CHAT () 'CAT)
(DEFUN LE () 'THE)
(DEFUN MANGE () 'EATS)
(DEFUN SOURIS () 'MOUSE)
(DEFUN A () 'HAS)
(DEFUN VOLE () 'STOLEN)
(DEFUN FROMAGE () 'CHEESE)
```

   Ensuite nous avons besoin d'une fonction qui prend une suite quelconque de mots – ça sera donc une FEXPR Le_LISP – et transforme chacun de ces mots en un appel de fonction et ramène en valeur la suite des résultats de chacune des fonctions-mots. Allons-y :

```
(DF TRADUIRE (PHRASE)
  (IF (NULL PHRASE) ()
      (CONS
        (APPLY (CAR PHRASE) ())          ; pour traduire un mot ;
        (EVAL (CONS 'TRADUIRE            ; pour traduire le reste ;
                    (CDR PHRASE))))))    ; de la phrase ;
```

En CommonLISP nous ne pouvons – pour l'instant – pas encore écrire des fonctions qui n'évaluent pas leurs arguments. Pour tout de même faire marcher notre traducteur en CommonLISP, nous allons légèrement modifier l'énoncé du problème : l'appel CommonLISP de **TRADUIRE** ne sera pas de la forme :

(TRADUIRE LE CHAT MANGE)

mais de la forme suivante :

(TRADUIRE '(LE CHAT MANGE))

Ce qui nous donnera le programme suivant :

```
(DEFUN TRADUIRE (PHRASE)
  (IF (NULL PHRASE) ()
      (CONS
        (APPLY (CAR PHRASE) ())
        (TRADUIRE (CDR PHRASE)))))
```

21.13 CHAPITRE 13

1. Voici, d'abord en Le_LISP, le macro-caractère '^' qui vous permet de charger un fichier simplement en écrivant

^nom-de-fichier

(DMC |^| () ['LOAD (READ)])

Remarquez bien qu'une fois que nous avons défini un macro-caractère, nous pouvons l'utiliser dans la définition d'autres macro-caractères. Ici, nous avons, par exemple, utilisé les macro-caractères '[' et ']' définis dans ce chapitre.

et ensuite la même chose en CommonLISP :

```
(SET-MACRO-CHARACTER #\^ '(LAMBDA (FLUX CARACTERE)
                            (LIST 'LOAD (READ FLUX T NIL T))))
```

2. Le problème dans l'écriture de ces deux macro-caractères est qu'ils se composent de plus d'un seul caractère. C'est donc à l'intérieur de la définition des macro-caractères '>' et '<' que nous devons tester qu'ils sont bien suivis du caractère '='. Voici les définitions

des deux macro-caractères (comme ci-dessus, d'abord en Le_LISP, ensuite en CommonLISP) :

```
(DMC |>| ()
  (LET ((X (PEEKCH)))
       (IF (NULL (EQUAL X #\=))
           ['> (READ)(READ)]
           (READCH)
           (LET ((X (READ))(Y (READ)))
                ['OR ['> X Y]['= X Y]]))))

(DMC |<| ()
  (LET ((X (PEEKCH)))
       (IF (NULL (EQUAL X #\=))
           ['< (READ)(READ)]
           (READCH)
           (LET ((X (READ))(Y (READ)))
                ['OR ['< X Y]['= X Y]]))))
```

et maintenant ces deux macro-caractères en CommonLISP :

```
(SET-MACRO-CHARACTER #\> '(LAMBDA (FLUX CARACTERE)
  (LET ((X (PEEK-CHAR FLUX T NIL T)))
    (COND ((EQUAL X #\=) (READ-CHAR)
           (LET ((X (READ FLUX T NIL T)) (Y (READ FLUX T NIL T)))
                (LIST 'OR (LIST '> X Y)(LIST '= X Y))))
          (T (LIST '|>| (READ FLUX T NIL T)(READ FLUX T NIL T)))))))

(SET-MACRO-CHARACTER #\< '(LAMBDA (FLUX CARACTERE)
  (LET ((X (PEEK-CHAR FLUX T NIL T)))
    (COND ((EQUAL X #\=) (READ-CHAR)
           (LET ((X (READ FLUX T NIL T)) (Y (READ FLUX T NIL T)))
                (LIST 'OR (LIST '< X Y)(LIST '= X Y))))
          (T (LIST '|<| (READ FLUX T NIL T)(READ FLUX T NIL T)))))))
```

3. D'abord, en CommonLISP, le programme qui écrit les nombres pairs de 2 à N dans le fichier **pair.nb** :

```
(DEFUN PAIR (N)
  (LABELS ((INTERNAL-PAIR (X FLUX)(COND
           ((<= X N) (PRINT X FLUX)
                     (INTERNAL-PAIR (+ X 2) FLUX))
           (T (PRINT 'END FLUX)(CLOSE FLUX)))))
    (INTERNAL-PAIR 2 (OPEN "pair.no" :DIRECTION :OUTPUT))))
```

Ensuite la même chose pour les nombres impairs :

```
(DEFUN IMPAIR (N)
  (LABELS ((INTERNAL-IMPAIR (X FLUX)(COND
           ((<= X N) (PRINT FLUX X)
                     (INTERNAL-IMPAIR (+ X 2) FLUX))
           (T (PRINT 'END FLUX)(CLOSE FLUX)))))
    (INTERNAL-IMPAIR 1 (OPEN "impair.no" :DIRECTION :OUTPUT))))
```

Enfin, il ne reste qu'à calculer la somme :

```
(DEFUN SOMME (IMPAIR PAIR)
  (LABELS ((INTERNAL-SOMME (IMPAIR PAIR OUT X Y)(COND
           ((EQUAL X 'END) (CLOSE IMPAIR)(CLOSE PAIR)(CLOSE OUT))
           ((EQUAL Y 'END) (CLOSE IMPAIR)(CLOSE PAIR)(CLOSE OUT))
           (T (PRINT (+ X Y) OUT)
             (INTERNAL-SOMME IMPAIR PAIR OUT
                             (READ PAIR)(READ IMPAIR))))))
    (LET ((IMPAIR (OPEN IMPAIR :DIRECTION :INPUT))
          (PAIR (OPEN PAIR :DIRECTION :INPUT))
          (OUT (OPEN "sum.no" :DIRECTION :OUTPUT)))

      (INTERNAL-SOMME IMPAIR PAIR OUT
                      (READ PAIR)(READ IMPAIR)))))
```

Regardons maintenant le même algorithme en Le_LISP. Tout d'abord le programme écrivant les nombres pairs de 2 à N dans le fichier **pair.no** :

```
(DE PAIR (N)
  (LET ((FLUX (OPENO "pair.no")))
       (OUTCHAN FLUX)
       (LETN INTERNAL-PAIR ((X 2))(COND
                ((<= X N) (PRINT X)
                          (INTERNAL-PAIR (+ X 2)))
                (T (PRINT 'END)(OUTCHAN NIL)(CLOSE FLUX))))))
```

Clairement, l'avantage de la version Le_LISP est que la technique qui consiste à connecter un canal (un *chanal*) permet d'économiser toute référence au flux à l'intérieur de la boucle **INTERNAL-PAIR**.

Faisons alors la même chose pour écrire les nombres impairs de 1 à N dans le fichier **impair.no** :

```
(DE IMPAIR (N)
  (LET ((FLUX (OPENO "impair.no")))
       (OUTCHAN FLUX)
       (LETN INTERNAL-PAIR ((X 1))(COND
                ((<= X N) (PRINT X)
                          (INTERNAL-PAIR (+ X 2)))
                (T (PRINT 'END)(OUTCHAN NIL)(CLOSE FLUX))))))
```

Finalement, il ne reste qu'à écrire le programme calculant les sommes des nombres dans les deux fichiers. Ici, CommonLISP est définitivement avantagé : l'introduction du flux

comme argument supplémentaire dans les fonctions d'entrées/sorties permet un changement aisé d'un flux vers un autre. Par contre, en Le_LISP, nous devons pour chaque changement de flux appeler la fonction **INCHAN**, si c'est un changement de flux d'entrée, ou la fonction **OUTCHAN**, si c'est un changement du flux de sortie.

Ici, dans la fonction **SOMME** nous connectons le flux de sortie à l'extérieur de la boucle qui calcule les sommes (puisqu'il n'y a qu'un seul flux de sortie dans le programme) et pour les opérations de lecture nous écrivons chaque fois la séquence :

(INCHAN *flux*)(READ)

Voici donc cette fonction **SOMME** en Le_LISP :

```
(DE SOMME (IMPAIR PAIR)
  (LET ((FLUX-IMPAIR (OPENI IMPAIR))
        (FLUX-PAIR (OPENI PAIR))
        (FLUX-SOMME (OPENO "sum.no")))
        (OUTCHAN FLUX-SOMME)
        (LETN INTERNAL-SOMME
                    ((X (PROGN (INCHAN FLUX-IMPAIR)(READ)))
                     (Y (PROGN (INCHAN FLUX-PAIR)(READ))))
            (COND
                ((EQUAL X 'END) (CLOSE FLUX-IMPAIR)(CLOSE FLUX-PAIR)
                    (CLOSE FLUX-SOMME)(INCHAN NIL)(OUTCHAN NIL))
                ((EQUAL Y 'END) (CLOSE FLUX-IMPAIR)(CLOSE FLUX-PAIR)
                    (CLOSE FLUX-SOMME)(INCHAN NIL)(OUTCHAN NIL))
                (T (PRINT (+ X Y))
                  (INTERNAL-SOMME
                              (PROGN (INCHAN FLUX-IMPAIR)(READ))
                              (PROGN (INCHAN FLUX-PAIR)(READ))))))))
```

21.14 CHAPITRE 14

1. D'après ce que nous savons, tous les appels récursifs qui ne sont pas arguments d'une autre fonction, c.à.d.: dont le résultat n'est pas utilisé par une autre fonction, sont des appels récursifs terminaux. Les autres sont des appels récursifs normaux. Ci-dessous, nous redonnons la fonction **MATCH** avec les appels récursifs terminaux écrits en *italique* :

```
(DEFUN MATCH (FILTRE DONNEE) (COND
      ((ATOM FILTRE)(EQUAL FILTRE DONNEE))
      ((EQUAL (CAR FILTRE) '|?|) (COND
            ((NULL (CDR FILTRE)) T)
            ((MATCH (CDR FILTRE) DONNEE))
            (DONNEE (MATCH FILTRE (CDR DONNEE)))))
      ((ATOM DONNEE) NIL)
      ((EQUAL (CAR FILTRE) '|$|)
         (MATCH (CDR FILTRE) (CDR DONNEE)))
      ((MATCH (CAR FILTRE) (CAR DONNEE))
         (MATCH (CDR FILTRE) (CDR DONNEE)))))
```

2. Afin d'ajouter le signe '%', nous devons introduire une clause supplémentaire dans la fonction **MATCH**. Evidemment, cette clause doit tester si le filtre commence par une liste et si cette liste commence par ce signé '%' particulier :

```
((AND (CONSP (CAR FILTRE)) (EQ (CAAR FILTRE) '|%|)) . . . )
```

Ensuite, comme pour le traitement du filtrage des *séquences*, nous allons récursivement appeler **MATCH** en testant succesivement un élément après l'autre de cette liste de candidats. Précisément : nous allons construire des nouveaux *filtres*, un filtre particulier pour chaque candidat, regarder si ce filtre correspond au premier élément de la donnée, si oui, nous avons trouvé une bonne donnée et il ne reste qu'à filtrer le reste du filtre avec le reste de la donnée, si non, il faut alors tester avec le filtre constitué du candidat suivant. C'est seulement si aucun des filtres ainsi construits ne correspond au premier élément de la donnée que nous devons signaler un échec.

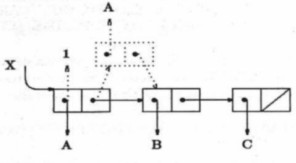
Etant donné que tout cela semble plutôt compliqué, nous allons écrire une petite fonction **CHOIX** locale à la fonction **MATCH**. Voici la nouvelle fonction **MATCH** avec les parties rajoutées écrites en caractères *italiques* :

```
(DEFUN MATCH (FILTRE DONNEE)
  (LABELS ((CHOIX (AUX FILTRE DONNEE) (COND
                ((NULL AUX) NIL)
                ((MATCH (CAR AUX) (CAR DONNEE))
                      (MATCH (CDR FILTRE)(CDR DONNEE)))
                (T (CHOIX (CDR AUX) FILTRE DONNEE)))))
    (COND
      ((ATOM FILTRE)(EQUAL FILTRE DONNEE))
      ((EQUAL (CAR FILTRE) '|?|) (COND
            ((NULL (CDR FILTRE)) T)
            ((MATCH (CDR FILTRE) DONNEE))
            (DONNEE (MATCH FILTRE (CDR DONNEE)))))
      ((ATOM DONNEE) NIL)
      ((EQUAL (CAR FILTRE) '|$|)
         (MATCH (CDR FILTRE) (CDR DONNEE)))
      ((AND (CONSP (CAR FILTRE))(EQUAL (CAAR FILTRE) '|%|))
            (CHOIX (CDR (CAR FILTRE)) FILTRE DONNEE)))
      ((MATCH (CAR FILTRE) (CAR DONNEE))
         (MATCH (CDR FILTRE) (CDR DONNEE))))))
```

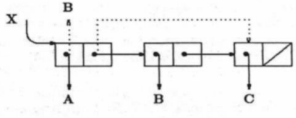
21.15 CHAPITRE 15

1. Dans les représentations graphiques suivantes, les listes originales sont dessinées avec des traits pleins, les modifications avec des traits en pointillés.

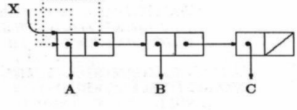
   a. **(SET 'X '(A B C)) (ATTACH 1 X)**



   b. **(LET ((X '(A B C))) (SMASH X))**



   c. **(SETQ X '(A B C)) (RPLACA (RPLACD X X) X)**



2. Voici une fonction qui enlève physiquement toutes les occurrences d'un élément donné à l'intérieur d'une liste. La fonction **FDELQ** ne sert que pour le cas particulier où la liste commence par l'élément cherché. Dans ce cas, on remplace le contenu du premier doublet par celui du doublet suivant. Ainsi nous sommes sûrs que tous les pointeurs vers la liste originale pointent toujours vers la liste modifiée.

```
(DEFUN FDELQ (X Y)(COND
    ((NULL Y) ())
    ((EQUAL X (CAR Y)) (IF (CDR Y) (FDELQ X (SMASH Y)) ()))
    (T (DEL1 X Y))))

(DEFUN DEL1 (X Y)(COND
    ((NULL Y) ())
    ((EQUAL X (CAR Y)) (DEL1 X (CDR Y)))
    ((CONSP (CAR Y))
        (RPLACA Y (DEL1 X (CAR Y))) (RPLACD Y (DEL1 X (CDR Y))))
    (T (RPLACD Y (DEL1 X (CDR Y))))))))
```

3. Pour inverser une liste circulaire simple, nous devons remplacer le test de fin de liste par un test d'égalité de la liste pointée actuellement avec la liste originale. Voici d'abord une version en Le_LISP (juste pour ne pas oublier les différences entre Le_LISP et CommonLISP) et ensuite en CommonLISP :

```
(DE C-REVERSE (L)
    (AND L
        (LETN INTTERNAL-C-REV ((C (CDR L)) (P L))
            (IF (NEQ L C) (INTERNAL-C-REV (CDR C) (RPLACD C P))
                (RPLACD L P) P))))

(DEFUN C-REVERSE (L)
  (LABELS ((INTERNAL-C-REV (C P)
            (IF (NOT (EQ L C))(INTERNAL-C-REV (CDR C)(RPLACD C P))
                (PROGN (RPLACD L P) P))))
    (AND L (INTERNAL-C-REV (CDR L) L))))
```

4. Finalement la fonction **FFREVERSE** qui inverse physiquement une liste (sans circularités) sur tous les niveaux :

```
(DEFUN FREVERSE (L)
  (LABELS ((INTERNAL-FREV (L P)
            (IF L
                (IF (CONSP (CAR L))
                    (INTERNAL-FREV (CDR L)
                                   (RPLACD
                                       (RPLACA L (FREVERSE (CAR L))) P))
                    (INTERNAL-FREV (CDR L) (RPLACD L P))) P)))
    (INTERNAL-FREV L NIL)))
```

5. Voici la définition de la fonction **NEXTL** en Le_LISP :

```
(DF NEXTL (VAR)
    (LET ((VARIABLE (CAR VAR)) (VALEUR (EVAL (CAR VAR))))
        (SET VARIABLE (CDR VALEUR))
        (CAR VALEUR)))
```

6. La fonction **RPLACB** peut être définie comme suit :

```
(DEFUN RPLACB (OBJET LISTE)
    (RPLACA (RPLACD OBJET (CDR LISTE)) (CAR LISTE)))
```

Cette fonction s'appelle **DISPLACE** en LE_LISP.

21.16 CHAPITRE 17

1. Evidemment, les macro-fonctions **INCR** et **DECR** sont très proches. Les voici :

```
(DEFMACRO INCR (L) '(SETQ ,L (1+ ,L)))

(DEFMACRO DECR (L) '(SETQ ,L (1- ,L)))
```

2. Voici la fonction **SETNTH**, d'abord en Le_LISP :

```
(DEFMACRO SETNTH (WHERE WHICH WHAT)
    '(RPLACA (NTHCDR (1- ,WHICH) ,WHERE) ,WHAT))
```

et ensuite en CommonLISP :

```
(DEFMACRO SETNTH (WHERE WHICH WHAT)
    '(SETF (NTH (1- ,WHICH) ,WHERE) ,WHAT))
```

3. Voici la redéfinition de la fonction **SETQ** qui trace ses appels :

```
(DEFMACRO SETQ (VAR VAL &REST AUTRES)
    '(LET ((VAL (EVAL VAL)))
        (FORMAT T "~A == ~A~%" ',VAR VAL)
        (SETF ,VAR VAL)
        (IF ',AUTRES
            (EVAL (CONS 'SETQ ',AUTRES))
            VAL)))
```

21.17 CHAPITRE 18

1. Voici la fonction **MAPS** qui applique une fonction à toutes les sous-structures d'une liste :

```
(DEFUN MAPS (L F)(COND
    ((NULL L)())
    ((ATOM L)(F L))
    (T (F L)(MAPS (CAR L) F)(MAPS (CDR L) F)))
```

2. L'unique différence entre la fonction **WHILE** et la fonction **UNTIL** est le sens du test : il suffit de l'inverser. Ce qui donne, en Le_LISP :

```
(DEFMACRO UNTIL CALL
    '(IF ,(CADR CALL) ()
         ,@(CDDR CALL) ,CALL))
```

et en CommonLISP :

```
(DEFMACRO UNTIL (&WHOLE ALL &REST CALL)
  '(IF ,(CAR CALL) ()
       (PROGN ,@(CDR CALL) ,ALL)))))
```

3. La fonction **FACTORIELLE** avec l'itérateur **DO** :

```
(DEFUN FACTORIELLE (N)
    (DO ((N N (1- N))
         (R 1 (* N R)))
        ((LE N 1) R)))
```

4. Afin d'écrire ces deux macro-fonctions, **MAPC** et **MAPCAR**, regardons d'abord schématiquement la fonction **MAPCAR** :

Si l'on appelle cette fonction comme :

```
(MAPCAR 'CONS '(A B) (LIST '(C D) '(E F)))
```

Nous nous attendons au résultat suivant :

```
((A C D) (B E F))
```

une liste obtenue en prenant les CARs successifs de la première liste et en les *consant* aux cars successifs de la deuxième liste. **MAPCAR** doit donc au moins générer une instruction du style :

```
(CONS
   (CONS (car arg1)(car arg2))
   (MAPCAR 'CONS (cdr arg1)(cdr arg2)))
```

Bien entendu, dans le cas général nous ne savons pas combien d'arguments la fonction envoyée à **MAPCAR** attend. Dans l'exemple d'appel ci-dessus, la fonction envoyée est la fonction **CONS** à deux arguments. Nous, programmeurs LISP le savons, c'est pourquoi nous avons écrit

```
(CONS (car arg1)(car arg2))
```

mais la machine ne le sait pas. Nous devons donc générer autant d'appels de la fonction **CAR** (et ensuite dans l'appel récursif des fonctions **CDR**) qu'il y a d'arguments envoyés à **MAPCAR**, c.à.d.: autant d'appels de la fonction **CAR** qu'il y a d'éléments dans **L** si nous

commençons à écrire la définition de **MAPCAR** comme suit :

```
(DEFMACRO MAPCAR (FONC &REST L) ...
```

Comment générer autant d'appels de **CAR** qu'il y a d'éléments ? Le plus simple est probablement d'écrire une fonction spécialisée pour générer des appels de **CAR** (et ensuite d'utiliser cette même fonction pour générer les appels de **CDR**). Voici cette fonction auxiliaire :

```
(DEFUN GENERE (F L)
  (IF (NULL L) NIL
      (CONS (FUNCALL F (CAR L))(GENERE F (CDR L)))))
```

Cette fonction *est* la définition de la fonction **MAPCAR** que nous avons vue dans ce chapitre. Voila une preuve de son utilité.

Nous devons également prendre en compte des questions d'évaluation : les diverses listes d'arguments doivent d'abord être calculées, et c'est seulement ensuite que nous pouvons trouver tous les **CAR**s de ces listes. Mais, si nous évaluons les listes *avant* cette opération, nous serons bien obligés de quoter chacun des **CAR**s (ou **CDR**s) trouvés. Voici donc finalement la macro-fonction **MAPCAR** :

```
(DEFMACRO MAPCAR (FONC &REST L)
          (MAPCAR-AUX FONC (GENERE 'EVAL L)))

(DEFUN MAPCAR-AUX (FONC L)
    `(WHEN ',(CAR L)
        (CONS (FUNCALL ,FONC
                  ,@(GENERE
                      '(LAMBDA (X) (LIST 'QUOTE (CAR X))) L))
              (MAPCAR ,FONC
                  ,@(GENERE
                      '(LAMBDA (X)(LIST 'QUOTE (CDR X))) L)))))
```

La macro-fonction **MAPC** est la même, sans l'appel de **CONS** :

```
(DEFMACRO MAPC (FONC &REST L)
          (MAPC-AUX FONC (GENERE 'EVAL L)))

(DEFUN MAPC-AUX (FONC L)
    `(WHEN ',(CAR L)
          (FUNCALL ,FONC
              ,@(GENERE '(LAMBDA (X) (LIST 'QUOTE (CAR X))) L))
          (MAPC ,FONC
              ,@(GENERE '(LAMBDA (X)(LIST 'QUOTE (CDR X))) L))))
```

22. BIBLIOGRAPHIE

[Abelson85] H. Abelson et G. J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, Mass (1985).

[Allen78] J. Allen, *The anatomy of LISP*, Mc Graw Hill Inc (1978).

[Berkeley64] E. C. Berkeley et D. G. Bobrow, *The Programming Language LISP : Its Operation and Applications*, Information International Incorporated, Cambridge, MA (1964).

[Boyer73] R. S. Boyer et J. S. Moore, "Proving Theories about LISP Functions," pp. 486-493 dans *Proc. 3rd International Joint Conference on Artificial Intelligence*, Stanford, Ca (1973).

[Brooks85] R. A. Brooks, *Programming in CommonLISP*, John Wiley, New York, NY (1985).

[Burge75] W. H. Burge, *Recursive Programming Techniques*, Addison-Wesley, Reading, MA (1975).

[Chailloux78] J. Chailloux, "VLISP 10.3, Manuel de Référence," RT-16-78, Université Paris 8 - Vincennes (Avril 1978).

[Chailloux79] J. Chailloux, "VLISP 8.2, Manuel de Référence," RT 11-79, Université Paris 8 - Vincennes (Aout 1979).

[Chailloux80] J. Chailloux, *Le modèle VLISP : Description, Implémentation et Evaluation*, Université Paris 7, LITP 80-20 (Avril 1980). thèse de 3ième cycle

[Chailloux81] J. Chailloux, *LE_LISP de l'INRIA : Le Manuel de référence*, INRIA, LeChesnay (Dec. 1984).

[Charniak80] E. Charniak, C. K. Riesbeck, et D. V. Mc-Dermott, *Artificial intelligence programming*, Lawrence Erlbaum Associates, Hillsdale, NJ (1980).

[Cousot77] R. Cousot, "Automatic Synthesis of Optimal Invariant Assertions Mathematical Foundations," *ACM - SIGPLAN Notices* **8**(12) pp. 1-12 (1977).

[Durieux78] J. L. Durieux , "T.LISP IRIS 80," Rapport LSI No 130, Université Paul Sabatier-Toulouse (Juin 1978).

[Farreny84] H. Farreny, *Programmer en LISP,* Masson, Paris (1984).

[Friedman74] D. Friedman, *The Little LISPer,* Science Research Associates, Palo Alto, CA (1974).

[Gord64] E. Gord, "Notes on the Debugging of LISP Programs," dans *The Programming Language LISP: its Operation and Applications,* ed. E. C. Berkeley & D. G. Bobrow, The M.I.T. Press, Cambridge, Mass (1964).

[Greussay76] P. Greussay, "VLISP : Structures et extensions d'un système LISP pour mini-ordinateur," RT 16-76, Département Informatique, Université Paris 8 - Vincennes (Janvier 1976).

[Greussay76a] P. Greussay, "Iterative interpretations of tail-recursive LISP procedures," TR 20-76, Département Informatique, Université Paris 8 - Vincennes (Septembre 1976).

[Greussay77] P. Greussay, *Contribution à la définition interprétative et à l'implémentation de λ-langages,* Université Paris 7 (Novembre 1977). Thèse d'Etat

[Greussay78] P. Greussay, *Le Système VLISP-16,* Ecole Polytechnique (Decembre 1978).

[Greussay79] P. Greussay, "VLISP-11 : Manuel de Référence," Département Informatique, Université Paris 8 - Vincennes (1979).

[Greussay79a] P. Greussay, "Aides à la Programmation en LISP : outils d'observation et de compréhension," *Bulletin du Groupe Programmation et Langages,* (9) pp. 13-25 AFCET, Division Théorique et Technique de l'Informatique, (Octobre 1979).

[Greussay82] P. Greussay, "Le Système VLISP-UNIX," Département Informatique, Université Paris 8 - Vincennes (Février 1982). Manuel

[Henderson80] P. Henderson, *Functionnel Programming Application and Implementation,* Prentice Hall International, Englewood Cliffs (1980).

[Hoare69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM* **12**(2) pp. 567-581 (1969).

[Igarashi75] et D. C. Luckham, "Automatic Program Verification 1 : Logical Basis and its Implementation," *Acta Informatica* **4**(1) pp. 145-182 (1975).

[Knuth68] D. E. Knuth, *The Art of Computer Programming,* Vol. **1 Fundamental Algorithms**, Addison-Wesley Publ. Company, Reading, Mass (1968).

[Landin64] J. Landin, *The Mechanical Evaluation of Expressions,* Vol. **6**, Computer Journal (January 1964).

[Landin65] P. J. Landin, "A Correspondance between ALGOL60 and Church's Lambda-Notation," *Comm. ACM* **8**(2-3) pp. 89-101 et 158-165 (1965).

[Laubsch76] J. H. Laubsch, "MACLISP Manual," CUU-Memo-3, Universität Stuttgart, Stuttgart, RFA (1976).

[Lombardi64] L. A. Lombardi et B. Raphael, "LISP a Language for an Incremental Computer," dans *The Programming Language LISP: its Operation and Applications*, ed. E. C. Berkeley & D. G. Bobrow, The M.I.T. Press, Cambridge, Mass (1964).

[Maurer73] W. D. Maurer, *A Programmer's Introduction to LISP*, American Elsevier, New York, NY (1973).

[McCarthy62] J. McCarthy, P. Abrahams , D. J. Edwards, T. P. Hart, et M. I. Levin, *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass. (1962).

[McCarthy78] J. McCarthy et C. Talcott, *LISP - Programming and Proving*, Stanford University, Stanford, CA (1978). Draft

[Minsky67] M. L. Minsky, *Computation : Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, NJ (1967).

[Moon74] D. A. Moon, "MACLISP Reference Manual," MIT Project MAC, Cambridge Mass. (April 1974).

[Moon79] D. A. Moon et D. Weinreb, *LISP MACHINE MANUAL*, MIT, Cambridge, Mass. (January 1979).

[Perrot79] J. F. Perrot, "Sur la structure des interprètes LISP," dans *Colloque "Codage et Transductions"*, Florence (17-19 Octobre 1979).

[Quam72] L. H. Quam et W. Diffie, "Stanford LISP 1.6 Manual," SAILON 28.6, Computer Science Dept., Stanford University (1972).

[Queinnec82] C. Queinnec, *Langage d'un autre type : LISP*, Eyrolles, Paris (1982).

[Queinnec84] C. Queinnec, *LISP mode d'emploi*, Eyrolles, Paris (1984).

[Ribbens70] D. Ribbens, *Programmation Non Numérique : LISP 1.5*, Dunod, Paris (1970).

[Saint-James84] E. Saint-James, "Recursion is more Efficient than Iteration," pp. 228-234 dans *Proc. ACM LISP Conference 1984*, Austin, Texas (1984).

[Sandewall75] E. Sandewall, "Ideas about Management of LISP data bases," AI-Memo No 332, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass (1975).

[Siklossy76] L. Siklossy, *Let's Talk LISP,* Prentice Hall, Englewood Cliffs, NJ (1976).

[Steele79] G. L. Steele et G. J. Sussman, "Design of LISP-Based Processors," Memo 514, MIT Artificial Intelligence Laboratory (March 1979).

[Steele84] G. L. Steele, *CommonLISP, the Language,* Digital Press, Burlington, MA (1984).

[Stoyan78] H. Stoyan, *LISP - Programmierhandbuch,* Akademie Verlag, Berlin, RDA (1978).

[Stoyan80] H. Stoyan, *LISP - Anwendungsgebiete, Grundbegriffe, Geschichte,* Akademie Verlag, Berlin, RDA (1980).

[Stoyan84] H. Stoyan, *LISP Eine Einführung in die Programmierung,* Springer Verlag, Berlin Heidelberg New York (1984).

[Teitelman78] W. Teitelman, *INTERLISP Reference Manual,* Xerox Palo Alto Research Center, Palo Alto (October 1978). 3rd revision

[Weissman67] C. Weissman, *LISP 1.5 Primer,* Dickenson Publ. Comp., Belmont, CA (1967).

[Wertz83] H. Wertz, "An Integrated, Interactive and Incremental Programming Environment for the Development of Complex Systems," pp. 235-250 dans *Integrated Interactive Computing Systems,* ed. P. Degano & E. Sandewall, North-Holland, Amsterdam, New York, Oxford (1983).

[Wertz85] H. Wertz, *Intelligence Artificielle, Application à l'analyse de programmes (2ème édition),* Masson, Paris New York (1987).

[Wilensky84] R. Wilensky, *Common LISPcraft,* Norton & Co, New York, NY (1984).

[Winston77] P. H. Winston, *Artificial Intelligence,* Addison Wesley (1977).

[Winston81] P. H. Winston et B. K. P. Horn, *LISP,* Addison Wesley (1981).

23. LES FONCTIONS LISP STANDARD

Dans cette annexe nous résumons toutes les fonctions LISP standard rencontrées dans ce livre. Pour chaque fonction nous donnons la page où cette fonction a été définie, son type, le nombre de ses arguments et une courte description de sa sémantique. Nous distinguons trois types de fonctions standard : les SUBRs, les MACROs et les FSUBRs (en Le_LISP) ou les formes spéciales (en CommonLISP). Une fonction Le_LISP de type FSUBR se dit être une *special form* (ou *forme spéciale*) en CommonLISP. Les SUBRs sont des fonctions où chaque argument est évalué. Les MACROs sont des fonctions qui n'évaluent pas leurs arguments et sont écrites en utilisant la fonction **DEFMACRO**, c'est-à-dire qu'elles sont expansées en d'autres appels de fonctions. Finalement, les FSUBR ou les SPECIAL FORMs sont des fonctions standard à nombre d'arguments variable où les arguments ne sont pas obligatoirement évalués.

Le premier tableau concerne les fonctions **Le_LISP**, le deuxième les fonctions de **CommonLISP**.

Le_LISP

\	caractère pour quoter des caractères spéciaux (page 119)
\"	caractère pour délimiter les chaînes de caractères (page 68-69, 119)
#'	abbréviation pour **FUNCTION**, utilisée pour quoter des arguments fonctionnels (page 104)
#\	préfixe pour la représentation imprimée des caractères (page 81)
#\BELL	le caractère *bell*, i.e.: <ctrl>G (page 114)
#\BS	le caractère *backspace*, i.e.: <ctrl>H (page 114)
#\CR	le caractère *retour charriot* (page 114)
#\DEL	le caractère *delete* (page 114)
#\ESC	le caractère *escape* (page 114)
#\LF	le caractère *linefeed* (saut de ligne) (page 114)

#\SP le caractère *espace* (page 114)

#\TAB le caractère *tabulation* (page 114)

' (l'accent aigu) abbréviation pour **QUOTE**, utilisée pour quoter des expressions (page 5)

(et) les caractères pour délimiter des listes (page 1)

* (* n_1 ... n_N), SUBR à n arguments (page 41)
calcule le produit de ses arguments. Si aucun argument n'est donné, * retourne la valeur **1**.

+ (+ n_1 ... n_N), SUBR à n arguments (page 41)
calcule la somme de ses arguments. Si aucun argument n'est donné, + retourne la valeur **0**.

, indique, à l'intérieur d'un *backquote*, la valeur d'un élément (page 128)

,@ indique, à l'intérieur d'un *backquote*, la valeur d'un élément *sans les parenthèses externes* (page 128)

- (- $n1$ ... nN), SUBR à n arguments (page 41)
calcule la différence de ses arguments. Si aucun argument n'est donné, - retourne la valeur **0**.

/ (/ n_1 ... n_N), SUBR à N arguments (page 41)
divise successivement le premier argument par tous les autres et retourne ce résultat.

1+ (**1+** n), SUBR à 1 argument (page 40)
calcule la somme de n et **1**.

1- (**1-** n), SUBR à 1 argument (page 40)
calcule la différence de n et **1**.

; caractère pour débuter un commentaire. Tous les caractères suivants jusqu'à la fin de la ligne seront ignorés.

< (< n_1 ... n_N), SUBR à n arguments (page 43)
si $n1 < n_2 < ... < n_N$ alors < retourne n_1, sinon **NIL**

<= (<= n_1 ... n_N), SUBR à n arguments (page 43)
si $n1 \leq n_2 \leq ... \leq n_N$ alors <= retourne n_1, sinon **NIL**

= (= n_1 ... n_N), SUBR à n arguments (page 43)
si $n1 = n2 = ... = nN$ alors = retourne n_1, **sinon NIL**

`>` (**>** $n_1 \ldots n_N$), SUBR à *n* arguments (page 43)
si $n_1 > n_2 > \ldots > n_N$ alors > retourne n_1, **sinon NIL**

`>=` (**>=** $n_1 \ldots n_N$), SUBR à *n* arguments (page 43)
si $n1 \geq n_2 \geq \ldots \geq n_N$ alors >= retourne n_1, **sinon NIL**

ABS (**ABS** *n*), SUBR à 1 argument (page 204)
retourne la valeur absolue de **n**.

AND (**AND** $s_1 \ldots s_N$), FSUBR à *n* arguments (page 74)
retourne la valeur de s_N si l'évaluation de toutes les expressions s_i retourne une valeur différente de **NIL**, sinon **AND** retourne **NIL**

APPEND (**APPEND** $l_1 \ldots l_N$), SUBR à *n* arguments (page 31)
retourne la concatènation des listes l_1, $l_2 \ldots l_N$

APPLY (**APPLY** *fn* $s_1 \ldots s_N$ l_1)**fR, SUBR à 2 ou** *n* arguments (page 103-104)
les arguments $s_1 \ldots s_N$ sont optionnels. l_1 doit être une liste. **APPLY** retourne la valeur de l'application de la fonction *fn* à la liste l_1, qui est, si les arguments s_i sont présents, augmentée par ces arguments.

ASSOC (**ASSOC** *s al*), SUBR à 2 arguments (page 220)
s doit être un indicateur, *al* doit être une A-liste. **ASSOC** retourne la sous-liste de *al* commençant par *s*

ATOM (**ATOM** *s*), SUBR à 1 argument (page 20)
retourne **T** si *s* est un atome, sinon ce prédicat retourne **NIL**.

CADDR (**CADDR** *l*), SUBR à 1 argument (page 7)
retourne le troisième élément d'une liste

CADR (**CADR** *l*), SUBR à 1 argument (page 7)
retourne le deuxième élément d'une liste

CAR (**CAR** *l*), SUBR à 1 argument (page 6)
retourne le premier élément d'une liste

CDR (**CDR** *l*), SUBR à 1 argument (page 6)
retourne la liste *l1* sans son premier élément

CLOSE (**CLOSE** *chan*), SUBR à 0 ou 1 argument (page 138)
ferment le canal *chan* et retourne **T**. Si aucun argument n'est donné, **CLOSE** ferme tous les canaux ouverts

CONCAT (**CONCAT** $str_1 \ldots str_N$), SUBR à n arguments (page 82)
construit un symbole dont le nom est la concatènation de str_1, str_2, ... str_N

COND **(COND** l_1 ... l_N**)**, FSUBR (page 36)
COND évalue le premier élément de chaque liste l_1, l_2, ... l_i jusqu'à ce que l'évaluation retourne une valeur différente de **NIL**, ensuite, les expressions restantes de l_i sont évaluées et la valeur de la dernière expression de li est retournée. Si toutes les évaluations des *CAR*s successifs retournent *NIL*, **COND** retourne **NIL**

CONS **(CONS** s_1 s_2**)**, SUBR à 2 arguments (page 7, 152)
construit et retourne une nouvelle liste dont le CAR est la valeur de s_1 et dont le CDR est la valeur de s_2

CONSP **(CONSP** *s***)**, SUBR à 1 argument (page 21)
retourne **T** si *s* est une liste différente de **NIL**, donc : une liste non vide

DE **(DE** *symb lvar* s_1 ... s_N**)**, FSUBR (page 16, 88-91)
permet de construire une nouvelle fonction utilisateur du type EXPR. *symb* est le nom de cette nouvelle fonction. **DE** retourne *symb*

DECR **(DECR** *symb n***)**, FSUBR (page 294)
décrémente la valeur de *symb* et retourne la nouvelle valeur. Si un deuxième argument est donné, *symb* est décrémenté *n* fois

DEFMACRO **(DEFMACRO** *symb lvar* s_1 ... s_N**)**, FSUBR (page 191)
définit une nouvelle MACRO de nom *symb*. Retourne *symb*.

DEFVAR **(DEFVAR** *nom valeur-initiale***)**, MACRO, (page 189)
déclare la variable *nom* comme variable globale. Cette variable suit une discipline de *liaison dynamique*. Si un deuxième argument, *valeur-initiale*, est donnée, ce sera la valeur initiale de cette variable.

DELETE **(DELETE** *s l***)**, SUBR à 2 arguments (page 37)
retourne la liste *l* sans les occurrences (au premier niveau) de *s*

DF **(DF** *symb lvar* s_1 ... s_N**)**, FSUBR (page 91-92)
permet de construire une nouvelle fonction utilisateur de type FEXPR. **DF** retourne *symb*, le nom de cette nouvelle fonction.

DISPLACE **(DISPLACE** *l ln***)**, SUBR à 2 arguments (page 171)
remplace physiquement le CAR de la liste *l* par le CAR de la liste *ln* et le CDR de la liste *l* par le CDR de la liste *ln*.

DMC **(DMC** *ch () s1 ... sN***)**, FSUBR (page 123)
definit *ch* comme macro-caractère. Retourne *ch*.

DO **(DO** *lv lr* s_1 ... s_N**)**, MACRO (page 211)
fonction d'itération générale : *lv* est une liste de triplets :
<variable, valeur initiale, forme de répétition>.
lr est une liste dont le CAR est le test de sortie de l'itération et dont le CDR est

la suite d'expressions à évaluer quand l'évaluation du test donne une valeur différente de **NIL**. Quand le test donne **NIL**, les expressions *s1, s2 ... sN* sont évaluées. Ensuite les *formes de répétition* sont évaluées et affectées aux variables correspondantes.

EQ **(EQ** s_1 s_2**)**, SUBR à 2 arguments (page 34, 162)
compare les adresses de s_1 et s_2. Retourne **T** si elles sont identiques, autrement **EQ** retourne **NIL**

EQUAL **(EQUAL** s_1 s_2**)**, SUBR à 2 arguments (page 34, 162)
compare les structures s_1 et s_2. Retourne **T** si elles sont similaires, sinon **NIL**.

EVAL **(EVAL** *s***)**, SUBR à 1 argument (page 101-102)
retourne la valeur de *s*. C'est la fonction principale de l'interprète LISP.

EVENP **(EVENP** *n***)**, SUBR à 1 argument (page 43)
retourne *n* si *n* est un nombre pair, sinon **NIL**

EXIT **(EXIT** *tag expression***)**, FEXPR, (page 182-184)
comme la fonction **THROW** de CommonLISP, **EXIT** redonne le contrôle à une construction **TAG** du même *tag*. Retourne la valeur de *expression*, qui sera également la valeur du **TAG** correspondant (notez, qu'au contraire de CommonLISP, *tag* n'est pas évalué)

EXPLODECH **(EXPLODECH** *s***)**, SUBR à 1 argument (page 78)
retourne une liste de tous les caractères composant l'expression *s*

FORMAT **(FORMAT** *dest ctrls* s_1 s_2 ... s_N**)**, SUBR à N arguments (page 76-77)
FORMAT edite la chaîne de caractères *ctrls* dans le canal de sortie *dest*. *ctrls* peut contenir des directives formées par le caractère ~ suivi d'une lettre spécifiant cette directive. Les directives possibles sont :

~% insère un caractère *fin de ligne*

~A insère l'argument comme une chaîne de caractères

~D insère l'argument comme un nombre décimal

~O insère l'argument comme un nombre octal

~S insère l'argument comme une S-expression

~T insère une tabulation

~X insère l'argument comme un nombre hexadécimal

~~ insère le caractère tilde

FUNCALL **(FUNCALL** *fn* s_1 ... s_N**)**, SUBR à N arguments (page 203)
retourne la valeur de l'application de la fonction *fn* aux arguments s_1, s_2 ... s_N

GENSYM **(GENSYM)**, SUBR à 0 arguments (page 196)
retourne un nouveau symbole de la forme **G***zzz*

GET **(GET** *pl ind***)**, SUBR à 2 arguments (page 55)

GETPROP **(GETPROP** *pl ind***)**, SUBR à 2 arguments (page 55)
ces deux fonctions retournent la valeur associée à *ind* sur la P-liste du symbole *pl*.

IF **(IF** *t* s_1 s_2 ... s_N**)**, FSUBR (page 24)
le premier argument, *t* est considéré comme un test. Si son évaluation donne une valeur non-**NIL**, **IF** retourne l'évaluation de s_1. Si l'évaluation de *t* donne **NIL**, les expressions s_2, s_3, ... s_N sont évaluées séquentiellement et la valeur de s_N est retournée.

IMPLODECH **(IMPLODECH** *s***)**, SUBR à 1 argument (page 80)
s est une liste de caractères. **IMPLODECH** retourne l'expression résultant de la lecture LISP de ces caractères

INCHAN **(INCHAN** *chan***)**, SUBR à 0 ou 1 argument (page 138)
associe le canal *chan* au flux d'entrée Si aucun argument n'est donné, **INCHAN** retourne le canal courant.

INCR **(INCR** *symb n***)**, FSUBR (page 294)
incrémente la valeur de *symb* et retourne la nouvelle valeur. Si le deuxième argument est donné, *symb* est incrémenté *n* fois.

LAMBDA **(LAMBDA** *lvar* s_1 ... s_N**)**, FSUBR (page 205)
la valeur d'une lambda expression est la lambda expression même

LAST **(LAST)**, SUBR à 1 argument (page 160)
retourne le dernier doublet de la liste *s*. Si *s* n'est pas une liste **LAST** retourne l'argument *s*.

LENGTH **(LENGTH** *s***)**, SUBR à 1 argument (page 81, 272)
retourne le nombre d'éléments de la liste *s*. Si *s* n'est pas une liste, **LENGTH** retourne 0

LET **(LET** *lvar* s_1 ... s_N**)**, FSUBR (page 45-46)
appelle une fonction anonyme du type EXPR. *lvar* est une liste de couples <variable, valeur-initiale>. Toutes les variables sont liées à leur valeur initiale, puis les expressions s_1 ... s_N sont évaluées.

LETN (**LETN** *symb lvar* s_1 ... s_N), FSUBR (page 47)
Comme **LET** mais permet de donner *symb* comme nom temporaire à la fonction

LIST (**LIST** s_1 ... s_N), SUBR à N arguments (page 90)
retourne la liste composée des s_1, s_2, ... sN

LOAD (**LOAD** *fichier*), FSUBR (page 133)
charge le fichier *fichier* et retourne le nom du fichier

LOADFILE (**LOADFILE** *fichier i*), SUBR à 2 arguments (page 138)
comme la fonction précédente, excepté : que 1) les arguments sont évalués et 2) le deuxième argument *i* est un indicateur dont le positionnement permet ou non la redéfinition de fonctions.

MACROEXPAND
(**MACROEXPAND** *s*), SUBR à 1 argument (page 198)
retourne l'expression *s* où tous les appels à des macro-fonctions ont été expansés. Cette fonction est utilisée pour tester la correction des macro-fonctions.

MAPC (**MAPC** *fn* l_1 ... l_N), SUBR à N arguments (page 207)
lance la fonction *fn* avec tous les CARs des listes l_1 ... l_N comme arguments et retourne **NIL**

MAPCAR (**MAPCAR** *fn* l_1 ... l_N), SUBR à N arguments (page 204)
comme **MAPC**, mais retourne la liste des applications fonctionelles successives.

MAX (**MAX** n_1 ... n_N), SUBR à N arguments (page 284)
retourne le plus grand nombre de n_1, n_2, ... n_N

MCONS (**MCONS** s_1 ... s_N), SUBR à N arguments (page 193)
retourne la liste obtenue en CONSant successivement tous les éléments *si*.

MEMQ (**MEMQ** *symb l*), SUBR à 2 arguments (page 268)
si *symb* est un élément de la liste *l*, **MEMQ** retourne la sous-liste débutant par *symb*, autrement elle retourne **NIL** (les comparaisons sont faites avec **EQ**)

MEMBER (**MEMBER** *s l*), SUBR à 2 arguments (page 269)
comme **MEMQ**, mais les comparaisons sont faites avec **EQUAL**.

MIN (**MIN** n_1 ... n_N), SUBR à N arguments (page 285)
retourne le plus petit nombre de n_1, n_2, ... n_N

MUL (**MUL** n_1 n_2), SUBR à 2 arguments (page 41)
retourne le résultat de la multiplication de n_1 et n_2. Les deux nombres doivent être des entiers.

NCONC **(NCONC** l_1 ... l_N**)**, SUBR à N arguments (page 160)
concatène physiquement toutes les listes l_i et retourne la nouvelle liste l_1.

NCONC1 **(NCONC1** *l a*), SUBR à 2 arguments (page 201)
ajoute physiquement à la liste *l* un nouveau dernier élément *a*. Retourne la nouvelle liste *l*

NEQUAL **(NEQUAL** s_1 s_2**)**, SUBR à 2 arguments (page 124)
c'est la même chose que **(NULL (EQUAL** s_1 s_2**))**

NEXTL **(NEXTL** $symb_1$ $symb_2$**)**, FSUBR (page 171)
$symb_1$ et $symb_2$ doivent être des symboles. **NEXTL** retourne le CAR de la valeur de $symb_1$ et affecte à $symb_1$ le CDR de sa valeur. Si $symb_2$ est présent, il reçoit la valeur retournée par **NEXTL**.

NTH **(NTH** *n l*), SUBR à 2 arguments (page 270)
retourne le *n*ième élément de *l*

NULL **(NULL** *s*), SUBR à 1 argument (page 22)
retourne **T** si *s* s'évalue à **NIL**, sinon **NIL**

NUMBERP **(NUMBERP** *s*), SUBR à 1 argument (page 21)
retourne *s* si c'est un nombre, sinon **NIL**

OBLIST **(OBLIST)**, SUBR à 0 arguments (page 115)
retourne la liste de *tous* les symboles connus

ODDP **(ODDP** *n*), SUBR à 1 argument (page 43)
retourne *n* si c'est un nombre impair, sinon **NIL**

OPENI **(OPENI** *fichier*), SUBR à 1 argument (page 138)
ouvre le fichier *fichier* pour la lecture et retourne le canal associé à ce fichier.

OPENO **(OPENO** *fichier*), SUBR à 1 argument (page 138)
ouvre le fichier *fichier* pour l'écriture et retourne le canal associé à ce fichier.

OR **(OR** s_1 ... s_N**)**, FSUBR (page 74)
évalue un *si* après l'autre jusqu'à ce qu'une valeur non-**NIL** soit obtenue ou jusqu'à la fin des s_i. **OR** retourne la valeur de la première expression s_i qui est différente de **NIL**. Si toutes les expressions s'évaluent à **NIL**, **OR** retourne **NIL**.

OUTCHAN **(OUTCHAN** *chan*), SUBR à 0 ou 1 argument (page 138)
associe le canal *chan* au flux de sortie courant, c'est-à-dire que toutes les sorties suivantes se feront sur ce canal. Si aucun argument n'est donné, **OUTCHAN** retourne **NIL**

PEEKCH (**PEEKCH**), SUBR à 0 arguments (page 111)
retourne le caractère suivant du flux d'entrée *sans* l'enlever

PLENGTH (**PLENGTH** *str*), SUBR à 1 argument (page 79)
retourne la longueur de la chaîne de caractères *str*

PLIST (**PLIST** *pl l*), SUBR à 1 ou 2 arguments (page 52)
si *l* n'est pas donné, **PLIST** retourne la P-liste de *pl*, sinon la liste *l* devient la nouvelle P-liste de *pl*

PLUS (**PLUS** n_1 ... n_N), SUBR à N arguments (page 89)
retourne la somme de n_1, n_2, ... n_N. Sans argument, **PLUS** retourne 0

PRIN (**PRIN** s_1 ... s_N), SUBR à N arguments (page 70)
prépare les expressions s_1, s_2, ... s_N pour l'impression. L'impression réelle n'aura lieu que quand le buffer de sortie sera vidé.

PRINCH (**PRINCH** *ch n*), SUBR à 1 ou 2 arguments (page 72)
édite *n* fois le caractère *ch* dans le flux de sortie, retourne *ch*

PRINT (**PRINT** s_1 ... s_N), SUBR à N arguments (page 63)
édite et imprime les expressions s_1, s_2, ... s_N dans le flux de sortie

PROBEFILE (**PROBEFILE** *fichier*), SUBR à 1 argument (page 138)
retourne **T** si le fichier *fichier* existe, sinon **NIL**

PROGN (**PROGN** s_1 ... s_N), SUBR à N arguments (page 169)
évalue successivement toutes les expressions s_i et retourne la valeur de s_N

PUTPROP (**PUTPROP** *pl s ind*), SUBR à 3 arguments (page 54)
associe *s* à *ind* sur la P-liste de *pl*. **PUTPROP** retourne la valeur de *s*

QUO (**QUO** n_1 n_2), SUBR à 2 arguments (page 41)
retourne le quotient entier de n_1 / n_2

QUOTE (**QUOTE** *s*), FSUBR (page 5)
s est retourné tel quel, non-évalué

RANDOM (**RANDOM** n_1 n_2), SUBR à 2 arguments (page 204)
retourne une valeur aléatoire entre n_1 et n_2

READ (**READ**), SUBR à 0 arguments (page 107)
lit et retourne l'expression suivante du flux d'entrée

READCH (**READCH**), SUBR à 0 arguments (page 110-111)
lit et retourne le caractère suivant du flux d'entrée

REM **(REM** n_1 n_2**)**, SUBR à 2 arguments (page 41)
retourne le reste de la division entière de n_1 / n_2

REMPROP **(REMPROP** *pl ind*), SUBR à 2 arguments (page 56)
enlève de la P-liste de *pl* l'indicateur *ind* et la valeur associée. Retourne l'ancienne P-liste si l'indicateur *ind* existe, sinon **NIL**

REVERSE **(REVERSE** *l*), SUBR à 1 argument (page 28)
retourne la liste *l* avec ses éléments dans l'ordre inverse

RPLACA **(RPLACA** *l s*), SUBR à 2 arguments (page 156)
remplace physiquement le CAR de *l* avec la valeur de *s* et retourne la liste *l* modifiée.

RPLACD **(RPLACD** *l s*), SUBR à 2 arguments (page 156)
remplace physiquement le CDR de *l* avec la valeur de *s* et retourne la liste *l* modifiée.

SET **(SET** *sym s*), SUBR à 2 arguments (page 167)
change la valeur de la valeur de sym en *s*

SETQ **(SETQ** sym_1 s_1 ... sym_N s_N), FSUBR (page 166)
sym_1, ..., sym_N sont des symboles (ces arguments ne sont pas evalués) s_1, ..., s_N sont des expressions. **SETQ** affecte à chaque sym_i la valeur correspondante de s_i. **SETQ** retourne la valeur de s_N.

SQRT **(SQRT** *n*), SUBR à 1 argument (page 204)
retourne la racine carrée de *n*

STRING **(STRING** *s*), SUBR à 1 argument (page 81)
retourne la chaîne de caractères résultant de la conversion de *s* en chaîne de caractères.

STRINGP **(STRINGP** *s*), SUBR à 1 argument (page 69)
retourne **T** si *s* est une chaîne de caractères, sinon **NIL**

SYMEVAL **(SYMEVAL** *sym*), SUBR à 1 argument (page 52)
retourne la valeur du symbole *sym*. C'est la même chose que **(EVAL** *symb*) en plus efficace.

TAG **(TAG** *tag* $expr_1$ ... $expr_n$), Special Form, (page 182-184)
semblable à la fonction **CATCH** de CommonLISP, évalue séquentiellement $expr_1$ à $expr_n$, jusqu'à la rencontre d'une forme **EXIT** du même *tag*. Dans ce cas **TAG** arrête l'évaluation des $expr_i$ et sort en ramenant la valeur de **EXIT**, sinon **TAG** retourne la valeur de $expr_n$ (notez qu'au contraire de CommonLISP, *tag* n'est pas évalué).

TERPRI (**TERPRI** *n*), SUBR à 0 ou 1 argument (page 71)
effectue 1 ou *n newlines*. Retourne *n*

TIMES (**TIMES** n_1 ... n_N), SUBR à N arguments (page 284)
retourne le produit de $1 * n_1 * n_2 * ... * n_N$

TYI (**TYI**), SUBR à 0 arguments (page 112)
lit un caractère au clavier et retourne son code interne

TYO (**TYO** o_1 ... o_N), SUBR à N arguments (page 112)
edite les caractères o_1, ... o_N sur le terminal

TYPECH (**TYPECH** *c symb*), SUBR à 1 ou 2 arguments (page 121)
retourne le type du caractère *c* si *symb* est donné, sinon **TYPECH** change le type du caractère *c* en *symb*

UNLESS (**UNLESS** *tst* s_1 ... s_N), FSUBR (page 60)
si l'évaluation de *tst* donne **NIL**, **UNLESS** évalue en séquence s_1, s_2, ... s_N et retourne la valeur de s_N. Sinon **UNLESS** retourne **NIL**

UNTIL (**UNTIL** *tst* s_1 ... s_N), FSUBR (page 210)
tant que l'évaluation de *tst* donne **NIL**, **UNTIL** évalue en séquence s_1, ... s_N. **UNTIL** retourne la première valeur de **tst** différente de **NIL**

WHEN (**WHEN** *tst* s_1 ... s_N), FSUBR (page 61)
si l'évaluation de *tst* donne une valeur non-**NIL**, **WHEN** évalue en séquence s_1, s_2, ... s_N et retourne la valeur de s_N. Sinon **WHEN** retourne **NIL**

WHILE (**WHILE** *tst* s_1 ... s_N), FSUBR (page 210)
tant que l'évaluation de *tst* donne une valeur non-**NIL**, **WHILE** évalue en séquence s_1, ... s_N. **WHILE** retourne **NIL** quand l'évaluation de *tst* donne **NIL**

ZEROP (**ZEROP** *n*), SUBR à 1 argument (page 42)
retourne **T** si *n* est égal à 0, **NIL** sinon.

‘ le macro-caractère *backquote* (page 128)

| le caractère d'échappement multiple (il sert à délimiter des noms de symboles comportant des caractères spéciaux) (page 79, 120)

’ le macro-caractère **QUOTE** (page 5)

CommonLISP

\ caractère pour quoter les caractères spéciaux (page 119, 122)

\" caractère pour délimiter des chaînes de caractères (page 68-69, 119-122)

#' abréviation pour **FUNCTION**, utilisée pour quoter des arguments fonctionnels (page 104)

#\ préfixe pour la représentation imprimée des caractères (page 81)

#\Newline le caractère *newline* (page 114)

#\Page le caractère *à la page* (page 114)

#\Return le caractère *retour chariot* (page 114)

#\Space le caractère *espace* (page 114)

#\Tab le caractère *tabulation* (page 114)

&KEY le *mot clef* de la liste des variables (page 97)

&OPTIONAL le mot-clef *optionnel* de la liste des variables (page 95)

&REST le mot-clef *reste* de la liste des variables (page 93)

&WHOLE le mot clef *tout* de la liste des variables (ce mot clef ne peut se trouver que dans les listes de variables des macro-fonctions) (page 210)

' abréviation pour **QUOTE**, utilisée pour quoter des expressions (page 5)

(et) caractères pour délimiter des listes (page 1)

* (* $n_1 \ldots n_N$), SUBR à n arguments (page 41)
calcule le produit de ses arguments. Sans argument * retourne la valeur **1**.

+ (+ $n_1 \ldots n_N$), SUBR à n arguments (page 41)
calcule la somme de ses arguments. Sans argument, + retourne la valeur **0**.

, indique, à l'intérieur d'un *backquote*, la valeur d'un élément (page 128)

,@ indique, à l'intérieur d'un *backquote*, la valeur d'un élément *sans les parenthèses externes* (page 128)

– (– n_1 ... n_N), SUBR à n arguments (page 41)
calcule la différence de ses arguments. Sans argument, - retourne la valeur **0**.

/ (/ n_1 ... n_N), SUBR à N arguments (page 41)
divise successivement le premier argument par tous les autres et retourne ce résultat. Avec seulement un argument, / donne sa valeur réciproque. / produit un nombre rationnel si le quotient de deux entiers n'est pas exactement un entier.

1+ (1+ n), SUBR à 1 argument (page 40)
calcule la somme de n et **1**.

1– (1– n), SUBR à 1 argument (page 40)
calcule la différence de n et **1**.

: indicateur de constante, utilisé principalement dans les paramètres mots-clef (page 119)

:DIRECTION mot-clef pour des opérations d'entrées/sorties (page 133)

:INPUT mot-clef pour des opérations d'entrées (page 134)

:IO mot-clef pour des opérations d'entrées et de sorties (page 134)

:OUTPUT mot-clef pour des opérations de sorties (page 133)

:PROBE mot-clef pour tester l'existence de fichiers (page 134)

; caractère pour débuter un commentaire. Tous les caractères suivants jusqu'à la fin de la ligne seront ignorés. (page 119)

< (< n_1 ... n_N), SUBR à n arguments (page 43)
si $n_1 < n_2 < ... < n_N$ retourne **T**, **NIL** sinon

<= (<= n_1 ... n_N), SUBR à n arguments (page 43)
si $n_1 \leq n_2 \leq ... \leq n_N$ retourne **T**, **NIL** sinon

= (= n_1 ... n_N), SUBR à n arguments (page 43)
si $n_1 = n_2 = ... = n_N$ retourne **T**, **NIL** sinon

> (> n_1 ... n_N), SUBR à n arguments (page 43)
si $n_1 > n_2 > ... > n_N$ retourne **T**, **NIL** sinon

>= (**>=** $n_1 \ldots n_N$), SUBR à *n* arguments (page 43)
si $n_1 \geq n_2 \geq \ldots \geq n_N$ retourne **T**,
NIL sinon

ABS (**ABS** *n*), SUBR à 1 argument (page 204)
retourne la valeur absolue de **n**.

AND (**AND** $s_1 \ldots s_N$), MACRO à *n* arguments (page 74)
retourne la valeur de s_N si l'évaluation de toutes les expressions s_i retourne une valeur différente de **NIL**, sinon **AND** retourne **NIL**

APPEND (**APPEND** $l_1 \ldots l_N$), SUBR à *n* arguments (page 31)
retourne la concatènation des listes l_1, l_2 ... l_N.

APPLY (**APPLY** *fn* $s_1 \ldots s_N$ *l1*) **fR, SUBR à 2 ou** *n* arguments (page 103-104)
les arguments s_1 ... s_N sont optionnels. l_1 doit être une liste. **APPLY** retourne la valeur de l'application de la fonction *fn* à la liste *l1*, qui est, si les arguments s_i sont présents, augmentée de ces arguments.

ASSOC (**ASSOC** *s al*), SUBR à 2 arguments (page 220)
s doit être un indicateur, *al* doit être une A-liste. **ASSOC** retourne la sous-liste de *al* commençant par *s*

ATOM (**ATOM** *s*), SUBR à 1 argument (page 20)
retourne **T** si *s* est un atome, sinon ce prédicat retourne **NIL**.

CADDR (**CADDR** *l*), SUBR à 1 argument (page 7)
retourne le troisième élément d'une liste

CADR (**CADR** *l*), SUBR à 1 argument (page 7)
retourne le deuxième élément d'une liste

CAR (**CAR** *l*), SUBR à 1 argument (page 6)
retourne le premier élément d'une liste

CATCH (**CATCH** *tag* $expr_1$ ... $expr_n$), Special Form, (page 182-184)
évalue *tag* et séquentiellement $expr_1$ à $expr_n$, jusqu'à la rencontre d'une forme **THROW** du même *tag*. Dans ce cas **CATCH** arrête l'evaluation des $expr_i$ et sort en ramenant la valeur de **THROW**, sinon **CATCH** retourne la valeur de $expr_n$

CDR (**CDR** *l*), SUBR à 1 argument (page 6)
retourne la liste *l1* sans son premier élément

CHAR> (**CHAR>** c_1 ... c_N), SUBR à N arguments (page 203)
retourne **T** si $c_1 > c_2 > \ldots > c_N$, sinon **NIL**

CHARACTERP
(CHARCACTERP *s*), SUBR à 1 argument
retourne **T** si *s* est caractère, sinon **NIL**

CLOSE **(CLOSE** *flux*), SUBR à 0 ou 1 argument (page 135)
ferme le flux *flux* et retourne **T**.

CONCATENATE
(CONCATENATE *type-de-résultat* seq_1 ... seq_N), SUBR à N arguments (page 82)
le résultat est la concatènation de toutes les séquences seq_1 à seq_N. *type-de-résultat* spécifie le type du résultat qui doit être un sous-type de séquence.

COND **(COND** l_1 ... l_N), MACRO (page 36)
évalue le premier élément de chaque liste l_1, l_2, ... l_i jusqu'à ce que l'évaluation retourne une valeur différente de **NIL**, puis évalue les expressions restantes de l_i et retourne la valeur de la dernière expression de l_i. Si toutes les évaluations des *CARs* successifs retournent *NIL*, **COND** retourne **NIL**

CONS **(CONS** s_1 s_2), SUBR à 2 arguments (page 7, 152)
construit et retourne une nouvelle liste dont le CAR est la valeur de s_1 et dont le CDR est la valeur de s_2

CONSP **(CONSP** *s*), SUBR à 1 argument (page 21)
retourne **T** si *s* est une liste différente de **NIL**

DECLARE **(DECLARE** (*spécif* var_1 ... var_N)) special form (page 217)
forme spéciale qui permet d'insérer des déclarations dans du code exécutable. La seule *spécification* de type que nous avons vue dans ce livre est le type **SPECIAL**, qui déclare que les variables var_1, ..., var_N doivent obéir au mécanisme de liaison dynamique.

DEFMACRO **(DEFMACRO** *symb lvar* s_1 ... s_N), MACRO (page 191)
définit une nouvelle MACRO de nom *symb*. Retourne *symb*.

DEFUN **(DEFUN** *symb lvar* s_1 ... s_N), MACRO (page 16)
permet de construire une nouvelle fonction utilisateur de nom *symb*. **DEFUN** retourne *symb*

DEFVAR **(DEFVAR** *nom valeur-initiale documentation*), MACRO, (page 189)
déclare la variable *nom* comme variable globale. Cette variable suit une discipline de *liaison dynamique*. Avec un deuxième argument *valeur-initiale*, ce sera la valeur initiale de cette variable. *valeur-initiale* n'est pas évaluée avant l'utilisation effective de cette variable. L'argument optionnel *documentation* associe la chaîne de caractères *documentation* à *nom*. Cette chaîne est accessible avec la fonction **DOCUMENTATION**

DELETE **(DELETE** *s l***)**, SUBR à 2 arguments (page 37, 81)
retourne la liste *l* sans les occurrences (au premier niveau) de *s*

DO **(DO** *lv lr* s_1 ... s_N**)**, MACRO (page 211)
fonction d'itération générale : *lv* est une liste de triplets :
<variable, valeur initiale, forme de répétition>.
lr est une liste dont le CAR est le test de sortie de l'itération et dont le CDR est la suite d'expressions à évaluer quand l'évaluation du test donne une valeur différente de **NIL**. Quand le test donne **NIL**, les expressions s_1, s_2 ... s_N sont évaluées. Ensuite les *formes de répétition* sont évaluées et affectées aux variables correspondantes.

DOCUMENTATION
(DOCUMENTATION *symbole type***)**, SUBR à 2 arguments, (page 189)
retourne la chaîne de caractère de documentation de type *type* associée au symbole *symbole*. Nous ne l'utilisons, dans ce livre, qu'avec la forme spéciale **DEFVAR**. Le type de documentation est alors **VARIABLE**

ELT **(ELT** *seq n***)**, SUBR à 2 arguments (page 82)
retourne le *n*ième élément de la séquence *seq*. Notez que le premier élément d'une séquence est à l'index 0.

EQ **(EQ** s_1 s_2**)**, SUBR à 2 arguments (page 34, 162)
compare les adresses de s_1 et s_2. Retourne **T** si elles sont identiques, **NIL** sinon.

EQUAL **(EQUAL** s_1 s_2**)**, SUBR à 2 arguments (page 34, 162)
compare les structures s_1 et s_2. Retourne **T** si elles sont similaires, sinon **NIL**.

EVAL **(EVAL** *s***)**, SUBR à 1 argument (page 101-102)
retourne la valeur de *s*. C'est la fonction principale de l'interprète LISP.

EVENP **(EVENP** *n***)**, SUBR à 1 argument (page 43)
retourne **T** si *n* est un nombre pair, sinon **NIL**

FLOOR **(FLOOR** n_1 n_2**)**, SUBR à 1 ou 2 arguments (page 42, 49)
avec un seul argument, **FLOOR** retourne le plus grand entier égal ou inférieur à n_1. Avec deux arguments, **FLOOR** retourne le plus grand entier égal ou inférieur au quotient n_1 / n_2

FORMAT **(FORMAT** *dest ctrls* s_1 s_2 ... s_N**)**, SUBR à N arguments (page 76-77)
FORMAT édite la chaîne de caractères *ctrls* dans le canal de sortie *dest*. *ctrls* peut contenir des directives formées par le caractère ~ suivi d'une lettre spécifiant la directive. Les directives possibles sont :

~% insère un caractère *fin de ligne*

~A insère l'argument comme une chaîne de caractères

~D insère l'argument comme un nombre décimal

~O insère l'argument comme un nombre octal

~S insère l'argument comme une S-expression

~T insère une tabulation

~X insère l'argument comme un nombre hexadécimal

~~ insère le caractère tilde

FUNCALL **(FUNCALL** *fn* s_1 ... s_N**)**, SUBR à N arguments (page 203)
retourne la valeur de l'application de la fonction *fn* aux arguments s_1, s_2 ... s_N

GENSYM **(GENSYM)**, SUBR à 0 arguments (page 196)
retourne un nouveau symbole de la forme **Gzzz**

GET **(GET** *pl ind def***)**, SUBR à 2 ou 3 arguments (page 55)
sans le troisième argument, cette fonction retourne la valeur associée à *ind* sur la P-liste du symbole *pl*. Le troisième argument donne la valeur par défaut à retourner si la valeur est **NIL**. Ceci permet de distinguer entre **NIL** en tant que valeur associée à *ind* ou en tant que valeur retournée par **GET** si *ind* n'existe pas sur la P-liste de **pl**.

GET-MACRO-CHARACTER
(GET-MACRO-CHARACTER *c***)**, SUBR à 1 argument (page 128)
retourne la fonction associée au caractère *c* ou **NIL** si *c* n'a pas de fonction associée

IF **(IF** *test then else***)**, Special Form (page 24)
le premier argument, *test* est considéré comme un test. Si son évaluation donne une valeur non-**NIL**, **IF** retourne l'évaluation de *then* la valeur de *else* sinon.

LABELS **(LABELS** *sym lvar* s_1 ... s_N**)**, Special Form (page 47)
permet la définition locale d'une fonction *sym*. *lvar* est la liste de variables et s_1, ..., s_N est le corps de la fonction.

LAST **(LAST** *s***)**, SUBR à 1 argument (page 160)
retourne le dernier doublet de la liste *s*.

LENGTH **(LENGTH** *s***)**, SUBR à 1 argument (page 81, 272)
retourne le nombre d'éléments de la séquence *s*.

LET **(LET** *lvar* s_1 ... s_N**)**, Special Form (page 45-46)
appelle une fonction anonyme du type EXPR. *lvar* est une liste de couples <variable, valeur-initiale>. Toutes les variables sont liées à leur valeur initiale, puis les expressions s_1 ... s_N sont évaluées.

LIST **(LIST** s_1 ... s_N**)**, SUBR à N arguments (page 90)
retourne la liste composée de s_1, s_2, ... s_N

LOAD **(LOAD** *fichier*), SUBR (page 133)
charge le fichier *fichier* et retourne le nom du fichier

MACROEXPAND
(MACROEXPAND *s*), SUBR à 1 argument (page 198)
retourne l'expression *s* où tous les appels à des macro-fonctions ont été expansés. Cette fonction est utilisée pour tester la correction des macro-fonctions.

MAPC **(MAPC** *fn* l_1 ... l_N**)**, SUBR à N arguments (page 207)
lance la fonction *fn* avec tous les CARs des listes l_1 ... l_N comme arguments et retourne **NIL**

MAPCAR **(MAPCAR** *fn* l_1 ... l_N**)**, SUBR à N arguments (page 204)
comme **MAPC**, mais retourne la liste des applications fonctionelles successives.

MAX **(MAX** n_1 ... n_N**)**, SUBR à N arguments (page 285)
retourne la plus grande valeur de n_1, n_2, ... n_N

MEMBER **(MEMBER** *s l*), SUBR à 2 arguments (page 268)
si *s* est un élément de la liste *l*, **MEMBER** retourne la sous-liste débutant par *s*, sinon **NIL** (les comparaisons sont faites avec **EQUAL**)

MIN **(MIN** n_1 ... n_N**)**, SUBR à N arguments (page 285)
retourne le plus petit nombre de n_1, n_2, ... n_N

NCONC **(NCONC** l_1 ... l_N**)**, SUBR à N arguments (page 160)
concatène physiquement toutes les listes l_i et retourne la liste l_1 modifiée.

NTH **(NTH** *n l*), SUBR à 2 arguments (page 270)
retourne le *n*ième élément de *l*. Le CAR de *l* est le *zero*-ième élément.

NULL **(NULL** *s*), SUBR à 1 argument (page 22)
retourne **T** si *s* s'évalue à **NIL**, sinon **NIL**

NUMBERP **(NUMBERP** *s*), SUBR à 1 argument (page 21)
retourne **T** si *s* est un nombre, **NIL** sinon

ODDP **(ODDP** *n*), SUBR à 1 argument (page 43)
retourne **T** si *n* est un nombre impair, **NIL** sinon

OPEN **(OPEN** *fichier* **:DIRECTION** *dir***)**, SUBR à 2 arguments (page 133)
retourne un flux connecté au fichier *fichier*. Quatre directions sont possibles : **:INPUT**, retourne un flux d'entrée, **:OUTPUT** retourne un flux de sortie, **:IO**, retourne un flux bidirectionnel et **:PROBE** retourne **NIL** si le fichier *fichier* n'existe pas et un flux non-directionnel sinon.

OR **(OR** s_1 ... s_N**)**, MACRO (page 74)
évalue les s_i l'un après l'autre jusqu'à ce qu'une valeur non-**NIL** soit obtenue ou jusqu'à la fin des s_i.

PEEK-CHAR **(PEEK-CHAR** *peek-type fluz-d'entrée eof-erreur eof-val rec-p***)** SUBR de 0 à 5 arguments (page 111)
sans arguments **PEEK-CHAR** retourne le caractère suivant du flux en le laissant accessible pour des lectures ultérieures. **PEEK-CHAR** saute des caractères blancs avant d'opérer la lecture. Si *peek-type* est un caractère, alors **PEEK-CHAR** saute tous les caractères d'entrée jusqu'à la rencontre d'un caractère égal à l'argument *peek-type*. Si *fluz-d'entrée* est non-**NIL**, alors l'opération de lecture a lieu dans le flux spécifié. Si *eof-erreur* est **T** (la valeur par défaut) la lecture après un fin de fichier signale une erreur, sinon aucune erreur n'est signalée et *eof-val* est retournée. Si *rec-p* est non-**NIL**, cela spécifie que l'appel de **PEEK-CHAR** est à l'intérieur d'un appel à **READ** comme c'est le cas si l'on appelle **PEEK-CHAR** de l'intérieur d'une définition de macro-caractère.

PRIN1 **(PRIN1** *s fluz***)**, SUBR à 1 ou 2 arguments (page 71, 136)
imprime *s* sur le flux *fluz* (par défaut la sortie standard). **PRIN1** imprime des caractères d'échappement et les guillemets de sorte que la sortie soit lisible pour le lecteur LISP.

PRINC **(PRINC** *s fluz***)**, SUBR à 1 ou 2 arguments (page 72, 136)
imprime *s* sur le flux *fluz* (par défaut la sortie standard). Aucun caractère d'échappement ni les guillemets ne sont imprimés, de sorte que la sortie soit lisible pour l'utilisateur humain.

PRINT **(PRINT** *s fluz***)**, SUBR à 1 ou 2 arguments (page 63, 136)
comme **PRIN1**, sauf que la sortie est précédée par un retour chariot.

PROGN **(PROGN** s_1 ... s_N**)**, SUBR à N arguments (page 169)
évalue successivement toutes les expressions s_i et retourne la valeur de s_N.

QUOTE **(QUOTE** *s***)**, Special Form (page 5)
s est retourné tel quel, non-évalué

RANDOM **(RANDOM** n_1 n_2**)**, SUBR à 2 arguments (page 204)
retourne une valeur aléatoire entre n_1 et n_2

READ **(READ** *fluz-d'entrée eof-erreur eof-val rec-p***)**, SUBR de 0 à 4 arguments (page 107, 125, 136)
retourne l'expression LISP suivante lue dans le flux *fluz-d'entrée* (par défaut

l'entrée standard); **T** désigne le terminal. Les autres arguments ont la même signification que pour **PEEK-CHAR**.

READ-CHAR
(READ-CHAR *flux-d'entrée eof-erreur eof-val rec-p*), SUBR de 0 à 4 arguments (page 111)
comme **PEEK-CHAR** sauf que le caractère lu est sorti du *flux-d'entrée* et n'est donc plus disponible.

READ-CHAR-NO-HANG
(READ-CHAR-NO-HANG *flux-d'entrée eof-erreur eof-val rec-p*), SUBR de 0 à 4 arguments (page 111)
comme **READ-CHAR**, mais si aucun caractère n'est disponible, cette fonction retourne *immédiatement* **NIL** et n'attend pas l'arrivée d'un caractère

REDUCE **(REDUCE** *fn seq* **:FROM-END :START :END :INITIAL-VALUE)**, SUBR de 2 à 6 arguments (pages 82-83)
combine tous les éléments d'une séquence par un opérateur binaire *fn*. Cette réduction est associative à gauche, sauf si l'argument **:FROM-END** est **T** (par défault **NIL**), dans ce cas la réduction est associative à droite. Avec une valeur initiale **:INITIAL-VALUE**, cette valeur est placée avant la séquence (après si **:FROM-END** est **T**) et inclue dans l'opération de réduction. Les mots-clef **:START** et **:END** peuvent spécifier une sous-séquence de *seq*. Si c'est le cas, la réduction ne s'applique qu'à cette sous-séquence.

REM **(REM** n_1 n_2), SUBR à 2 arguments (page 41)
retourne le reste de la division entière de n_1 / n_2

REMPROP **(REMPROP** *pl ind*), SUBR à 2 arguments (page 56)
enlève de la P-liste de *pl* l'indicateur *ind* et la valeur associée. Retourne l'ancienne P-liste si l'indicateur *ind* existe, sinon **NIL**

REVERSE **(REVERSE** *seq*), SUBR à 1 argument (pages 28, 81)
retourne la séquence *seq* avec ses éléments dans l'ordre inverse

RPLACA **(RPLACA** *l s*), SUBR à 2 arguments (page 156)
remplace physiquement le CAR de *l* avec la valeur de *s*. **RPLACA** retourne la liste *l* modifiée

RPLACD **(RPLACD** *l s*), SUBR à 2 arguments (page 156)
remplace physiquement le CDR de *l* avec la valeur de *s*. **RPLACD** retourne la liste *l* modifiée

SET **(SET** *sym s*), SUBR à 2 arguments (page 167)
change la valeur de la valeur de sym en *s*

SET-MACRO-CHARACTER

(SET-MACRO-CHARACTER *c fn ntp*), SUBR à 2 ou 3 arguments (page 124)
définit le caractère *c* comme un macro-caractère avec la fonction associée *fn*. Si *ntp* est non-**NIL** (**NIL** par défaut), ce sera un macro-caractère non terminant. **SET-MACRO-CHARACTER** retourne **T**.

SET-SYNTAX-FROM-CHAR

(SET-SYNTAX-FROM-CHAR c_1 c_2), SUBR à 2 arguments (page 122)
donne au caractère c_1 la même syntaxe que possède le caractère c_2.

SETF **(SETF** pl_1 v_1 ... pl_N v_N), MACRO (pages 55, 168)
la fonction d'affectation généralisée : elle prend les lieux pl_1, pl_2, ..., pl_N et leurs affecte les nouvelles valeurs v_1, v_2, ..., v_N. Les lieux pl_i peuvent être toute variable ou tout lieu calculé par des appels des fonctions : CAR et CDR et leurs combinaisons, NTH, GET, SYMBOL-VALUE, SYMBOL-FUNCTION, SYMBOL-PLIST.

SETQ **(SETQ** sym_1 s_1 ... sym_N sN), MACRO (page 166)
sym_1, ..., sym_N sont des symboles (ces arguments ne sont pas évalués) s_1, ..., s_N sont des expressions. **SETQ** affecte à chaque sym_i la valeur correspondante de s_i. **SETQ** retourne la valeur de s_N.

SQRT **(SQRT** *n*), SUBR à 1 argument (page 204)
retourne la racine carrée de *n*

STRING **(STRING** *s*), SUBR à 1 argument (page 81)
retourne la chaîne de caractères résultante de la conversion de *s* en chaîne de caractères

STRINGP **(STRINGP** *s*), SUBR à 1 argument (page 69)
retourne **T** si *s* est une chaîne de caractères, **NIL** sinon

SYMBOL-PLIST

(SYMBOL-PLIST *sym*), SUBR à 1 argument (page 52)
retourne la P-liste associée au symbole *sym*

SYMBOL-VALUE

(SYMBOL-VALUE *sym*), SUBR à 1 argument (page 52)
retourne la valeur actuelle du symbole spécial (dynamique) *sym*

THROW **(THROW** *tag expression*), Special Form, (page 182-184)
THROW redonne le contrôle à une construction **CATCH** du même *tag*. Retourne la valeur de *expression*, qui sera également la valeur du **CATCH** correspondant

TERPRI **(TERPRI** *flux*), SUBR à 0 ou 1 argument (page 71)
effectue un retour chariot sur le flux *flux* ou sur la sortie standard si aucun flux n'est spécifié. **TERPRI** retourne **NIL**.

UNLESS **(UNLESS** *tst* s_1 ... s_N**)**, MACRO (page 60)
si l'évaluation de *tst* donne **NIL**, **UNLESS** évalue en séquence s_1, s_2, ... s_N et retourne la valeur de s_N. Sinon, si *tst* s'évalue à une valeur non-**NIL**, **UNLESS** retourne **NIL**

WHEN **(WHEN** *tst* s_1 ... s_N**)**, MACRO (page 61)
si l'évaluation de *tst* donne une valeur non-**NIL**, **WHEN** évalue en séquence s_1, s_2, ... s_N et retourne la valeur de s_N. Sinon, si *tst* s'évalue à **NIL**, **WHEN** retourne **NIL**

WRITE-CHAR
(WRITE-CHAR *c flux***)**, SUBR à 1 ou 2 arguments (page 111)
imprime le caractère *c* sur le flux *flux* ou sur la sortie standard si aucun flux n'a été spécifié. **WRITE-CHAR** retourne *c*.

ZEROP **(ZEROP** *n***)**, SUBR à 1 argument (page 42)
retourne **T** si *n* est égal à 0, **NIL** sinon

` le macro-caractère *backquote* (page 128)

| le caractère d'échappement multiple (il sert à délimiter les noms de symboles comportant des caractères spéciaux) (pages 79, 120, 122)

' le macro-caractère **QUOTE** (page 5)

24 INDEX

[(crochet ouvrant) 124, 126
' (quote-caractère) 5, 119, 123
. (point) 1, 68, 111, 120, 153
` (antiquote) 122, 127
- 41
\ (antislash) 119, 122
] (crochet fermant) 124, 127
^G (BELL) 112
{ 121
| (barre verticale) 120, 122
} 121
~ (tilde) 76
~% 78
~A 76
~D 77
~O 77
~S 77
~T 78
~X 77
~~ 78
" (guillemet) 64, 68, 119, 122
(dièse) 93, 120
#' 93
#\ 81
#\< 288
#\> 288
#\Newline 114, 122
#\Page 122
#\Return 114, 122
#\Space 114, 122
#\Tab 114, 122
#\` 122, 287
$ 141, 232
$- 215
% 78, 232
&KEY 97
&OPTIONAL 95
&REST 93
&WHOLE 210
( ) 3
((parenthèse ouvrante) 1, 111, 119, 122
) (parenthèse fermante) 1, 111, 119, 122
* 41
standard-input 136
standard-output 136
+ 41
, (virgule) 128
,@ 128
/ 41
1+ 40
1- 40
3NOMBRES
4-IEME 15, 191-192
4FOIS 266
: 97
:DIRECTION 133-
:INPUT 134
:IO 134
:OUTPUT 133-
:PROBE 134
; (point virgule) 111, 119, 122
< 43
<= 43
= 43
> 43
>= 43
? 141, 232
?- 215
@ (escargot) 128

A

A-liste 215
ABS 204
accès 81
accès aux éléments d'une liste 6-7
accolades 121
accumulateur 29
ACK 272
Ackermann 51
action-si-faux 24
action-si-vrai 24
adresse 149-

adresse mémoire 147
affectation 166-, 238
affectation généralisée 168
algorithme 249
ALL-CDR-REVERSE 135
ALLLIVRE 274
ALLTITRE 274
analyse 110
ANALYSE 225
ANCETRES 278
AND 74
antiquote 127
antislash 119
appel de fonction 5, 100
appel de fonction anonyme 47
appel de fonction implicite 200
appel de procédure 246
APPEND 31-32, 49, 83, 209
application de fonction 103
APPLY 103
arbre 10, 57
arbre d'appels 181
argument clef 97
argument de fonction 5
argument fonctionnel 201
argument obligatoire 96-97
argument optionnel 55, 95
argument rest 93
arithmétique 40-, 173-
Arsac, Jacques 208
assertion 236
ASSOC 219-220
ATOM 20, 69
ATOM-CAR? 22
atome 1
atome littéral 52
atome mono-caractère 125
atome variable 89
ATTACH 161
AUTEUR 274
AUX-SUM 137
AUXILIAIRE 48, 50
axiome 237
à la ligne 114

B

BACKQUOTE 129
backquote 129
backtracking 228
balayer une liste 233
BAR 39, 96, 160, 217
base de données 52, 56
BIDON 141
BIN-DEC 50
binaire 48
BIZARRE 26
blanc 121
BQ-' 129
branchement 200
buffer 70, 109
bug 235

C

C-MEMQ 164, 165
C-REVERSE 293
C-valeur 52
CADDDDAR 256
CADDDDR 256
CADDR 7
CADR 7
CALCUL 102, 109
calcul ascendant 32
calcul descendant 32
CALCULATEUR 109
CALCULATEUR-INTERNE 109
canal d'entrée/sortie 138
canevas 141
CAR 6, 149
CARACTERE_ACTUEL 185
caractère 110
caractère blanc 121
caractère constituant 121
caractère d'échappement 121
caractère illégal 121
caractère prompt 7
caractère spécial 114
CARRE 42
CARRE-DU-CARRE 42
Carroll, Lewis 100
CATCH 182-184
CBCOM 121
CDDR 192
CDOT 121
CDR 6, 149

CECOM 121
cell-value 52
chaînage 152
chaîne de caractères 64, 68, 119
chaîne de contrôle 76
CHAR< 202
CHAR> 202
chargement de fichiers 132-
CHERCHE 274
CHERCHEALL 274
CHOIX 229, 291
Church, Alonzo 172
circonflexe 122
clause de COND 36
clavier 107
clef 97
CLOSE 134
CLPAR 121
CM-INTERNE 164
CMACRO 121
CMSYMB 121
CNULL 121
COLLE 268
COMBINE-EXPRS 130
commentaire 119
comparaison (cf filtrage)
compilateur 217
COMPTE 113
COMPTE-INTERNE 113
compter 112
concaténation 32
CONCATENATE 81-82
COND 36
conditionnel 240
CONJ1 84, 282
conjugaison 84
CONJUGUE 84, 282
CONS 7, 152
CONS? 25
conséquence 239, 241
conser 31
CONSP 21
constante 69, 142
constituant 121
construction de listes 25
contrainte 226
corps de fonction 16, 45, 47
corps du LABELS 47
couple situation/action 224
Cousot, Patrick 244
CPKGC 121
CPNAME 121
CQUOTE 121
création d'un fichier 135
CREE-MOT 84
CROCHET-FERMANT 127
CROCHET-OUVRANT 126
CRPAR 121
CSEP 121
CSPLICE 121
CSTRING 121

D

data-driven 251
DE 16, 88
débordement 249
déclaration 188, 216
définir une fonction 15
définir une variable 189
définition de macro-caractères 116-, 124-
définition de macro-fonctions 191-
définition de procédure 246
délimiteur de chaîne 119, 122
délimiteur de commentaire 111, 119, 122
DEC-BIN 48, 49
DEC-HEX 270
DEC-OCT 270
DEC-OCT-HEX 271
DECLARE 217
DECR 294
DEFMACRO 191
DEFUN 15
DEFVAR 189
DEL1 293
DELETE 37, 81, 268
DELINEAR-DONNEE 231
DF 91
dièse 93, 120
différence 41
Dijkstra, Edsgar 236
DIRECTION 133-
DISPLACE 171
DIVISE_LA_BANDE_EN_DEUX 189
division 41
division entière 41

division entière, reste 41
DMC 123
DO 211
DOCUMENTATION 189
documentation 57
donnée abstraite 240
DOUBLE 268
doublet 149

E

échange de deux variables 166, 237
échappement 180
échappement multiple 121
échappement simple 121
écran 63, 107
édition 71-72
égalité 34, 43, 161-
égalité d'adresses 161
égalité de listes 161
égalité de nombres 43
état 172
évaluation 17, 46, 88, 91, 100-
évaluation d'une forme 5
effet de bord 63
Eliza 224
ELIZA 225
ELT 82
ENL-A 206
ENL-A-AUX 206
ENL-A-SANS-REPETITIONS 206
enregistrement 110
entrée standard 136
entrée/sortie 63-, 107-, 133
environnement 215, 235
EQ 34
EQUAL 34
escargot 128
espace 128
espace mémoire 147
et logigue 74
Euclide 44
EVAL 101-102
EVENP 43
EXECUTE 186, 190
EXEMPLE 97
EXEMPLE1 98, 192
EXEMPLE2 98
EXIT 182
EXPL 80
EXPLODECH 78, 80
EXPR 91
expression algébrique 62
expression arithmétique 102, 109
expression LISP 80
expression symbolique 154

F

F 39
F1 74, 181
factorielle 44
FACTORIELLE 44, 278
falsification 236
faux 20
FDELQ 293
fermer un fichier 134
FEXPR 92, 105
FIB 59-60, 271
Fibonacci 51
FIBONACCI 58, 271
fichier d'entrée 107, 132-
fichier de sortie 107, 132-
filtrage 141-, 214-
FIND 255
FINDALL 276
FINDALL-TROIS 277
FIND_INSTR 185, 190
FIND_INSTRUCTION 185, 190
FLOOR 42, 49
flux 125, 133
flux d'entree 110
flux de contrôle 180
flux de sortie 110
fonction 5
fonction anonyme 45, 205
fonction arithmétique 40-
fonction d'accès 168
fonction d'entrée/sortie immédiate 112
fonction linéaire 20
fonction locale 46
fonction numérique 40-
fonction récursive 30
fonction récursive terminale 33
fonction standard 15
fonction utilisateur 15

FOO 39, 42, 88, 96, 156, 193, 217
FOOBAR 39
FORALL 274
forcer une évaluation 101
FORMAT 75-76
FORMAT-AUX 73
FORMATAGE 73
formatage 73
forme 5, 102, 192
Fortran 76
FRERES 276
FREV-INTERNE 160
FREVERSE 160, 293
funarg 197
FUNCALL 203

G

garbage collection 148
GCV 252, 262
générateur de condition de vérification 249
GENERE 296
GENSYM 196
GET 55
GET-MACRO-CHARACTER 128
Goossens, Daniel 200, 250
GR 269
Greussay, Patrick 165
guillemet 64

H

HAN 283
HANOI 67, 283
HANOI-AUX 283
HEX-DEC 271
HEX-DEC1 271
hexadécimal 77
Hoare, C.A.R. 236

I

IBM-704 6
identité 161
identité d'atomes 161
identité de listes 162
IF 24
IMPAIR 289
implémentation 150
IMPLODECH 80, 83
impression 63-
IMPRIME-TOUS 64
INCHAN 138
INCR 294
incrémentation 294
indéfinie 53
INDEFINIE 53
index 82
indicateur 54, 96
indirection 167
inférieur 43
infixe 102
INPUT 134
INSERE 201
INSERE-1 202
INTERN-C-MEMQ 164
INTERNAL-C-REV 293
INTERNAL-IMPAIR 289
INTERNAL-PAIR 289
INTERNAL-SOMME 289
interprétation 217
INTERPRETE 233
interprète 107, 114, 172
interprète de règles 232
interruption 41
invariante 243
IO 134
ISCONST 130
itérateur 209-
itération 243

K

KEY 97
keyword argument 97
Knuth, Donald 44
Kraus, Karl 57

L

LABELS 46-47
LAMBDA 205
lambda-expression 205
lambda-liste 194
lambda-liste déstructurante 194
LAST 160
lecture immédiate 111-112

lecture LISP 107-
lecture physique 110
lecture récursive 125
LENGTH 81, 272
LENGTH-INTERNE 272
Leonardo de Pise 51
LET 45, 206
LETALL 223, 232
LETN 47
LEXPR 91
liaison 30, 52, 88-
liaison de variables 17
liaison dynamique 166, 217
liaison lexicale 217
liaison superficielle 215
lier 17
linéariser 228
LINEAR 231
linefeed 122
LISP 108, 115
LISP impure 168
LISP lexical 217
LISP pur 168
LISP-INTERNE 115
LIST 90
LIST-EACH 204
liste 1
liste d'arguments 104
liste d'associations 215
liste de paramètres 217
liste de propriété 52
liste de variables 16, 89, 95-98, 193
liste libre 152
liste vide 3, 8, 20
liste-arbre 10
LISTE-CAR? 266
LISTE3 17
listes circulaires 163, 200
listes partagées 162
LIT 209
LOAD 133
LOADFILE 138
logique de Hoare 235
longueur 2, 81
longueur d'une liste 2
Luckham, David 250

M

machine de Turing 172
machine finie 188
macro caractère 116, 124
macro caractère non-terminant 121
macro caractère terminant 121
macro expansion 198
macro fonction 191
macro fonction destructive 195
MACROEXPAND 198
MAPC 206, 296
MAPC-AUX 296
MAPCAR 204, 209, 296
MAPCAR-AUX 296
MAPS 294
MATCH 229
MATCH-AUX 229
MAX 284
MAXIMUM 284
McCarthy, John 154
MCONS 193
mémo-fonction 58-
mémoire 147-149
mecanismes de liaison 88-
MEMBI 261
MEMBER 269
MEMQ 268
micro-manuel 115
MIN 285
MINIMUM 285
Minsky, Marvin 173
mnémonique 16
modification de listes 155-
modification destructive 155-
modification physique 156
mono-caractère 125
mot-clef 93
MUL 209
MULT-PLUS 94
multiplication
MY-QUOTE 125

N

NBATOM 272
NBATOM-INTERNE 272
NCONC 160

NCONC1 201
NEQUAL 124
newline 78, 122
NEXPR 91
NEXTL 171, 196, 293
NEXTL-DONNEE 232
NEXTL-PRESQUE-CORRECT 196
NIL 3, 8, 20, 150
nom d'échappement 183
nom d'un caractère 114
nom de fichier 133
nom de fonction 16
nombre 1, 48
nombre d'éléments, cf. longueur
nombre quelconque d'arguments 89, 93
notation infixée 102
notation polonaise 102
notation prefixée 102
NTH 270
NTHCDR
NULL 22
NUMBERP 21
NUMBERP-CADR 23, 266
NUMBERP-CADR? 25

O

OBLIST 115
obliste 115
occurrence d'un élément 37
occurrence libre 217
OCT-DEC 271
OCT-DEC1 271
octal 77
ODDP 43
opérations destructive 160
OPEN 133-
OPENI 138
OPENO 138
OPTIONAL 95
OR 74
ou logique 74
Oulipo 74
OUTCHAN 138
OUTPUT 133-
ouvrir un fichier 133-

P

P 105
P-liste 52-
P-name 53
P-P-P 257
page 122
PAIR 288
paire pointée 155
PAL 223-224
PALINDROME 222
palindrome 222
paramètre 17, 93, 246
parenthèses 1, 111
partager 162
pattern 142
pattern matching, cf. filtrage
périphérique 63, 109
périphérique standard 63
PEEK-CHAR 111
PEEKCH 111
Perec, Georges 222
PERMUT 207
PERMUT1 207
permutation 207
PERSONNES 276
PGCD 44-45
PGCD1 45
pile 147
PLENGTH 73, 79, 83
PLIST 52
PLUS 89-90, 93, 105
PLUS-UN 204
point 1, 68, 153
point-virgule 8
pointeur 70, 110, 149
postcondition 249
PPLUS 89, 105
PR 257
PR-COMPOSE 282
précondition 249
prédicat 20
prédicats numériques 42-43
préfixe 102
PREFIX 102
PREV-INTERNE 171
PREVERSE 167, 171
PRIN 70

PRIN1 71
PRINC 72
PRINCH 72
PRINT 63, 135
PRINT-FIRST-ELEMENT 73
print-name 53
PROBE 134
PROBE 134
PROBEFILE 138
problème de 8 reines 207
procédure 245-
profondeur 4
profondeur d'une liste 4
PROGN 169
programmation de jeux 112
programmation par règles 224
programme 237
programme linéaire 20
propriété 51-
pseudo-langage 237
PUT-SEXE 275
PUTPROP 54

Q

QLIST 91
QSLIST 284
Queneau, Raymond 35
quintuplet 172
QUO 41
QUOI 51
QUOTE 5, 101, 116
quote-caractère 5
quoter 5, 101

R

racine 46
RACINE 84
RACINE-ENTIER 46, 47, 286
ramener une valeur 5
RANDOM 204
récupération de mémoire 148
récurrence 44-45
récursion 30
récursivité 30
récursivité terminale 33
répétition 28-
READ 107, 125, 136
READ-ATOM 115
READ-CHAR 110
READ-CHAR-NO-HANG 111
READ-LIST 114
READCH 110
récursive-p 136
rédiriger 136
REDUCE 82
REFLECHIT 225
REINE 208, 209
REINE-AUX 209
REM 41
remplacer 156
REMPROP 56
repetition 28-
REPOND 225
représentation de données 177
REST 93
REST 93
retour charriot 78
return 111, 113, 119
REV 267
REVERSE 28, 81, 94-95, 210
REVERSE3 266
règle de reécriture 233, 249
Rogerien 224
RPLACA 156
RPLACB 294
RPLACD 156

S

S-expression 154
saut de ligne 78
schéma 204
segment 221
sélection généralisée 36
sélection 20
sémantique 236
sémantique axiomatique 236
séparateur 1, 68
séquence 81, 143
séquence d'arguments 76
SEQUENCES 222, 227, 229
SET 166-
SET-MACRO-CHARACTER 124
SET-SYNTAX-FROM-CHAR 122

SETF 55, 168
SETNTH 298
SETQ 166-, 294
SIBLINGS 277
SIMP 279
SIMPL 234
simplificateur algébrique 62, 232
SIMPLIFY-PLUS 279
SIMPLIFY-TIMES 279
simulateur 177
SKE 165
SLIST 284
SMASH 161
SOEURS 277
SOMME 289
SOMME-DE-NOMBRES 137
SOMME-DIFFER 42
sortie 63-
sortie standard 63, 136
sous-liste 3
spécification 31, 235
SPECIAL
SQRT 204
SSLIST 284
STAT 115, 116
STAT-INTERNE 115
STAT-READ 114, 117
Steele, Guy Lewis 98
STRING 81
STRING-APPEND 83
STRINGP 69
structure 141
SUBS 256
SUBST 253, 256
suite-de-caractères 69
supérieur 43
SYMBOL-PLIST 52
SYMBOL-VALUE 52
symbole 52, 196
SYMEVAL 52
système d'exploitation 133

T

T 20
tableau 80
tableau de correspondance 54
tabulation 78, 114, 119, 122
TAG 182
tampon 70, 109
tautologie 240
terminal 107, 63
TERPRI 70, 72
test 20, 25
test d'arrêt 29, 37
théorème 235
THROW 182-184
tilde 76
TIMES 284
toplevel 63, 233
tour de Hanoi 65, 87
trace d'exécution 30
TRAD 271
TRADUIRE 287
transformation 240
transfert 244
TRIE-UNE-LISTE 201, 202
TRIE-UNE-LISTE-EN-ORDRE-DECROISSANT 202
tronquer 49
TRUC 18
TRUC1 19
TTIMES 284
TURING 179, 184, 189
Turing, Alain 172
turingule 172
TURING_AUX 189
TWICE 203
TYI 112
TYO 112
TYPE 84 type abstrait 81
type d'objet 80
type de caractère 118-122
type de donnée 133
type de résultat 82
TYPE? 26
TYPECH 121

U

uni-dimensionnel 81
unité centrale 172
UNLESS 60
UNTIL 210, 295