LISP

UNE INTRODUCTION A LA PROGRAMMATION

Département Informatique
Université Paris VIII Vincennes à Saint-Denis

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 la 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, Common-LISP, VLISP, LE_LISP, etc, en témoigne. De plus, signalons l'apparition 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 l'ordinateur Maia, actuellement en développement à la CGE.

En LISP, de même que dans les langages-machine, la représentation des données et la représentation des programmes sont identiques : ainsi un programme LISP peut construire d'autres programmes LISP ou se modifier au fur et à mesure de son exécution. Surtout, 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 sont 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. Les dialectes utilisés

dans ce livre sont d'une part VLISP, le dialecte LISP développé à l'Université Paris VIII (Vincennes) par Patrick 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 dans des notes en bas de page.

Chaque chapitre 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 chapitres 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 la différence entre les fonctions de type EXPR et de type FEXPR. Nous y examinons 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 une petite fonction de filtrage.
• Les chapitres 15 et 16 examinent en détail la structure des listes. Nous y verrons également les fonctions d'affectation et un troisième type de fonction : les macro-fonctions.
• Le chapitre 17 présente diverses formes de répétitions itératives.
• Au chapitre 18 nous reprenons l'exemple de la fonction de filtrage. Nous y construisons une fonction de filtrage très puissante, intégrant toutes les notions abordées précédemment.
• Finalement, le chapitre 19 donne les solutions à tous les exercices que vous avez rencontrés dans ce livre.

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, par leurs questions et par leur enthousiasme, ont largement influencé l'écriture de ce livre. C'est à eux tous que ce livre est dédié.

Ce livre a été édité par l'auteur à l'IRCAM sur ordinateur DEC VAX-780 sous système UNIX, graˆce 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 et remarques d'un grand nombre de personnes. Qu'il me soit permis ici de remercier particulièrement Jean-Pierre Briot, Annette Cattenat, Frédéric Chauveau, Pierre Cointe, Gérard Dahan, Jacques Ferber, Patrick Greussay, Daniel Goossens, Eric Halle, Christian Jullien, Jean Mehat, Gérard Nowak, Gérard Paul, Yves-Daniel Pérolat, Jean-François Perrot, Christian Riesner, Nicole Rocland, Bernard-Paul Serpette, Jacqueline Signorini, Patrick Sinz, Ellen Ann Sparer, Roger Tanguy.

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 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 :

liste ::= **(** {qqc$_1$ qqc$_2$ ... 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 en avoir ou pas dépendant du cas où vous vous trouvez. 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 peut donc parler 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 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 :

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

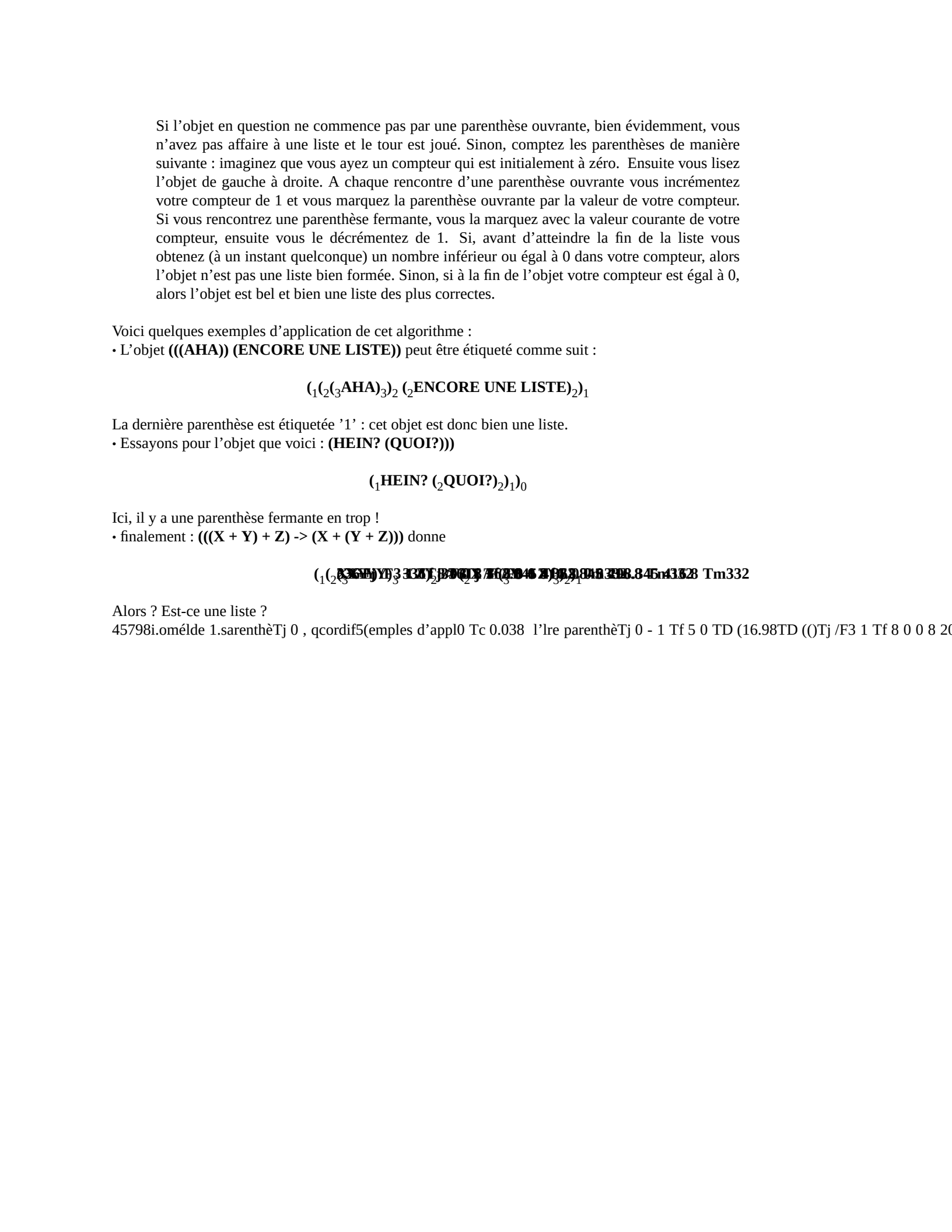
La dernière parenthèse est étiquetée '1' : cet objet est donc bien une liste.

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

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

Ici, il y a une parenthèse fermante en trop !

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



Alors ? Est-ce une liste ?

45798i.omélde 1.sarenthèTj 0 , qcordif5(emples d'appl0 Tc 0.038 l'lre parenthèTj 0 - 1 Tf 5 0 TD (16.98TD (()Tj /F3 1 Tf 8 0 0 8 20

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

3. Pour chacune des listes ci-dessus, donnez 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 à l'appel.

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

(QUOTE arg_1**)** $\rightarrow$ 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 " $\rightarrow$ ".

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

(QUOTE TIENS)	$\rightarrow$	**TIENS**
(QUOTE (A B C))	$\rightarrow$	**(A B C)**
(QUOTE (THE (PUMPKIN (EATER))))	$\rightarrow$	**(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	$\rightarrow$	**TIENS**
'(A B C)	$\rightarrow$	**(A B C)**
'(THE (PUMPKIN (EATER)))	$\rightarrow$	**(THE (PUMPKIN (EATER)))**

Cette notation n'est qu'une abréviation : la machine, elle, comprend toujours la même chose, i.e. 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))**

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 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

[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 **C**ontenue de la partie **A**dresse d'un **R**egistre livrait alors le **CAR** d'une liste et le **C**ontenue de la partie **D**écrement d'un **R**egistre livrait le **CDR** d'une liste. En honneur de cette première implémentation de LISP, toute les versions suivantes ont gardé ces deux noms.

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 une image d’une petite interaction avec la machine (La machine écrit un "**?**" quand elle attend que vous entrez quelque chose. Elle imprime la valeur de ce que vous avez demandé, précédée du signe "=", sur la ligne suivante.) :

```
?   ’A
=   A

?   ’(A B C)
=   (A B C)

?   ’’(A B C)
=   ’(A B C)

?   (CAR ’(A B C))
=   A

?   (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

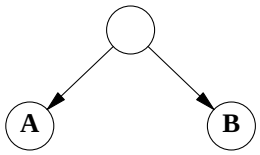
1. Donnez les résultats des appels des fonctions suivantes :

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)))) → ?**

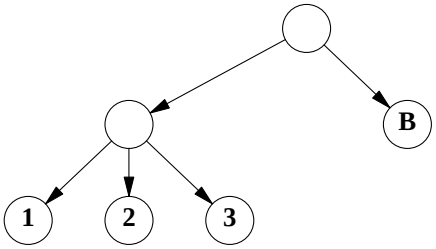
3. UN AUTRE POINT DE VUE SUR LES LISTES

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 :

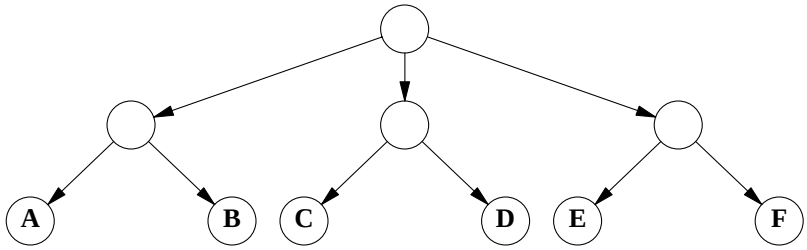


La liste **((1 2 3) B)** peut être représentée comme :

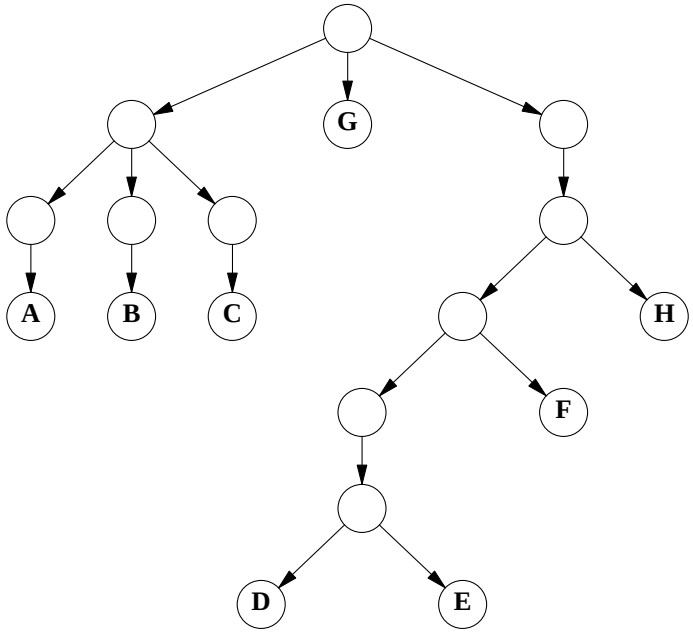


Cette représentation permet de reconnaître visuellement à la fois les éléments et leur profondeur. Une liste (ou sous-liste) est représentée comme un cercle vide. Les éléments d'une liste sont les cercles directement connectés au cercle de la liste, et la profondeur d'un élément se compte simplement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

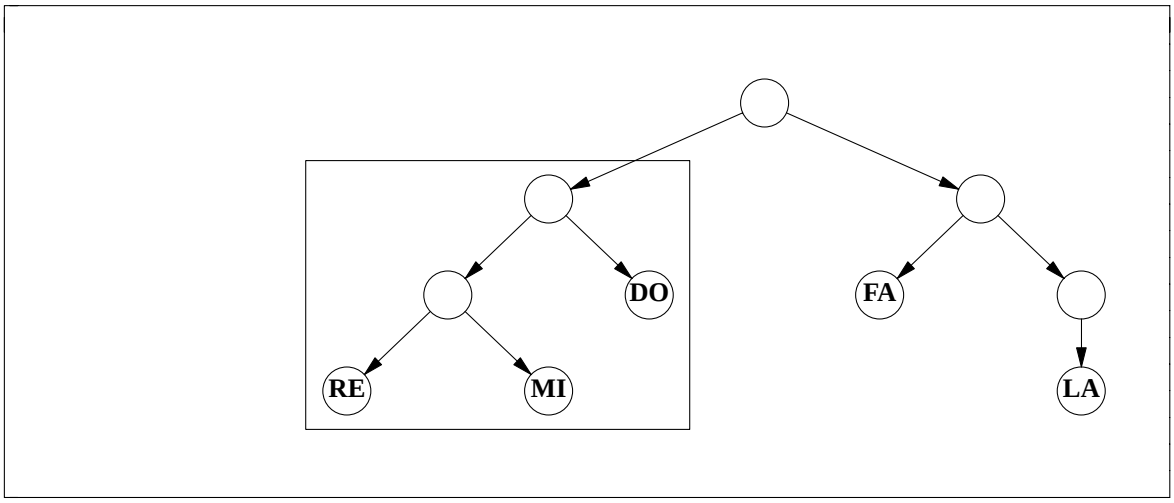
La liste **((A B) (C D) (E F))** peut être représentée comme :



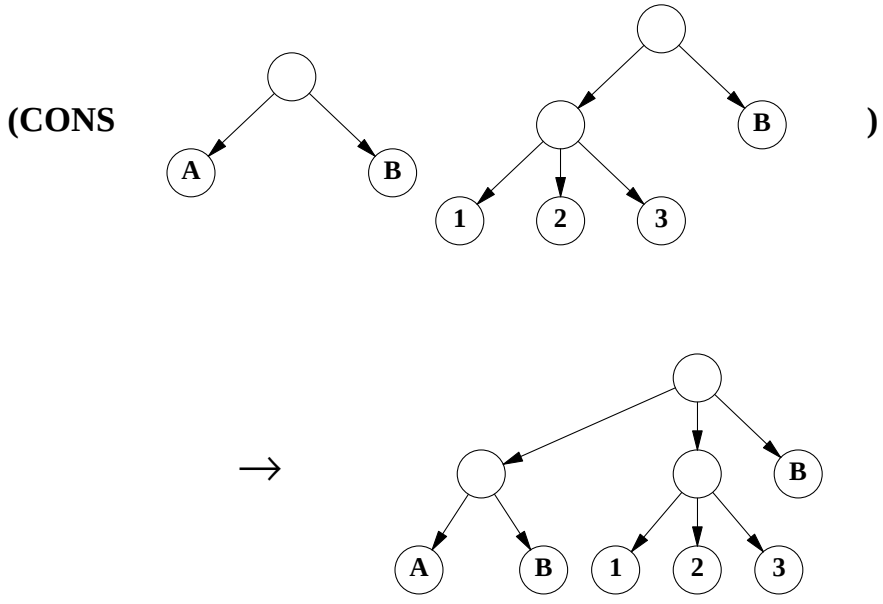
La liste **(((A)(B)(C)) G (((((D E)) F) H)))** peut être représentée comme :



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** necessaires pour remplacer le signe "?" dans les appels de fonctions suivants

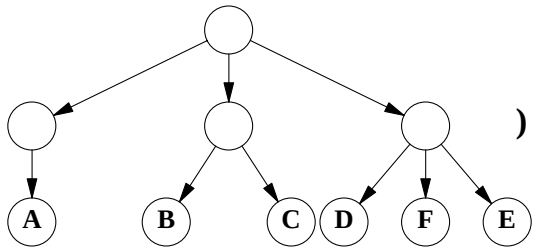
(? '(A B C D))	→	**D**
(? '((A (B C)) E))	→	**C**
(? '(((DIEU) ENCORE) UNE))	→	**DIEU**
(? '(((DIEU) ENCORE) UNE))	→	**ENCORE**

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) ()))	→	?

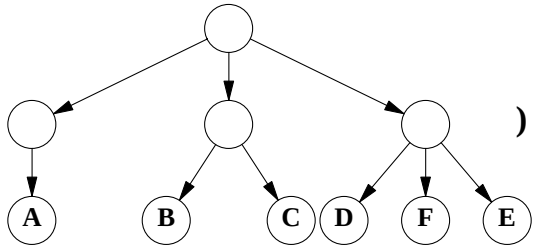
4. Donnez l'arbre résultant des opérations suivantes :

(CAR



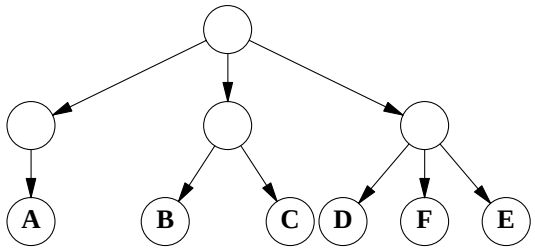
) → ?

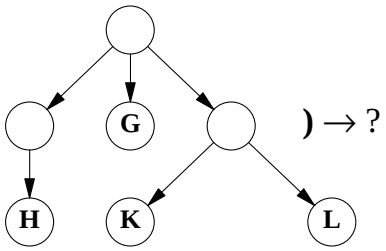
(CDR



) → ?

(CONS





) → ?

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

4. LA DEFINITION DE FONCTIONS

Nous avons vu qu'il est possible de construire une liste, graˆce à la fonction **CONS**, de prendre le premier élément d'une liste, graˆce à la fonction **CAR**, de récupérer la liste sans son premier élément, graˆce à la fonction **CDR**, et, par des combinaisons de ces fonctions, d'accéder à n'importe quel élément d'une liste ou 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 :

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

DE est une fonction standard indiquant à la machine qu'on est en train de DEfinir 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 à la fonction un *argument*. 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 **DE** :

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

avec

nom-de-la-fonction

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.

Le premier argument de la fonction **DE** 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 **DE** ?!) 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 **DE**) 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.

La fonction une fois *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))

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 à la valeur des arguments de l'appel*. L'argument à l'appel de l'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 !). Toutefois, il est important de distinguer la *définition* d'une fonction de 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)) $\rightarrow$ **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 :

```
(DE 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 qui lui 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 :

```
(DE TRUC (QQC)
    (CONS QQC (CONS QQC (CONS QQC ()))))
```

voici un appel :

(TRUC 'HMMM) → ?

Examinons le 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 des fonctions standard et ensuite seulement nous pouvons calculer la valeur de la fonction. Détaillons :

1. La première chose à faire est d'évaluer les deux arguments du premier **CONS** : l'expression **'HMMM**, le premier argument, et **(CONS 'HMMM (CONS 'HMMM ()))**, le deuxième argument. 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 :

```
(DE 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 à zéro 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 petits 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 auquelles 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 **T**rue, 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 LE_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 nombres. 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**[1] teste si son argument est une *liste*. Voici leurs définitions syntaxiques et quelques exemples d'appels :

(CONSP *arg***)** → **NIL** si *arg* n'est pas une liste
→ **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** :

(NUMBERP *arg***)** → **NIL** si *arg* n'est pas un nombre
→ *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. Tout 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. 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** :

[1] En VLISP le prédicat **CONSP** s'appelle **LISTP** et le prédicat **NUMBERP** s'appelle **NUMBP**.

(NUMBERP 0)	→	**0**
(NUMBERP -110)	→	**-110**
(NUMBERP NIL)	→	**()**
(NUMBERP 'HMMM)	→	**()**
(NUMBERP 3214)	→	**3214**
(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 ?" ! C'est une fonction extrèmement utile, comme nous le verrons dans le chapitre suivant.

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 :

```
(DE 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 :

```
(DE 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 (ou de manuels quelconques) : 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 :

(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)) $\rightarrow$ **(UN NOMBRE)**

L'évaluation du test **(NUMBERP 1)** ramène en valeur le nombre **1**, qui est 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)) $\rightarrow$ **(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 :

(IF (NULL '(A B)) (CAR '(DO RE))(CDR '(DO RE))) $\rightarrow$ **(RE)**
(IF (CONSP (CONS NIL NIL)) (CONS NIL NIL) 'BRRR) $\rightarrow$ **(NIL)**
(IF NIL 1 2) $\rightarrow$ **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) $\rightarrow$ **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 enfin commencer à écrire de vraies (petites) 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 dans les autres cas, elle ramène la liste **(**deuxième-élément **N EST PAS UN NOMBRE)** :

```
(DE 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)) $\rightarrow$ **(2 EST UN NOMBRE)**
(NUMBERP-CADR? '(DO RE MI)) $\rightarrow$ **(RE N EST PAS UN NOMBRE)**
(NUMBERP-CADR? NIL) $\rightarrow$ **(NIL N EST PAS UN NOMBRE)**

encore quelques exemples :

```
(DE 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 :

```
(DE 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. quelques 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 :

```
(DE 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. Regardons d'abord un exemple d'une fonction qui inverse les éléments d'une liste quelconque :

```
(DE 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** :

	L	**RES**
état initial	**(A B C D)**	**( )**
opération	↓	↑
première transaction	**(B C D)**	**(A)**
opération	↓	↑
deuxième transaction	**(C D)**	**(B A)**
opération	↓	↑
troisième transaction	**(D)**	**(C B A)**
opération	↓	↑
dernière transaction	**()**	**(D C B A)**

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 regarder 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**. D'ailleurs, 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))

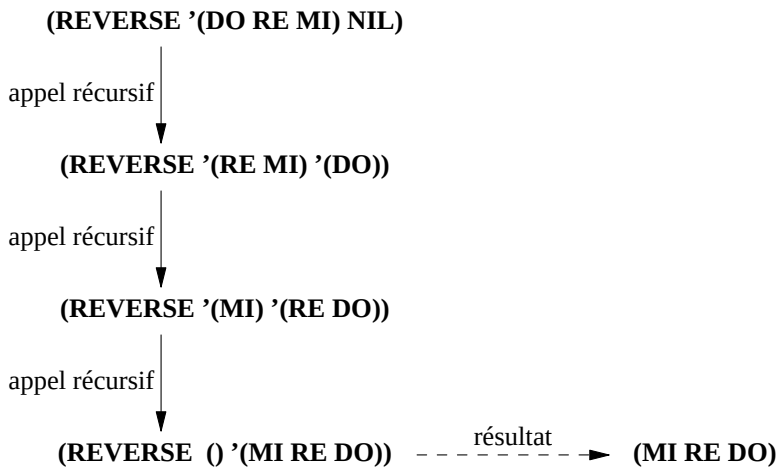
   Pour connaître le résultat de l'appel initial, il suffit donc de connaitre le résultat de ce nouvel appel.
2. On entre alors de nouveau dans la fonction **REVERSE**, liant la variable **L** à la nouvelle valeur **(RE MI)** et la variable **RES** à **(DO)**. Comme **L** n'est toujours pas égale à **NIL**, 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)
```

NOTE

Notons, qu'en VLISP il est possible d'omettre, à l'appel, les derniers arguments si l'on veut qu'ils soient liés à la valeur **NIL**. Les appels exemples de la fonction **REVERSE** peuvent donc également s'écrire de la manière suivante :

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

En LE_LISP, par contre, tous les arguments doivent être *impérativement* fournis, même quand ceux-ci sont égaux à **()**.

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 :

(DE 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 :

```
(DE APPEND (LISTE1 LISTE2)
      (REVERSE (REVERSE LISTE1) 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 prendrait si la liste était très, très longue.

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 differents éléments rencontrés, ensuite *consons*[2] chaque élément, en commençant par le dernier, à la liste **LISTE2**. Ce qui nous donne cette deuxième définition :

```
(DE 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

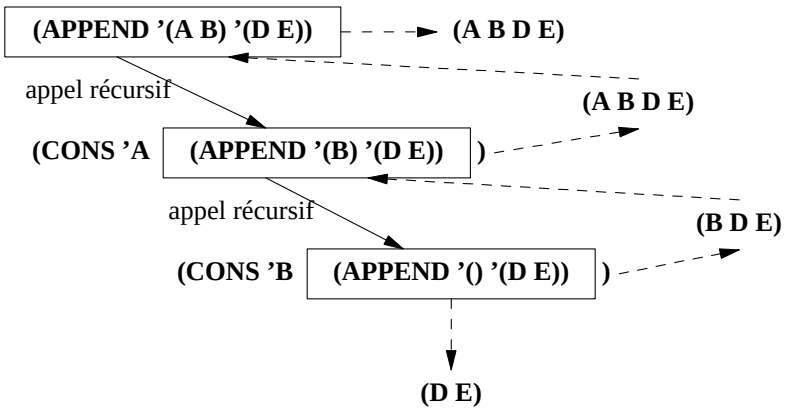
(APPEND '(A B) '(D E))

peut être représentée de la manière suivante :

---

[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´.

[2] voir la note précédente.



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** :

```
? (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, donc: ;
                                                ; le 1-er 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 de l'appel initial ;
```

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 tard 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**.

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

6.3. LA FONCTION EQUAL

Notre troisième exemple de 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 une fonction 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 :

```
(DE 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))))
```

Visiblement, 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 premières 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 :

---

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

[4] L'exception à cette règle est discutée dans le chapitre 15.

(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-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 :

```
(DE 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

[5] 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.

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 une fonction avec un ensemble de **IF**s imbriqués, comme ici, où vous aviez trois **IF**s 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'*act*ions à évaluer on sort du **COND** en ramenant en valeur la valeur de l'évaluation de *test*)

[2] si aucun des *test*s 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) :

```
? (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 successifs de deux listes. 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 :

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

avec la fonction **FOOBAR** que voici :

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

8. Que fait la fonction suivante :

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

9. Et que fait la fonction suivante :

```
(DE 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. Ici nous nous limiterons au calcul avec des nombres entiers compris entre -2^{15} et $2^{15} - 1$.[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]

(+ *n1 n2***)**	$\rightarrow$	$n1 + n2$
(- *n1 n2***)**	$\rightarrow$	$n1 - n2$
(* *n1 n2***)**	$\rightarrow$	$n1 * n2$
(/ *n1 n2***)**	$\rightarrow$	$\lfloor n1 / n2 \rfloor$

Toutes ces opérations arithmétiques sont génériques, c.à.d. les arguments peuvent être d'un type numérique quelconque : si toutes 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 / : En VLISP cette division est une *division entière*, ce qui veut dire que le résultat d'une division de *n1* par *n2* est

[1] 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 16 bits.

[2] Dans LE_LISP et dans quelques versions de VLISP (par exemple en VLISP-10), les fonctions +, - et * admettent un nombre quelconque d'arguments. Avant de commencer à écrire de grands programmes sur une machine, testez donc d'abord ces quelques fonctions arithmétiques.

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, dans le cas de deux arguments entier, LE_LISP offre la fonction **QUO**.

Dans ce livre d'introduction, nous nous limitons au calcul avec des entiers.

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

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

Voici quelques exemples numériques :

(1+ 1024)	$\rightarrow$	**1025**
(1- 0)	$\rightarrow$	**-1**
(1- -345)	$\rightarrow$	**-346**
(+ 17 9)	$\rightarrow$	**26**
(+ 1 -1)	$\rightarrow$	**0**
(- 10 24)	$\rightarrow$	**-14**
(* 6 4)	$\rightarrow$	**24**
(/ 1024 2)	$\rightarrow$	**512**
(/ 17 7)	$\rightarrow$	**2**
(REM 1024 2)	$\rightarrow$	**0**
(REM 17 7)	$\rightarrow$	**3**
(+ (* 3 3)(* 4 4))	$\rightarrow$	**25**
(- 111 (* 4 (/ 111 4)))	$\rightarrow$	**3**
(/ (+ 10 90)(- (* 5 6) 10))	$\rightarrow$	**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 :

```
(DE 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 :

```
(DE CARRE-DU-CARRE (N) (CARRE (CARRE N)))
```

ou, plus simplement :

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

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

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

que fait donc la fonction suivante ?

```
(DE 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 connaitre quelques prédicats numériques. Voici les plus importants :

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

(ZEROP *n***)**
- → **0** si $n = 0$
- → **NIL** si $n \neq 0$

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

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

le prédicat **>** se lit *greater* (en français : strictement supérieur)

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

le prédicat **<** se lit *less* ou *smaller* (en français : strictement inférieur)

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

le prédicat **>=** se dit *greater or equal*

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

et le prédicat **<=** se prononce *less or equal*[3]

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

Naturellement, le prédicat **EQ** fonctionne aussi bien sur des nombres entiers que sur des atomes

[3] Dans quelques versions de VLISP, la fonction >= s'appelle **GE** et la fonction <= s'appelle **LE**.

quelconques.

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

(EVENP *n***)**
→ *n1* si *n* est pair
→ **NIL** si *n* est impair

(ODDP *n***)**
→ *n1* si *n* est impair
→ **NIL** si *n* est pair

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

```
(DE EVENP (N) (IF (ZEROP (REM N 2)) N NIL))

(DE ODDP (N) (IF (= (REM N 2) 1) N NIL))
```

7.3. LA PROGRAMMATION NUMERIQUE

Pour nous habituer un tant soit 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 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 :

```
(DE 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 croit très vite.[4] Pensez-y quand vous la testez, sachant que si vous travaillez sur une machine à 16 bits, LISP ne connait pas les nombres entiers supérieurs à $2^{15} - 1$!

Dans son temps, Euclide avait developpé un algorithme pour trouver le *plus grand commun diviseur*, abrégé PGCD.[5] Son algorithme est le suivant :

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 $\lfloor$(n / m)$\rfloor$, m)

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

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

(DE 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 la valeur de l'appel **(PGCD1 M N)** quelque part, afin d'y avoir accès dans la suite. Il aurait été possible d'écrire la fonction **PGCD** comme suit :

---

[4] 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 trivial ? C'est

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

[5] 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.

```
(DE 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 intérmédiaire 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 - i.e.: le sens - de la fonction **LET** est la suivante :

> 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 18369)
= 9
```

La fonction **LET** est - nous l'avons dit - à la fois la définition et l'appel d'une fonction. Naturellement,

comme toute fonction, une fonction définie avec **LET** peut être appelée récursivement. Pour cela, nous avons besoin d'une fonction particulière puisqu'aucun nom pour nommer la fonction est disponible (rappelons que les fonctions construites avec **LET** n'ont *pas* de noms). En VLISP, la fonction **SELF** sert à appeler (sans préciser de nom) la fonction *à l'intérieur de laquelle elle se trouve*. Afin de voir comment ça marche, regardons l'exemple d'une fonction trouvant 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 '≤' veut dire *est inférieur ou égal à*). Voici la définition LISP de la fonction **RACINE-ENTIER**

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

La fonction **SELF** renvoie à la fonction construite à l'aide de **LET**. Tout se passe comme si la fonction **LET** définissait une fonction appelée **SELF** *suivie* immédiatement de son appel avec les arguments donnés dans le **LET**. On pourrait donc s'imaginer que **LET** (dans cet exemple précis) définit d'abord une fonction :

```
(DE  SELF (P K N)
     (IF (< N 0) K
         (SELF (+ P 2)(+ K 1)(- N (+ 2 P)))))
```

et modifie la fonction **RACINE -ENTIER** en ceci :

```
(DE RACINE-ENTIER (N)
     (SELF 1 0 (1- N)))
```

CE N'EST PAS CE QUI SE PASSE EN REALITE, mais c'est un bon modèle pour comprendre comment ça fonctionne !

Note important concernant LE_LISP :

Pour éviter quelques problèmes inhérants à la fonction **SELF**, LE_LISP a remplacé cette fonction par une deuxième construction du style **LET**. C'est la fonction **LETN**, pour **LET N**ommé. 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 SELF ((P 1)(K 0)(N (1- N)))
        (IF (< N 0) K
            (SELF (+ P 2)(+ K 1)(- N (+ 2 P))))))
```

Dans la suite de ce livre, si vous programmez en LE_LISP, il faut alors remplacer toutes les occurrences de

(LET . . . (SELF . . .) . . .)

par

(LETN SELF . . . (SELF . . .) . . .)

Remplacez donc toute expression **LET** qui contient un appel de la fonction **SELF** à l'intérieur, par une expression **LETN** avec le premier argument **SELF** en laissant le reste de l'expression tel quel.

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 **DEC-BIN** qui traduit un nombre décimal en une liste de zéro et de un (donc en une liste représentant un nombre binaire) :

```
(DE DEC-BIN (N)
    (IF (> N 0)
        (APPEND (DEC-BIN (/ 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** :

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

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.

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

quelques exemples d'utilisation :[6]

```
? (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 **CNTH**, 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 :

---

[6] Rappelons que, d'après notre algorithme de traduction, le programme **BIN-DEC** s'écrit en LE_LISP comme suit:

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

(CNTH 2 '(DO RE MI FA)) → **RE**
(CNTH 3 '(DO RE MI FA)) → **MI**
(CNTH 1 '((TIENS TIENS) AHA AHA)) → **(TIENS TIENS)**

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

fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)

Ecrivez une fonction qui calcule les nombres de Fibonacci pour tout argument *n* positif.
5. Que fait la fonction suivante :

```
(DE QUOI (N)
    (LET ((RES 0) (N N))
        (IF (> N 0)
            (SELF (+ (* RES 10)(REM N 10)) (/ N 10))
            RES)))
```

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

ack(0, n) = n + 1
ack(m, 0) = ack(m - 1, 1)
ack(m, n) = ack(m - 1, ack(m, n - 1))

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 **SYMEVAL**. La P-liste d'un symbole est donnée par la fonction **PLIST**. Ainsi, l'appel

(SYMEVAL 'XYZZY)

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

(PLIST 'XYZZY)

ramène la P-liste du même symbole. Notons que 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.

Une des particularités de VLISP est d'avoir une symétrie entre l'accès aux listes et l'accès aux symboles : la valeur d'un symbole est, en VLISP, le **CAR** du symbole, la P-liste d'un symbole est son **CDR**. Vous pouvez donc, en VLISP, demander le **CAR** ou le **CDR** d'un symbole, et, si le symbole **XYZZY** est lié à la valeur numérique **100**, les valeurs de l'expression LISP :

(1+ (CAR 'XYZZY))

et de l'expression

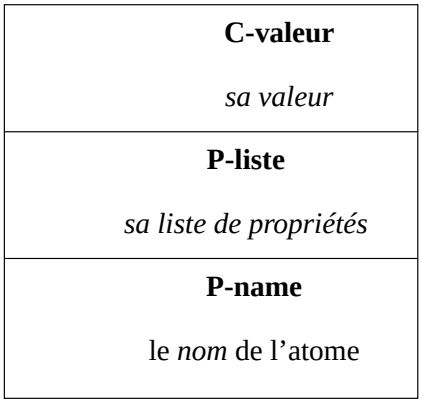
(1+ XYZZY)

sont identiques. Dans les deux cas la valeur sera **101**. De temps à autre, cette caractéristique peut être fort utile.

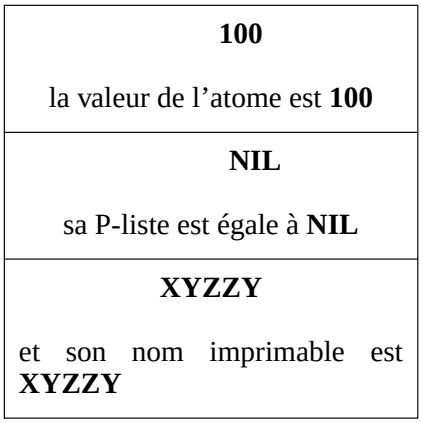
Voici à présent comment un symbole peut être représenté de façon interne :

---

[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**.



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'acces 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.



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** à 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 quatre fonctions, agissant sur la P-liste, à votre service : les fonctions **PUT**, **GET**, **REMPROP** et **ADDPROP**.

Notons qu'en LE_LISP, de même que dans quelques autres dialectes de LISP, **PUT** s'appelle **PUTPROP** et **GET** s'appelle **GETPROP**.

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** :

(PUT 'PIERRE 'PERE 'JEAN)

(PUT 'PIERRE 'MERE 'JULIE)

(PUT '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 **PUT** 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 **PUT** :

(PUT *symbole indicateur valeur***)** $\rightarrow$ *symbole*

ce qui a comme effet :

> **PUT** met sur la P-liste su 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 **PUT** est la valeur de son premier argument, donc de *symbole*.

En **MACLISP**, le dialecte LISP tournant - entre autre - 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 **PUT** admet un nombre quelconque d'arguments et est défini comme suit :

(PUT *symbole indicateur*$_1$ *valeur*$_1$ . . . *indicateur*$_n$ *valeur*$_n$**)** $\rightarrow$ *symbole*

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

(GET *symbole indicateur***)** $\rightarrow$ *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*.

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

(GET 'PIERRE 'FILS)

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

(REMPROP *symbole indicateur***)** $\rightarrow$ *symbole*

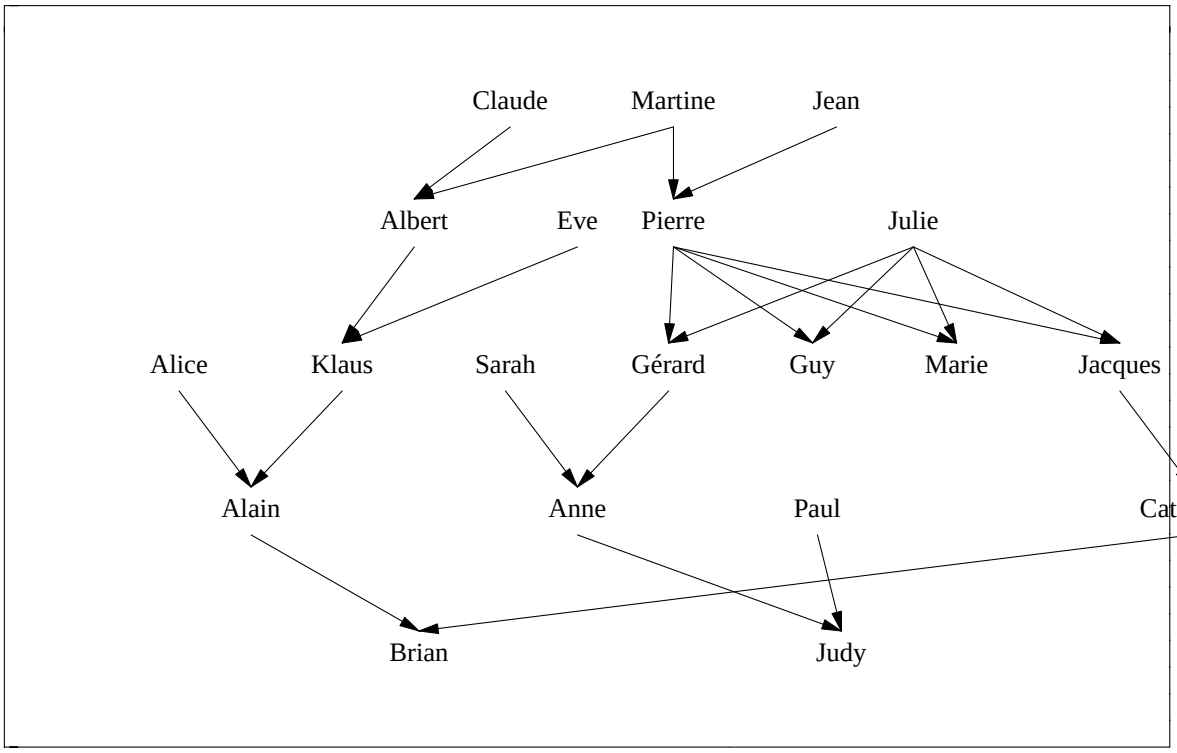
REMPROP enlève de la P-liste du symbole *symbole*, l'indicateur *indicateur*, ainsi que la valeur associée. **REMPROP** ramène le symbole donné en premier argument.

8.3. EXERCICES

1. Traduisez en LISP, à l'aide des fonctions **PUT**, **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 feminin.
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 age que la somme de l'age 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 petite 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



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 :

```
(DE 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 deuxième 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 :

```
(DE FIB (N)
    (COND
        ((= N 0) 1)
        ((= N 1) 1)
        ((GET 'FIB N))      ; c'est ICI qu'on économise ! ;
        (T (PUT '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 deuxième 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)      ; !! deuxième 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 au 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 :

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

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

(UNLESS *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

(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*$_n$**)**

peux 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 :

```
(DE 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'utilisez 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*. Eventuellement, 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 traitera ce chapitre.

10.1. LA FONCTION GENERALE D'IMPRESSION : PRINT

PRINT est la fonction LISP utilisée pour imprimer des valeurs. Cette fonction est la deuxième que nous rencontrons, qui est non seulement utilisée 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** :

$$\textbf{(PRINT } arg_1\ arg_2 \ldots arg_n\textbf{)} \quad \rightarrow \quad arg_n$$

PLUS : impression des valeurs des différents arguments

PRINT est donc une fonction à nombre quelconque d'arguments. A l'appel chacun des arguments sera évalué *et* imprimé sur le périphérique standard de sortie (écran de visualisation, papier du télétype, fichier disque ou bande magnétique, dépendant du contexte dans lequel vous vous trouvez).[2] La valeur du dernier 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 :

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

---

[1] Les deux autres fonctions à *effet de bord* que nous connaissons sont la fonction **PUT** (ou **PUTPROP**) 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.

En LE_LISP, l'impression des arguments n'est pas séparée par des espaces. Ainsi, le premier exemple ci-dessus 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 le même résultat, 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 en plus de détail.

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

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

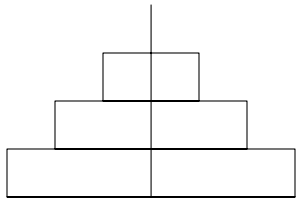
et voici quelques appels de cette fonction :

```
?  (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 *line-feed* (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 :



[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.

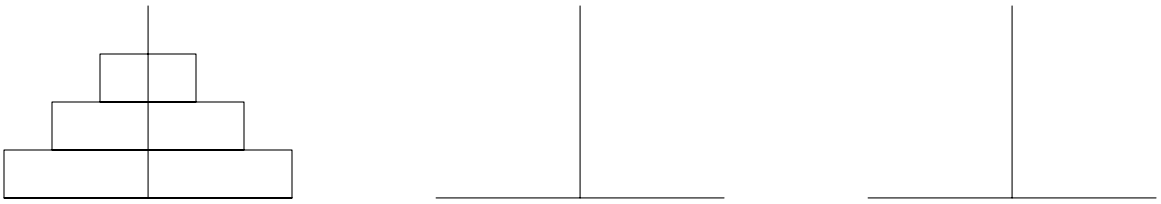
D'ailleurs, quel sera l'âge total de l'univers si le déplacement d'un disque prend une seconde ?

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 :

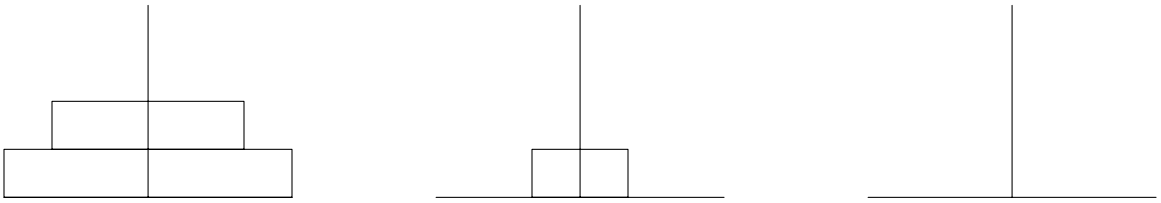
1. on ne peut déplacer qu'un disque à la fois
2. à aucun instant, un disque ne peut se trouver sur un disque de diamètre inférieur.

Regardez, voici la solution pour 3 disques :

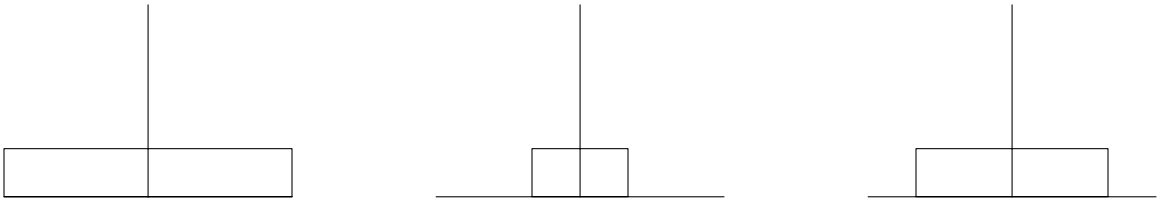
1. état initial



2. premier déplacement:

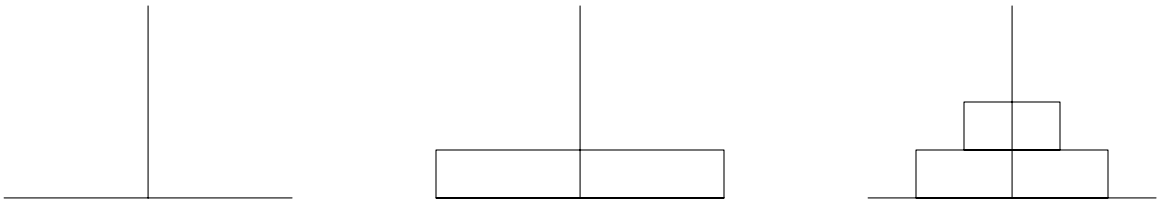


3. deuxième déplacement:

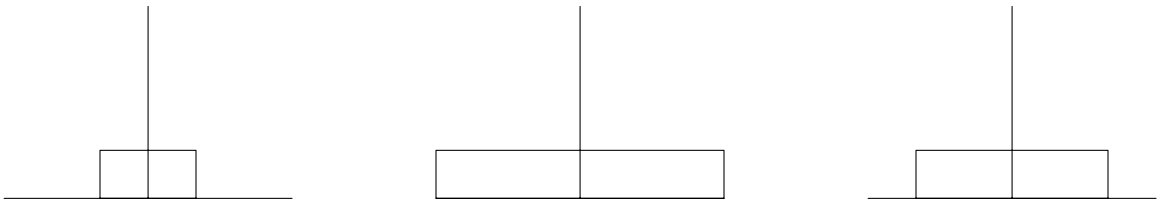


4. troisième déplacement :

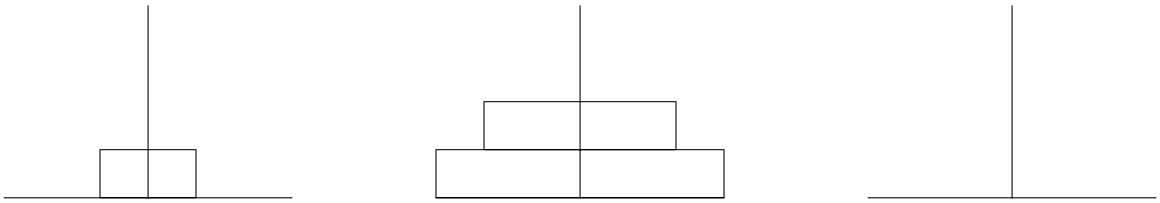
5. quatrième déplacement :



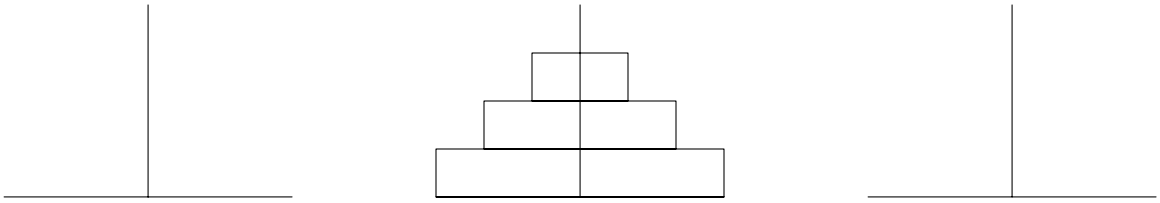
6. cinquième déplacement :



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 :

```
(DE HANOI (N DEPART ARRIVEE INTERMEDIAIRE)
   (IF (= N 0) '(ENFIN, C EST FINI)
      (HANOI (1- N) DEPART INTERMEDIAIRE ARRIVEE)
      (PRINT '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 pourrait avoir normalement des caractères pouvant faire partie d'une *chaîne* (forme courte 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 VLISP, 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 VLISP :

(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 considérée comme des atomes qui ont eux-mêmes comme valeur.

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

```
? (PRINT 1 2 3)                          ; normalement ;
  1 2 3                                  ; en LE_LISP : 123 ;
= 3

? (PRINT 1 "      " 2 "      " 3)        ; avec des espaces ;
  1       2       3
= 3

? (PRINT 1 "........." 2 "........." 3)   ; voyez vous l'utilité ? ;
  1 ......... 2 ......... 3
= 3
```

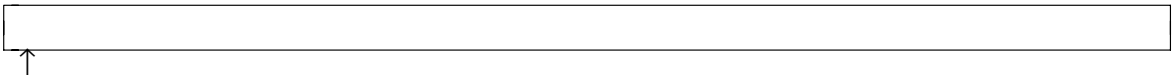
Pour l'instant rappelons-nous juste que chaque fois que nous aurons besoin d'imprimer des caractères spéciaux, nous utiliserons des chaînes de caractères.

10.3. LES FONCTIONS PRIN1, TERPRI ET PRINCH

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 VLISP, 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

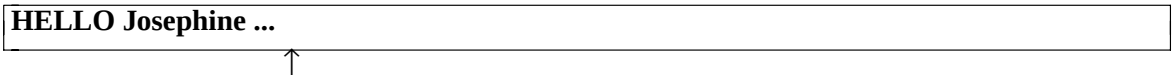
(PRIN1 'HELLO) *; en* VLISP
(PRIN 'HELLO) *; en* LE_LISP

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" "...") *; en* VLISP
(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 entrainera l'apparition de la ligne suivante :

HELLO Josephine ...

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_1 arg_2 ... arg_n**)** $\rightarrow$ arg_n

PLUS : remplissage du tampon de sortie avec les valeurs des différents arguments.

PRIN1 est donc, comme **PRINT**, une fonction à nombre quelconque d'arguments. **PRIN1** évalue tous les arguments, les introduit dans le tampon de sortie, et ramène la valeur du dernier argument (arg_n) 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 $\{n\}$**)** $\rightarrow$ n ou **NIL**

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**.

Les fonctions **PRINT** et **PRIN1** séparent les différents éléments à imprimer par un espace. Eventuellement il peut arriver que l'on ait besoin d'imprimer *sans* espaces. La fonction **PRINCH**, acronyme pour *PRIN*1 *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)
~~~~= ~
```
; C;a 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 ;

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]

```
(DE FORMAT (ELEMENTS COLONNES)
  (TERPRI)        ; pour une ligne vide ;
  (LET ((N 0)(ELEMENTS ELEMENTS)(COLONNES COLONNES))
       (IF (NULL ELEMENTS) ()      ; c'est fini ;
           (IF (NULL COLONNES) ()  ; c'est fini ;
               (SELF (+ (LET ((N N))(COND
                                    ((= N (CAR COLONNES))
                                         (PRIN1 (CAR ELEMENTS)) N)
                                    (T (PRINCH " " 1)
                                    (SELF (1+ N)))))
                                    ; pour la place de l'élément ;
                        (PLENGTH (CAR ELEMENTS)))
                     (CDR ELEMENTS)
                     (CDR COLONNES))))))
```

La construction :

```
(IF (NULL ELEMENTS) ()
    (IF (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 tests est vrai, LISP fournit la fonction **OR** (anglais pour *ou*) qui est définie comme suit :

(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 :

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

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

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**

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 :

```
(DE F1 (LISTE-D-ELEMENTS COLONNES)
  (IF (NULL LISTE-D-ELEMENTS) (TERPRI)
     (FORMAT (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 :

(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

10.4. L'ACCES AUX NOMS DES ATOMES

Comme en physique, en LISP 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 VLISP, ces caractères sont rendus accessibles par la fonction **EXPLODE** et la fusion d'un atome à partir de particules élémentaires est effectuée à l'aide de la fonction **IMPLODE**. En LE_LISP, les deux fonctions s'appellent respectivement **EXPLODECH** et **IMPLODECH**.

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

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

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

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

? (EXPLODE 123)
= (1 2 3)

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

? (CDR (EXPLODE 'UNTRESLONGAT))))
= (N T R E S L O N G A T)

? (LENGTH (EXPLODE '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 LE_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 **EXPLODE**, 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 des belles impressions, comme nous l'avons fait dans 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, 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 (EXPLODE ELE)) 0))
```

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

(PLENGTH 'HOP)	$\rightarrow$	**3**
(PLENGTH 'MARYLIN)	$\rightarrow$	**7**
(PLENGTH (CAR '(A B C)))	$\rightarrow$	**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 **EXPLODE**, est effectuée par la fonction **IMPLODE**.

IMPLODE 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 :

(IMPLODE (EXPLODE *atome*)**)** $\rightarrow$ *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 constituants une expression LISP, et livre comme résultat l'expression LISP correspondante. Nous avons donc, en LE_LISP la relation plus générale :

(IMPLODECH (EXPLODECH *expression*)**)** $\rightarrow$ *expression*

Regardons quelques exemples d'utilisation de cette fonction :

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

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

? (IMPLODE (APPEND (EXPLODE 'BON)(EXPLODE 'JOUR)))
= BONJOUR

? (IMPLODE (EXPLODE 'IDENTITE))
= IDENTITE
```

IMPLODE 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.

La suite de ce paragraphe sera un petit 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 ;*

```
(DE CREE-MOT (RACINE TERMINAISON)
  (IF (NULL TERMINAISON) RACINE
    (IMPLODE (APPEND (EXPLODE RACINE)
                     (EXPLODE TERMINAISON)))))
```

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

```
(DE RACINE (VERBE)
  (IMPLODE (REVERSE (CDR (CDR (REVERSE (EXPLODE VERBE)))))))
```

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

```
(DE TYPE (VERBE)
   (LET ((AUX (REVERSE (EXPLODE VERBE))))
      (IMPLODE (CONS (CADR AUX)
                     (CONS (CAR AUX) ())))))
```

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

```
(DE 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 le* **LET** *intérieur, elle fait imprimer la conjugaison pour chaqu'un des pronoms ;*

```
(DE CONJ1 (RACINE TYP PRONOMS)
  (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)))))
    (LET ((PRONOMS PRONOMS)(TERMINAISONS TERMINAISONS))
      (IF (NULL PRONOMS) "voila, c'est tout"
        (PRINT (CAR PRONOMS)
            (CREE-MOT RACINE (CAR TERMINAISONS)))
        (SELF (CDR PRONOMS)(CDR TERMINAISONS))))))
```

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)

devrait 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. subjonctif-imparfait (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** juste 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 divers conjugaisons du verbe 'avoir'. Introduisez une autre fonction supplémentaire, **PR-COMPOSE**, responsable de l'impression des temps composés.

2. Ecrivez un petit 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)) $\rightarrow$ **(NOUS FINISSONS AVEC CE CHAPITRE)**

ou

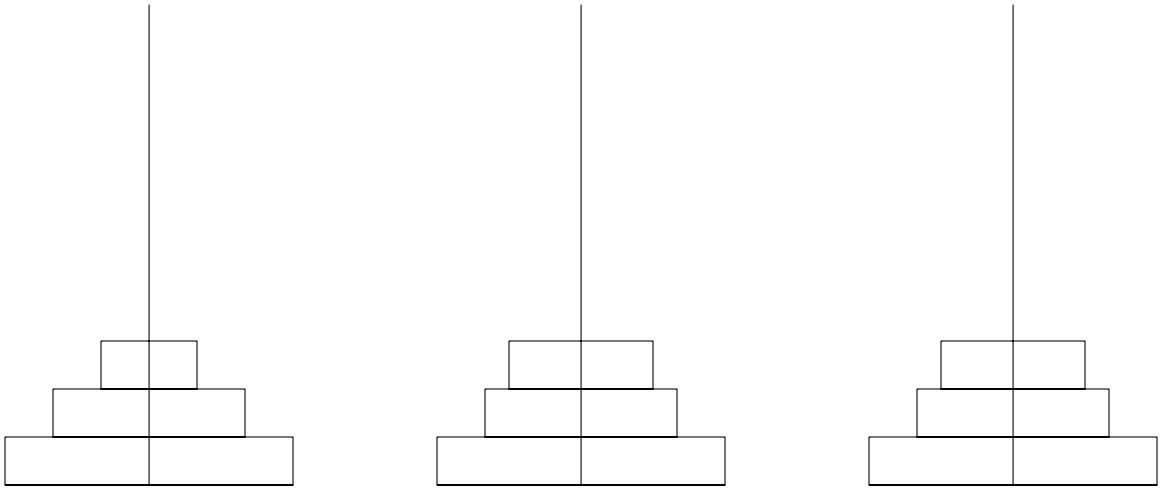
(PL '(IL ME PARLE)) $\rightarrow$ **(ILS ME PARLENT)**

ou encore

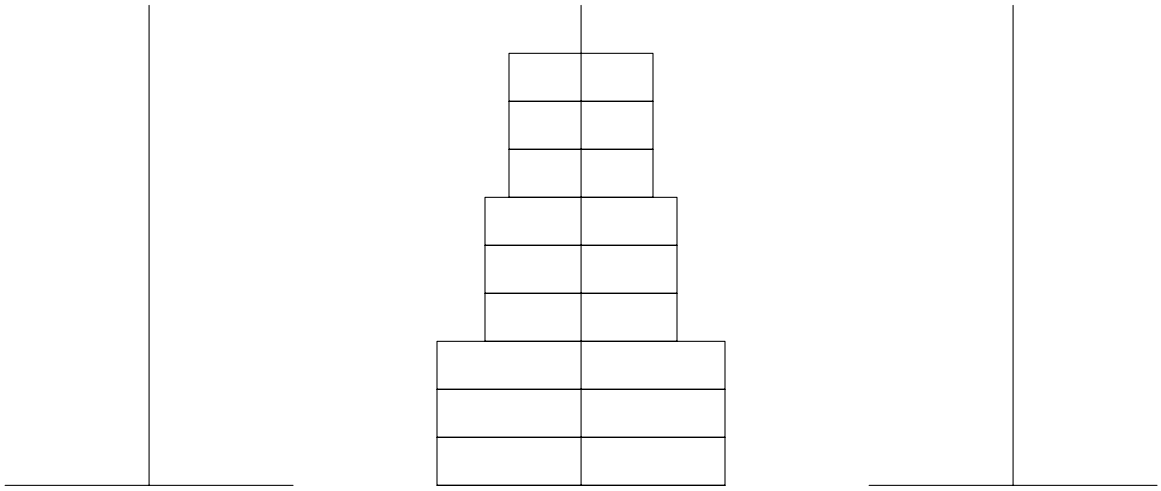
(PL '(TU NE FOULERAS PLUS CETTE TERRE))
$\rightarrow$ **(VOUS NE FOULEREZ PLUS CETTE TERRE)**

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 :

état initial :



état final :



11. LES DIVERS TYPES DE FONCTIONS UTILISATEURS : DE et DF

11.1. LA FONCTION DE

Pour définir une fonction utilisateur nous utilisons la fonction LISP 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 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 LISP, il suffit d'écrire :

(DE PLUS L (PPLUS L))

la fonction **PPLUS** étant 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 instant un autre exemple : vous avez suˆrement 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 duˆ 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-mêmes !

Regardez, à l'appel initial :

(PLUS 1 2 3 4)

la variable **L** est liée à la *liste* **(1 2 3)**, 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 nombre 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 suˆrement pas ce qu'on voulait faire (Quelle est la valeur de **(2 3 4)** ? suˆrement 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 LISP les fonctions utilisateurs définies avec la fonction **DE** s'appellent des *EXPR*s, et, dans des anciens dialectes de LISP, tels que MACLISP par exemple, les fonctions à nombre d'arguments quelconque s'appellent des *LEXPR*s ou des *NEXPR*s.

11.2. 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 graˆce à la fonction de définition de fonctions utilisateurs **DF**, qui est définie comme suit :

(DF *nom-de-fonction* **(***variable***)** *corps-de-la-fonction***)**

Syntaxiquement, **DF** est identique à **DE**, avec, en VLISP, la seule exception qu'une fonction définie par **DF** n'a qu'une seule et unique variable paramètre. A l'appel, cette variable est liée à la liste des arguments *non évalués*, donc : la liste des arguments tels quels.

En LE_LISP, **DF** peut avoir plusieurs variables. 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) *en* **VLISP**
(DF QLIST L L) *en* **LE_LISP**

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 *FEXPR*s.

11.3. 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 appels exemples :

   (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**

12.

formes ci-dessous :

(+ 1 (+ 2 3))
(+ 1 (∗ 2 3))
(∗ (+ 1 2) 3)
(∗ 1 (∗ 2 3))
. . .

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

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

Voici deux exemples d'appels de cette fonction :

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

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

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

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

(CALCUL '(1 + 2 + 3)) → **6**
(CALCUL '(1 + 2 + (3 ∗ 4) + (80 / 10))) → **23**

et finalement

(CALCUL '(2 ∗ 3 ∗ (1 - 5) + 10)) → **36**

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

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

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

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

qui imprime

```
LE  THE
```

et ramène l'atome **THE** en valeur.

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

12.2. LA FONCTION APPLY

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

APPLY est syntaxiquement définie comme :

(APPLY *fonction liste***)**

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

Donnons quelques exemples :

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

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

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

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

   et

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

2. ensuite elle évalue cette expression nouvellement construite.

Il faut avoir à l'esprit que cette fonction :

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

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

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

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

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

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

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

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

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

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

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

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

Bien évidemment, nous aurions aussi bien pu écrire :

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

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

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

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

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

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

12.3. EXERCICES

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

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

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

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

comme :

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

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

13. LES ENTREES / SORTIES (DEUXIEME PARTIE)

Il est indispensable de disposer de fonctions permettant la lecture des objets sur lesquels nous voulons travailler. 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.

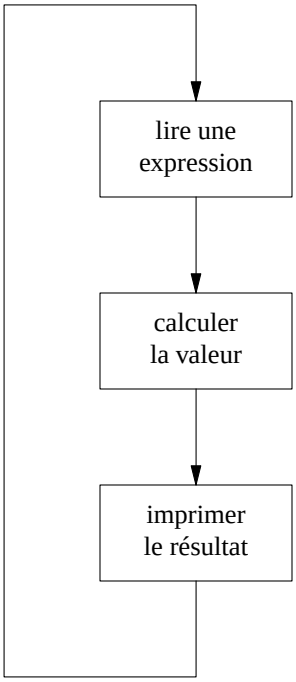
13.1. LA FONCTION GENERALE DE LECTURE : READ

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 :

```
(DE LISP ()
    (PRINT "-->" (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 :

```
(DE CALCULATEUR ()
   (PRINT "donnez une expression arithmétique s.v.p.")
   (LET ((X (READ)))
         (IF (EQ X 'FIN) 'fini
             (PRINT (CALCUL X))
             (PRINT "une autre expression arithmétique s.v.p.")
             (SELF (READ)))))
```

Ci-dessous un instantané de l'utilisation de cette fonction :

? (CALCULATEUR)	*on la lance*
donnez une expression arithmétique s.v.p.	*l'invitation*
? (1 + 2 + 3)	*la première expression arithmétique*
6	*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

Souvent nous avons envie d'imprimer un message sans aller à la ligne. Dans l'exemple ci-dessus, on aimerait imprimer le message de manière à pouvoir entrer l'expression sur la même ligne que le message. A cette fin, LISP met à votre disposition la fonction **PRINC** en VLISP ou la fonction **PRINFLUSH** en LE_LISP. Elle est tout à fait identique à la fonction **PRINT** *sauf* qu'elle ne fait pas de passage à la ligne en fin d'impression. Ainsi, si nous changeons les deux occurrences de **PRINT** (dans les deux impressions de messages) en **PRINC**, l'interaction ci-dessus apparaiˆt sur votre terminal comme :

```
? (CALCULATEUR)
 donnez une expression arithmétique s.v.p.? (1 + 2 + 3)
 6
 une autre expression arithmétique s.v.p.? (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
= fini
?
```

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)) ® [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)) ®
↑

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)) ®
↑

Il faudra un appel supplémentaire de la fonction **READ** pour *lire* également la deuxième expression. (Vous voyez la différence entre lecture physique et lecture LISP ?) Evidemment, si LISP rencontre pendant une lecture le caractère <RETURN> (représenté ici par le signe '® '), LISP relance cette fonction auxiliaire de lecture physique.

Ce qui est important, c'est de bien voir la différence entre lecture physique et l'analyse du texte lue. La fonction **READ** peut induire ces *deux* processus.

Si dans l'état

(CAR '(A B C)) (CDR '(A B C)) ®
↑

nous appelons la fonction **READCH**, LISP va lire *sans faire une analyse quelconque* le *caractère* suivant du buffer d'entrée. 'Lire' veux 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 l'état suivant :

(CAR '(A B C)) (CDR '(A B C)) ®
↑

Le caractère lue est (dans ce cas-ci) le caractère <ESPACE>, normalement un caractère séparateur ignoré.

Voici la définition de la fonction **READCH** :

(READCH) → *caractère*
lit le caractère suivant dans le flot d'entrée et le ramène en valeur sous forme d'un atome mono-caractère. **READCH** 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 **PEEKCH** est exactement comme la fonction **READCH** mais elle ne fait pas avancer le pointeur du buffer d'entrée. Le caractère lu reste donc disponible pour le **READCH** ou **READ** suivant. Voici sa définition :

[1] le signe ® indique ici le caractère <RETURN>

(PEEKCH) → *caractère*
lit le caractère suivant dans le flot d'entrée et le ramène en valeur sous forme d'un atome mono-caractère. **PEEKCH** lit vraiment n'importe quel caractère mais le laisse disponible dans le flot d'entrée pour des lectures suivantes.

LISP met deux fonctions d'entrée/sortie immédiates à votre disposition : les fonctions **TYI** et **TYO**. '*Immédiat*' veux dire que ces deux fonctions n'utilisent pas les tampons d'entrée/sortie. Voici leurs définitions respectives :

(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'apparaiˆt 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 correspondantes.

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>.

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.

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 :

```
(DE 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 petit compteur qui compte les lignes imprimées a été inseré. Après l'impression de 23 lignes,[2] 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)
   (LET ((X X)(Y 1))
        (IF (LE X 0) "boum"
            (PRINT X)
            (SELF (1- X) (IF (< Y 23) (1+ Y) (TYI) 1)))))
```

Observez donc la différence entre les deux versions de cette fonction.

Rappelons encore une fois qu'en LE_LISP l'appel de la fonction **LET** dans **COMPTE** doit être remplacé par **(LETN SELF . . . .**

Construisons maintenant un petit 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**.

```
(DE STAT-READ ()
  (LET ((X (PEEKCH))) (COND
        ((EQUAL X "(") (READCH) (READ-LIST))
        ((EQUAL X ")") (READCH) 'PARENTHESE-FERMANTE)
        ((MEMBER X '(" " "\\n" "\\t")) (READCH) (STAT-READ))
        (T (READ-ATOM)))))
```

Cette fonction débute, avec l'appel de la fonction **PEEKCH**, 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 **READCH**) 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 suˆrs 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 chaiˆnes spéciales :

"\n" et **"\t"**

qui sont des écritures commodes pour le caractère <RETURN> et le caractère <TAB> à l'intérieur des chaiˆnes de caractères. Ces caractères s'écrivent en LE_LISP **#\CR** et **#\TAB** respéctivement, mais ne sont pas interprétés dans des chaines de caractères.

Regardez bien l'utilité de la fonction **PEEKCH** : c'est elle qui permet de faire toute cette analyse en cas

[2] 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.

distincts, *sans* toucher réellement au caractère analysé. Celui-ci reste disponible pour des lectures ultérieures. On aura bien 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 :

```
(DE READ-LIST ()
    (LET ((X (STAT-READ)))
        (IF (EQ X 'PARENTHESE-FERMANTE) ()
            (CONS X (SELF (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 !

```
(DE READ-ATOM ()
    (LET ((X (READ)))
        (IF (NUMBERP X) X
            (IF (GET X 'USE)
                (PUT X 'USE (1+ (GET X 'USE)))
                (PUT 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 :

```
(DE LISP ()
    (PRINC "une expression: ")
    (LET ((X (STAT-READ)))
        (IF (NEQ X 'PARENTHESE-FERMANTE) (PRINT (EVAL X)))
        (PRINC "une expression: ")
        (SELF (STAT-READ))))
```

Ci-dessous une petite fonction qui imprime nos statistiques. Cette fonction utilise la 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*.[3] 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 ()
   (LET ((Y (OBLIST)))
        (IF (NULL Y) ()
            (IF (GET (CAR Y) 'USE)
                (PRINT (CAR Y) (GET (CAR Y) 'USE)))
        (SELF (CDR Y)))))
```

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*
USE	*sa valeur ??!!??*

Que se passe-t-il ?

[3] L'obliste peut être considérée comme un sorte de micro-manuel LISP, accessible en ligne.

C’est très compliqué ! Regardons : le tout est évidemment duˆ 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>’, ’<TAB>’ 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 **PUT**. Horreur ! Nous étions suˆrs que la valeur de **X** serait toujours un atome; ici c’est une liste ! D’ailleurs, la fonction **PUT** attend un argument de type atome. Que fait donc la fonction **PUT** ? Soit qu’elle produit une erreur, soit qu’elle modifie, comme en VLISP, la P-liste de l’atome argument. Rappelons-nous que la P-liste est (en VLISP) le **CDR** de l’atome. La fonction **PUT** insère donc dans le **CDR** de son argument le couple <indicateur, valeur>. Ici, ça veut dire que l’appel :

   (PUT ’(QUOTE a) ’USE 1)

   qui ne correspond pas à la définiton de **PUT**, aura, en VLISP, comme résultat la nouvelle liste :

   (QUOTE USE 1 a)

   C’est cette liste qui sera le résultat de l’appel de la fonction **READ-ATOM**.
4. Finalement, de retour dans la fonction **LISP** nous aurons encore à calculer la valeur de l’expression
   (QUOTE USE 1 a)

   et, étant donné que la valeur d’un appel de la fonction **QUOTE** est son premier argument *tel quel*, le résultat final sera donc l’atome **USE**. On est loin de ce qu’on voulait calculer, c.à.d.: **a**.

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 :

```
(DE STAT-READ ()
  (LET ((X (PEEKCH))) (COND
        ((EQUAL X "(") (READCH) (READ-LIST))
        ((EQUAL X ")") (READCH) 'PARENTHESE-FERMANTE)
        ((EQUAL X "'") (READCH) (LIST 'QUOTE (STAT-READ)))
        ((MEMBER X '(" " "\\n" "\\t")) (READCH) (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)
CAR 1                                     regardons si ça compte bien le nombre d'occurrences
CONS 1                                    de chacun des atomes
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 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

Visiblement les divers caractères du jeu ascii n'ont pas tous le même effet pendant la lecture. La plupart sont considérée comme des caractères constituants des noms des atomes. Par contre, les parenthèses, l'espace <ESPACE>, la tabulation <TAB>, le return <RETURN>, le point-virgule, le double-quote et le quote-caractère 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 soit le caractère *début de commentaire*, soit le caractère *fin de commentaire*. Le caractère <RETURN> est également un caractère *fin de commentaire*. Si vous voulez mettre des commentaires à l'intérieur de votre programme, ils doivent donc débuter par un point-virgule. Les commentaires se terminent soit par un deuxième point-virgule ou par un <RETURN> en VLISP, soit uniquement par un <RETURN> en LE_LISP. Le lecteur LISP standard ignore tout commentaire.
"	Le doublequote délimite des chaîˆnes de caractères. A l'intérieur d'une chaîˆne de caractères aucun contrôle quant au rôle des caractères n'est effectué. Par contre, comme dans le langage C, il existe en VLISP quelques conventions pour y mettre des caractères bizarres. Ainsi, pour mettre une fin de ligne (un <RETURN>) à l'intérieur d'une chaîˆne, vous pouvez écrire ´\\n´, pour y mettre une tabulation vous pouvez utilisez ´\\t´ et pour un 'backspace´ utilisez ´\\b´.
(,)	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*.
---	---
	Tous les autres caractères (sauf le point (**.**), ainsi que le crochet ouvrant ([) et le crochet fermant (]), et en LE_LISP le caractère **#** et le caractère **|**, qui seront examinés par la suite) n'ont aucun rôle particulier et peuvent donc être utilisés pour les noms des atomes.

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 **TYPCH** (cette fonction s'appelle **TYPECH** en LE_LISP) qui vous donne le TYPe du CHaractère. Voici sa définition :

(TYPCH *e***)** → *type*
l'argument *e* peut être une chaîne de caractères (d'un unique caractère) ou un atome mono-caractère. *type* est le type courant de ce caractère.

(TYPCH *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 *type* d'un caractère est définie comme suit :

type **VLISP**	*type* **LE_LISP**	*caractère*
0	CNULL	caractères ignorés
1	CBCOM	début de commentaire
2	CECOM	fin de commentaire
rien	CQUOTE	caractère QUOTE en LE_LISP
4	CLPAR	parenthèse ouvrante
5	CRPAR	parenthèse fermante
6	*rien*	crochet ouvrant en VLISP
7	*rien*	crochet fermant en VLISP
8	CDOT	le point **.**
9	CSEP	séparateur
10	CMACRO	macro-caractère
11	CSTRING	délimiteur de chaînes
12	CPNAME	caractère normal
rien	CSYMB	délimiteur symbole spécial |
rien	CPKGC	délimiteur de package
rien	CSPLICE	splice-macro
rien	CMSYMB	symbole mono-caractère

Voilà, muni de ces connaissances et de cette nouvelle fonction, vous pouvez écrire des choses étranges. Par exemple, si les parenthèses ne vous plaisent plus, après les instructions

(TYPCH "{" (TYPCH "("))
(TYPCH "}" (TYPCH ")"))

Vous pouvez aussi bien écrire

(CAR '(DO RE MI))

que

{CAR '{DO RE MI}}

13.3.2. les macros-caractères

Les macro-caractères modifient donc 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 la fonction de Définition de Macro-Caractères : **DMC**. Cette fonction est syntaxiquement définie comme suit :

(DMC *c* **(***variable*$_1$ *variable*$_2$ . . . *variable*$_n$**)** *corps-de-fonction***)**

Le premier argument, *c*, peut être une chaîne ou un atome mono-caractère. Lorsque le lecteur LISP rencontrera ce caractère, il lancera l'évaluation du *corps-de-fonction*, les variables de la liste de variables, $variable_1$ à $variable_n$, serviront de variables locales si besoin est. 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)))

En LE_LISP, il est préférable de quoter le nouveau macro-caractère par le caractère spécial '|'. La définition du macro-caractère quote s'écrit donc, en LE_LISP :

(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-caractères '[' 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 "[" ()
   (LET ((X 'LIST))
      (AND (NEQUAL X "]") (CONS X (SELF (READ))))))

(DMC "]" () "]")
```

La fonction **NEQUAL** est bien évidemment une abréviation pour **(NULL (EQUAL**,[4] 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

[4] Si cette fonction n'existe pas dans votre LISP, vous pouvez la définir aisément comme :

(DE NEQUAL (X Y) (NULL (EQUAL X Y)))

'[' et ']' pour générer des appels de la fonction **LIST**.

Construisons donc un dernier jeu de macro-caractères et définissons le macro-caractère *back-quote* (**‘**). Ce macro-caractère, d'une grande utilité, 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** suivi 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 *back-quote* 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)**

Visiblement, le macro-caractère backquote utilise deux macro-caractères supplémentaires : le virgule (**,**) et l'escargot (**@**). A l'intérieur d'une liste *back-quotée* 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 back-quotée.

Il ne reste qu'à écrire la définition de ce macro-caractère, dont voici la définition (très compliquée) :

```
(DMC "‘" ()
   (LET ((VIRGULE (TYPCH ",")))
      (TYPCH "," (TYPCH "’"))
      (LET ((QUASI [’EVAL [’BACKQUOTE [’QUOTE (READ)]]]))
         (TYPCH "," 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 **‘** (backquote) le macro-caractère **,** (virgule) soit défini, à la sortie du backquote nous restaurons, par l’instruction

(TYPCH "," VIRGULE)

le type de ce caractère à l’extérieur du backquote. Ensuite, nous déléguons toute l’activité du backquote 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** :

```
(DE 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 **‘** (backquote). 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** :

```
(DE 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** :

```
(DE ISCONST (X)
    (OR (NULL X) (EQ X T)(NUMBERP 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 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 (TYPCH ",")))
    (DMC "," ()
        (LET ((X (PEEKCH)))
            (IF (NULL (EQUAL X "@")) ['*UNQUOTE* (READ)]
                (READCH)
                ['*SPLICE-UNQUOTE* (READ)])))
    (TYPCH "," VIRGULE))
```

Regardez bien l'utilisation de la fonction **PEEKCH** !

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 tout 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)

Arrêtons ici avec les macro-caractères d'entrée (nous y reviendrons) et 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 petit paragraphe vous montrera comment réaliser ce genre d'opérations.[5]

[5] Les entrées/sorties sur fichiers sont une des choses qui changent le plus d'un système LISP à l'autre. Dans ce court paragraphe nous ne donnons que quelques fonctions significatives d'*un* système LISP particulier. Avant de tester ces fonctions sur votre version personnelle de LISP, consultez donc d'abord votre documentation

Tout d'abord, LISP met à votre disposition une fonction de *lecture* de fichiers. Cette fonction s'appelle **LIB** en VLISP et **LOAD** en LE_LISP. Elle est fort utile pour *charger* un programme qui se trouve dans un fichier. Voici la définition de **LIB** :

(LIB *nom-de-fichier***)** → *nom-de-fichier*
et lit le fichier de nom *nom-de-fichier* en entier.

Ainsi, si vous travaillez sous un système UNIX, et si le programme du paragraphe précédent (la définition du *backquote*) se trouve dans le fichier **backquote.vlisp**, la commande

(LIB backquote.lisp) ou **(LOAD "backquote.lisp")**

va lire ce fichier, définition par définition.

L'argument *nom-de-fichier* peut être une chaiˆne de caractères ou un atome.

Si vous travaillez sous VLISP, et si le nom de votre fichier se termine par *.vlisp*, vous pouvez omettre cette partie. L'écriture :

(LIB backquote)

est donc tout à fait suffisante pour lire le fichier **backquote.vlisp**. En LE_LISP, l'extension *.ll* sert le même rôle : pour lire le fichier **backquote.ll**, il suffit d'écrire :

(LOAD "backquote")

Notez toutefois que si vous avez réellement un fichier nommé 'backquote´, ce sera ce fichier qui sera lu, et non le fichier 'backquote.vlisp' ou 'backquote.ll' !

En VLISP, si vous voulez lire un fichier, dont vous n'êtes pas suˆr qu'il existe, le prédicat **PROBEF** répond à la question : "est-ce que tel ou tel fichier existe ?". ce prédicat est défini comme :

(PROBEF *nom-de-fichier***)** → *nom-de-fichier* si le fichier de ce nom existe
→ **NIL** si aucun fichier de nom *nom-de-fichier* n'existe

Donc : si, dans un programme, vous voulez lire un fichier particulier et que vous êtes très angoissé quant à son existence, la suite d'instruction :

(IF (PROBEF *nom***) (LIB** *nom***)** *message-d'erreur***)**

est la plus adéquate.

De même que pour la fonction **LIB**, l'argument *nom-de-fichier* de **PROBEF** peut être un atome ordinaire ou une chaîne de caractères.

Sous VLISP-UNIX il existe une généralisation de la fonction **LIB**, c'est la fonction **INCLUDE**.

Elle a la même syntaxe et la même sémantique (le même sens) que **LIB**, mais, cette instruction peut se trouver à l'intérieur d'un fichier que vous êtes en train de charger. Ainsi vous pouvez *pendant* le chargement d'un fichier lancer la commande de chargement d'un autre fichier.

Cette fonction **INCLUDE** n'est pas nécessaire en LE_LISP : la fonction **LOAD** permet pendant le chargement d'un fichier le rencontre d'autres appels de **LOAD**.

---

technique !

Afin de rendre ceci plus explicite, imaginez que vous ayez un fichier nommé **foo.vlisp** qui contienne - entre autre - un appel de **INCLUDE** du fichier **bar.vlisp** :

fichier **foo.vlisp** :

premières définitions
. . .
(INCLUDE bar)
reste des définitions
. . .

Quand vous lancez la commande :

(INCLUDE foo)

LISP va commencer à lire les *premières définitions* du fichier **foo.vlisp** jusqu'à la rencontre de l'instruction :

(INCLUDE bar)

Là, LISP va interrompre la lecture du fichier **foo.vlisp**, lire tout le fichier **bar.vlisp** (et éventuellement d'autres fichiers, si **bar.vlisp** contient des appels de **INCLUDE**), pour ensuite continuer à lire le *reste des définitions* du fichier **foo.vlisp**.

Vous pouvez ainsi *inclure* jusqu'à 10 niveaux de fichiers. C'est très utile de temps à autre.

Voilà, maintenant vous pouvez garder vos programmes d'une session à l'autre !

Remarquez que les fonctions **LIB** et **INCLUDE**

1. lisent tout le fichier et
2. n'évaluent pas leur argument, nul besoin de quoter le nom de fichier.

Vous ne pouvez donc pas utiliser ces deux fonctions pour lire par programme une expression après l'autre. Pour faire ceci, c.à.d.: pour lire des données d'un fichier une à une, vous devez utiliser la fonction **INPUT**. Cette fonction ne fait rien d'autre qu'indiquer à la machine *où* doivent lire les appels des fonctions **READ**, **READCH** ou **PEEKCH** ultérieurs.

Voici la définition de cette fonction :

(INPUT *atome***)** → *atome*
indique à la machine LISP que les lectures suivantes se font à partir du fichier de nom *atome*. Si *atome* est égal à **NIL**, les lectures suivantes se font à partir du terminal.

C'est comme si LISP ne pouvait se connecter, à un instant donné, qu'à *un seul* fichier d'entrée particulier et que la fonction **INPUT** réalise les différentes connections nécessaires. Bien entendu, comme sous système UNIX, le terminal n'*est* rien d'autre qu'un fichier particulier.

Prenons un exemple, et supposons qu'on ait un fichier, nommé **valeurs.vlisp**, contenant la suite d'atomes que voici :

1
2
3
4
5

6
7
8
9
10
FIN

et construisons une fonction qui lit sur un fichier un nombre après l'autre, jusqu'à rencontrer l'atome **FIN**, et calcule leur somme. La voici :

```
(DE SOMME-DE-NOMBRES (FICHIER)
    (INPUT FICHIER)
    (SOMME-AUX 0))

(DE SOMME-AUX (X)
    (IF (NEQ X 'FIN) (+ X (SOMME-AUX (READ)))
        (INPUT ())  ; pour revenir vers le clavier ;
        0))
```

Pour calculer la somme des nombres contenus dans le fichier **valeurs.vlisp**, il suffit d'appeler cette fonction comme :

```
(SOMME-DE-NOMBRES "valeurs.vlisp")
```

Finalement, pour terminer ce chapitre, regardons comment écrire les résultats d'un calcul vers un fichier avec la fonction **OUTPUT**. Cette fonction est l'inverse de la fonction **INPUT**. Voici sa définition :

(OUTPUT *atome***)** → *atome*
indique à la machine LISP que les impressions suivantes se font vers le fichier de nom *atome*. Si *atome* est égal à **NIL**, les impressions suivantes se font vers le terminal.

La fonction **OUTPUT** crée le fichier de nom *atome* s'il n'existe pas encore. Par contre, s'il existe déjà, les impressions suivantes seront ajoutées à la fin du fichier.

Pour prendre un exemple, voici la fonction **CREE-VALEURS** qui génère les valeurs nécessaires pour notre fonction **SOMME-DE-NOMBRES** ci-dessus :

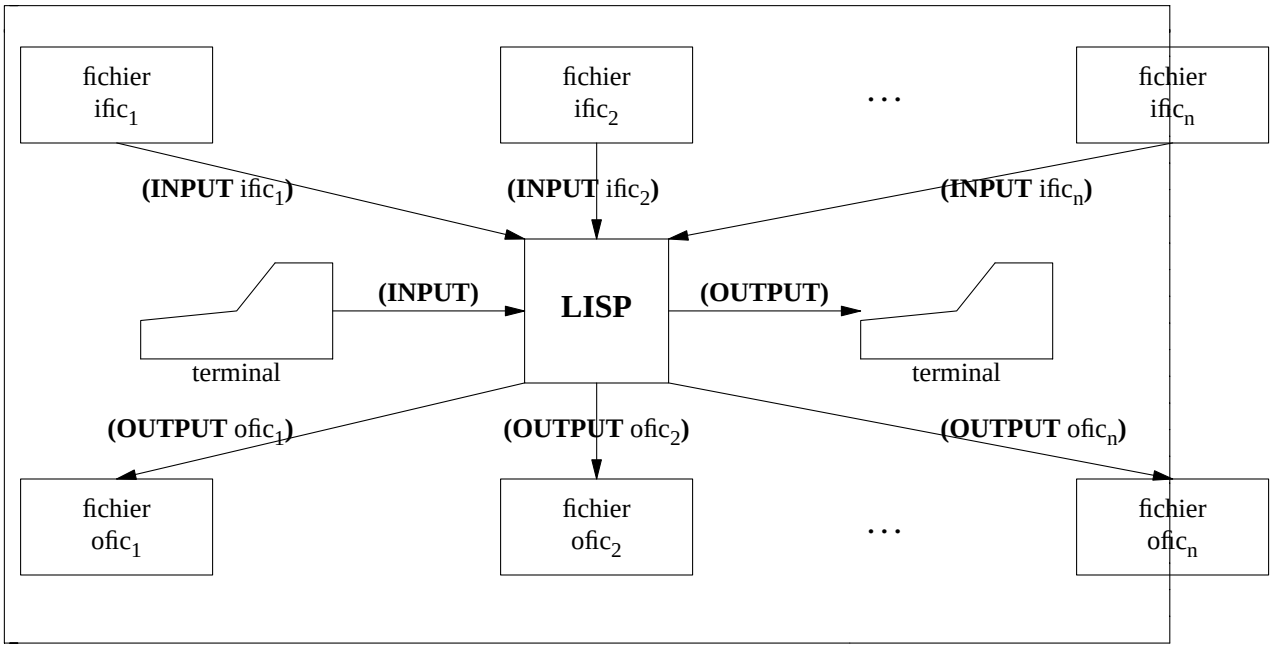
```
(DE CREE-VALEURS (N FICHIER)
    (OUTPUT FICHIER)
    (CREE-AUX 1))

(DE CREE-AUX (X) (COND
    ((> X N) (PRINT 'FIN)
             (OUTPUT ()))  ; pour revenir vers le terminal ;
    (T (PRINT X) (CREE-AUX (1+ N)))))
```

Pour créer le fichier **valeurs.vlisp** ci-dessus, il suffit d'activer cette fonction par l'appel :

```
(CREE-VALEURS 10 "valeurs.vlisp")
```

Voici graphiquement, les possibilités d'échanges entre LISP et le monde extérieur :



13.5. EXERCICES

1. Au lieu d'ecrire **(LIB** *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. Redéfinissez les macro-caractères >= et <= pour **GE** et **LE** respectivement.
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 pair et impair 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 chaque problème de comparaison indépendamment. 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 :

```
(DE DOREMI (L)
    (LET ((X (MEMQ 'DO L))) (COND
          ((NULL L) ())
          ((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 :

```
(DE BIDON (L)
  (IF (AND (= (LENGTH L) 3) (EQ (CAR (REVERSE L)) 'MI)) T NIL))
```

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 quelconque 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 se 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 : qu'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) :

```
(DE 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 :

```
(DE 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 quelconque 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 d'implémenter 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** :

```
(DE MATCH (FILTRE DONNEE) (COND
        ((ATOM FILTRE) (EQ FILTRE DONNEE))
        ((ATOM DONNEE) NIL)
        ((EQ (CAR FILTRE) '$)
            (MATCH (CDR FILTRE) (CDR DONNEE)))
        ((EQ (CAR FILTRE) '&) (COND
            ((MATCH (CDR FILTRE) DONNEE))
            (DONNEE (MATCH 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 unique* élément et l'autre filtre *une séquence* (éventuellement vide) d'éléments.

Faire des tentatives d'appels d'une fonction est très courante 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 répétons 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 cela 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 &)

   Pour donner 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*)

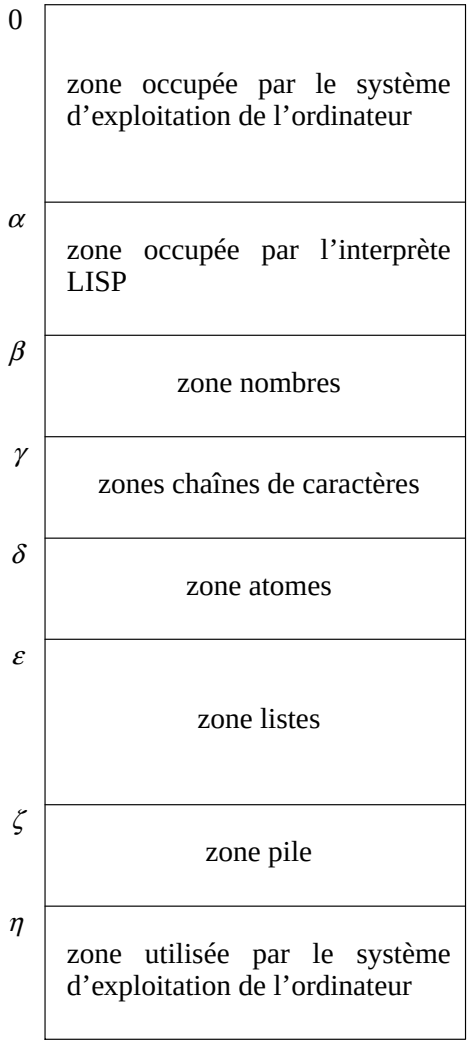
ne filtre que les quatre listes : **(A A) (A B) (B A) (B B)**.

15. RETOUR VERS LES LISTES

15.1. L'ORGANISATION DE LA MEMOIRE LISP

Comment LISP utilise-t-il la mémoire de l'ordinateur ? 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 votre ordinateur.

La figure de la page suivante montre graphiquement l'utilisation de l'espace mémoire de votre ordinateur sous LISP :[1]



Les numéros a‘ gauche du tableau représentent des adresses. Ainsi, dans notre figure, les premiers α mots

[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.

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 puisse pas eˆtre utilisée par LISP.

Regardons donc chacune des zones de LISP : la première, allant de l'adresse α a' l'adresse β-1, est occupée par l'interprète LISP meˆme. C'est ici qu'on trouvera les programmes d'entrée/sortie, la petite boucle *toplevel* que nous avons vu au chapitre 13, etc. Cette zone n'est pas accessible pour l'utilisateur : c'est la zone système de votre machine LISP, de meˆme 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 intermédiares. Nous verrons l'utilité de cette zone plus tard. Concentrons-nous sur la zone liste, puisque c'est elle qui nous intéresse pour l'instant.

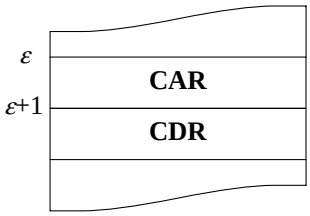
15.2. LA ZONE LISTE

Les listes ont été inventées pour pouvoir utiliser la mémoire sans contrainte de taille. Les idées de base sont :

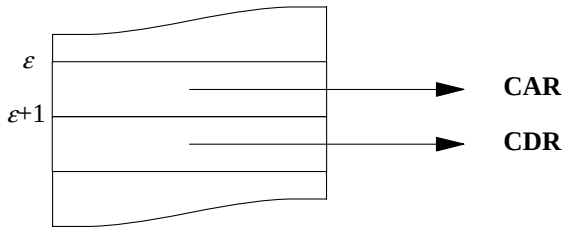
1. d'organiser cette mémoire de manière non continue : une liste ne doit pas obligatoirement occuper des mots mémoires contigus. Pour cela, les listes sont implémentées tel qu'il soit toujours possible a' partir d'*un* élément de la liste de trouver le suivant.
2. de disposer d'un algorithme de *récupération 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 a' nouveau dans l'espace utilisable.

Tout cela a' l'air bien abstrait ! Examinons donc l'organisation de cette zone en plus de 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 a' 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 adjacent de cette zone :

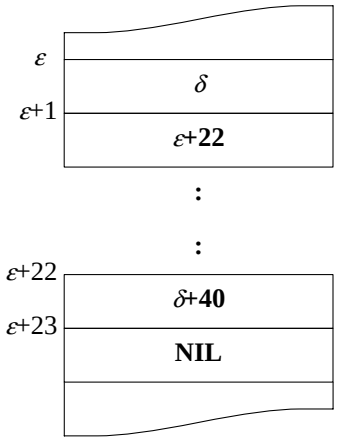


Chacune des boîtes représente un mot mémoire dans la zone liste. Les étiquettes **CAR** et **CDR** a' l'intérieur des boîtes sont des *pointeurs* vers les objets LISP constituant le **CAR** et le **CDR** de cette liste. Graphiquement :



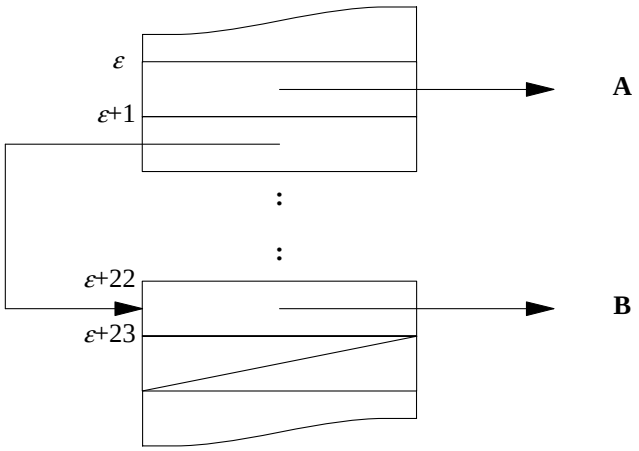
Le **CAR** ou le **CDR** d'une liste peuvent eˆtre de n'importe quel type : atome ou liste. De fait, les pointeurs CAR et CDR peuvent pointer dans n'importe quel *zone* des objets LISP. Ce sont donc, en réalité, des adresses de la mémoire.

Ainsi, disposant d'une liste **(A B)**, d'un atome **A** se trouvant en mémoire a' l'adresse δ, et d'un atome **B** se trouvant a' l'adresse δ+40, cette liste peut eˆtre implémentée en mémoire de la manière suivante :

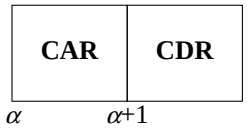


Les listes sont donc des couples (on dit : des *doublets*) de mots constituées d'un pointeur vers l'emplacement de la valeur du **CAR** et d'un pointeur vers l'emplacement de la valeur du **CDR**, c'est-a'-dire : l'*adresse* du mot mémoire où se trouve la valeur correspondant. La fin d'une liste est indiquée par un pointeur vers l'atome *très* particulier **NIL**.

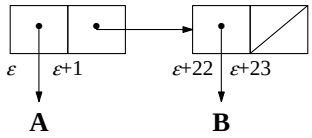
Afin de ne pas surcharger la représentation graphique par trop d'adresses, nous allons représenter les pointeurs vers des listes par des flèches vers les doublets correspondants, la fin d'une liste par une diagonale dans le **CDR** du dernier doublet et au lieu de marquer les adresses des atomes, nous allons juste indiquer des pointeurs vers les noms des atomes. Ce qui donne pour la meˆme liste **(A B)** :



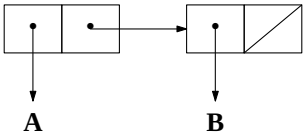
Bien évidemment, nous ne pouvons pas constamment prendre en considération les adresses réelles, c'est pourquoi nous utiliserons une représentation graphique des listes qui ne prend en considération que les *doublets* directement utilisés dans une liste. Une liste sera donc représentée comme une suite de *doublets* composée des **CAR**s et **CDR**s de cette liste :



La liste **(A B)** de ci-dessus sera donc représentée comme :

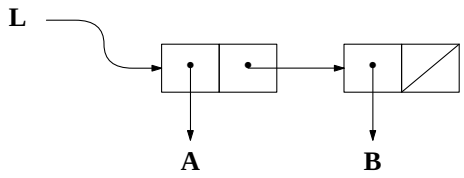


Normalement, quand nous allons donner une représentation graphique d'une liste, nous allons ignorer les adresses d'implémentation réelle et dessiner, par exemple, la liste **(A B)**, comme :



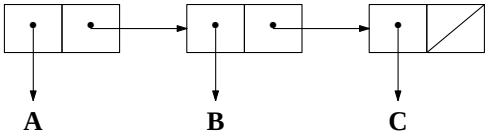
Mais, meˆme si nous prenons cette représentation (plutôt) abstraite, il ne faut jamais oublier qu'en réalité toute la représentation interne se fait en terme d'adresses réelles !

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**

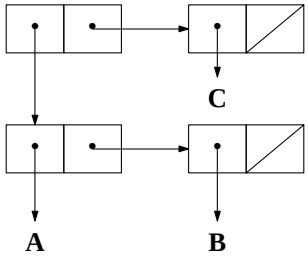


On peut aisément reconnaître ce que font la fonction **CAR** et **CDR** : elles ramènent respectivement le pointeur du champ **CAR** et **CDR**.

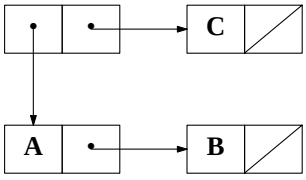
Regardons quelques exemples de cette représentation : La liste **(A B C)** sera donc implémentée comme :



et la liste **((A B) C)** sera, en terme de doublets, implémentée comme :

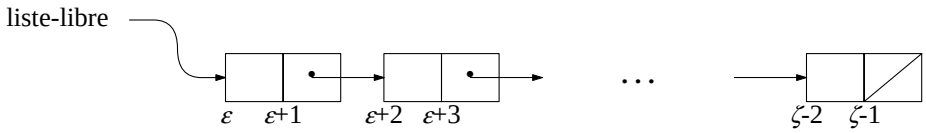


Nous pouvons encore simplifier en ne fléchant que les pointeurs vers des listes; ce qui donne pour cette dernière liste :



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.

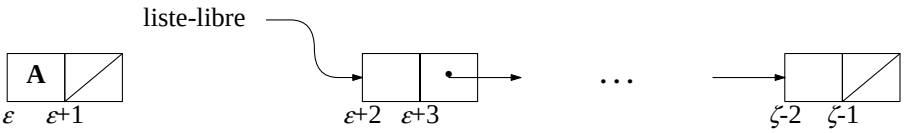
Au lancement de LISP toute la zone liste est organisée comme une *liste libre*, c’est-a‘-dire : tous les doublets sont *chaînés* entre eux pour former une énorme liste :



Chaque appel de la fonction **CONS** enlève un doublet de cette liste libre. Ainsi, si le premier appel de **CONS** est

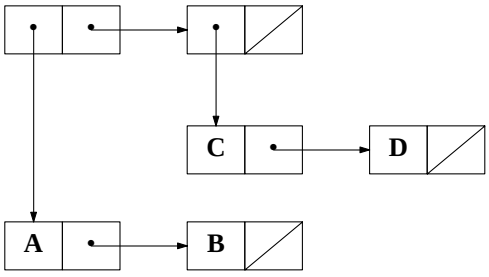
(CONS 'A NIL)

CONS cherche d'abord un doublet dans la liste libre, l'enlève de cette liste, insère dans le champ **CAR** de ce doublet un pointeur vers l'atome **A** et dans le champ **CDR** de ce meˆme doublet un pointeur vers l'atome **NIL**. ce qui donne:

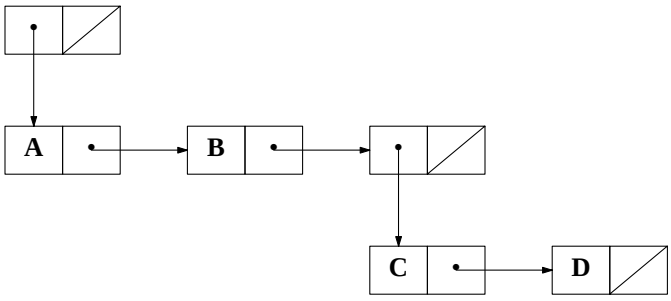


Quand la liste libre est vide, **CONS** met en marche un mécanisme de récupération de mémoire, qui construit une nouvelle liste libre a' partir des doublets qui ne sont plus nécessaires, c'est-a'-dire : a' partir des doublets qui ne sont plus utilisés.

Ci-dessous encore deux exemples de représentation de listes sous forme de doublets : d'abord la liste **((A B) (C D))** :

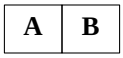


et ensuite la liste **((A B (C D)))** :

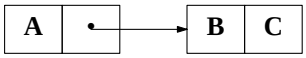


Si nous avons dit que le contenu du champ **CAR** ou du champ **CDR** d'un doublet peut eˆtre un pointeur vers n'importe quel objet LISP, il peut s'avérer que nous trouvions dans le champ **CDR** d'un doublet un pointeur non pas vers une liste mais un pointeur vers un atome, comme, par exemple dans le 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 telles 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 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 a‘ partir de symboles atomiques, les caractères "**(**", "**)**" et "**.**" (le point). L'opération de base pour former une S-expression consiste a‘ en combiner deux pour produire une S-expression plus grande. Par exemple, a‘ 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 :

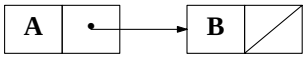
A

évidemment, c'est un atome et ne peut donc pas eˆtre représenté sous forme de doublets.

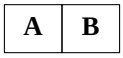
(A . NIL)

[2] rappelons qu'un pointeur vers l'atome **NIL** dans le champ **CDR** indique la fin d'une liste.

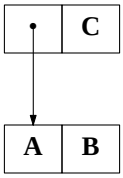
(A . (B . NIL))



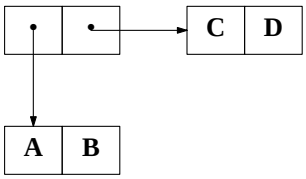
(A . B)



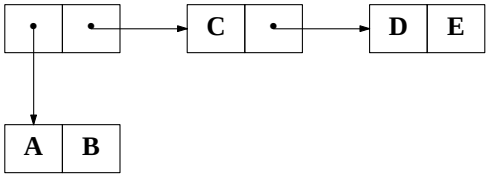
((A . B) . C)



((A . B) . (C . D))



((A . B) . (C . (D . E)))



Pour passer de cette notation des S-expressions a‘ la notation des listes a‘ laquelle nous sommes habituées, il suffit de

1. remplacer chaque occurrence d’un point suivi d’une parenthèse ouvrante par un caractère blanc 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 de ci-dessus nous avons les correspondances que voici :

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-a‘-dire : les doublets où le **CDR** est un pointeur vers un atome différents de **NIL**, restent représentés a‘ l'aide du caractère '**.**'.

15.3. LES FONCTIONS DE MODIFICATION DE LISTES

Les fonctions de traitement de listes que nous connaissons jusqu'a‘ maintenant, **CAR**, **CDR**, **REVERSE** etc, ne modifient pas les listes données en argument. Ainsi, si a‘ 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 étudierons quelques fonctions de traitement de listes *destructives*, c'est-a‘-dire, qui modifient après évaluation les listes passées en argument.

Les deux fonctions les plus importantes 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*
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*
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 A B)**
(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 :

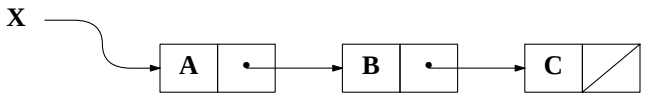
```
(DE FOO (X)
    (CONS (RPLACA X 1) X))
```

Que fait cette fonction ? Est-ce que le résultat est le meˆme si nous remplaçons le corps par **(CONS (CONS 1 (CDR X)) X)** ?

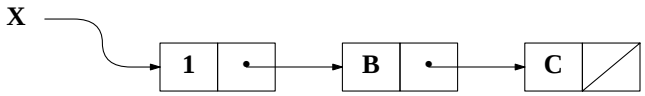
Regardons donc en détail. Si nous appelons cette fonction comme :

(FOO '(A B C))

la variable **X** sera liée a‘ 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 a‘ 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)

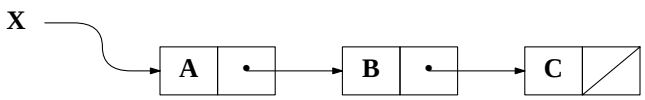
et non point la liste **((1 B C) A B C)**.

Quand vous utilisez ces fonctions, faites attention, leur usage est très dangéreux, soyez suˆrs que vous voulez vraiment *modifier* la structure interne des listes !

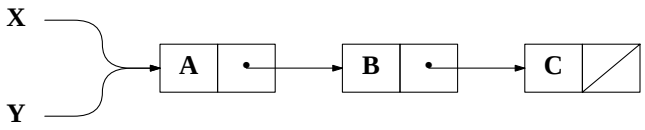
Imaginons que la personne qui a construit le programme **FOO** ci-dessus veuille, en réalité, construire une fonction qui produise une liste contenant en **CAR** la meˆme liste que le **CDR** avec le premier élément remplacé par l'atome **1**, qu'il voulait donc que **(FOO '(A B C))** livre **((1 A B) A B C)**. Constatant l'erreur, elle construit alors la nouvelle version que voici :

```
(DE FOO (X)
  (LET ((X X) (Y X))
       (CONS (RPLACA X 1) Y)))
```

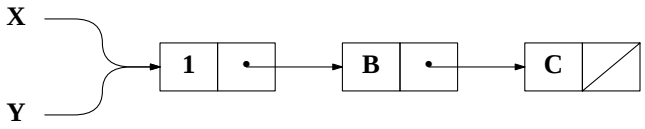
Est-ce que cette solution est meilleure ? L'idée est claire : puisque **RPLACA** modifie la liste, on sauvegarde, avant de faire le **RPLACA**, la valeur de **X** dans la variable **Y**, et cette variable sera ensuite le deuxième argument du **CONS**. Regardons donc ce qui se passe, et appelons a‘ nouveau **(FOO '(A B C))**. Comme préalablement, la variable **X** sera liée a‘ la liste **(A B C)** :



ensuite, dans le **LET**, **X** sera reliée a‘ cette meˆme liste, et **Y** sera liée a‘ la valeur de **X** ce qui donne :



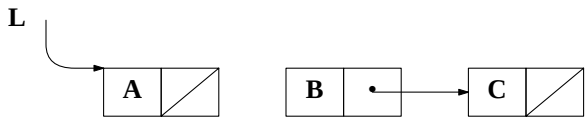
ensuite, comme préalablement, l'évaluation du premier argument du **CONS** livre la liste **(1 B C)** et modifie la valeur de **X**, ce qui donne :



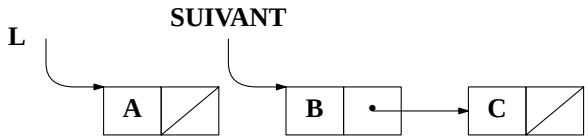
Etant donnée que la variable **X** *et* la variable **Y** sont liées a‘ la *meˆme* liste (c'est-a‘-dire : la C-valeur de **X** et la C-valeur de **Y** contiennent la meˆme adresse de la zone liste), la fonction **RPLACA** modifie les valeurs de tous les deux variables ! Ainsi, meˆme avec cette sauvegarde initiale de la valeur de **X**, rien n'est changé : on se trouve dans le meˆme cas que précédemment, c'est-a‘-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-a‘-dire : sans faire un seul appel a‘ la fonction **CONS**. Pour résoudre ce problème nous changerons les pointeurs de la liste originale, telle que les *meˆmes* doublets seront, après exécution de la fonction, chaînés a‘ 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 eˆtre après inversement de la liste et où **L'** est la valeur de **L** a‘ la fin :

L

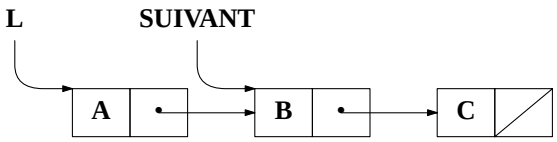


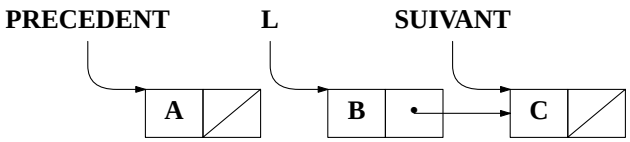
Le pas suivant consiste a‘ remplacer le pointeur du **CDR** du deuxième doublet par un pointeur vers ce premier doublet juste modifié. Mais, après cette opération du *remplacement* du **CDR** du premier doublet par **NIL**, nous n’avons plus aucun accès au deuxième doublet. Donc : avant de faire cette 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 :

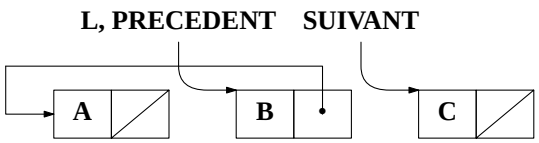


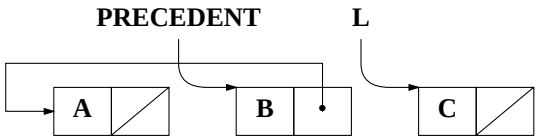
Si maintenant nous remplaçons le **CDR** du deuxième doublet par un pointeur vers la valeur de **L**, nous perdons a‘ nouveau la suite de la liste. Nous devons donc d’abord avancer le pointeur **SUIVANT**, mais sans perdre sa valeur actuelle, puisque, dans la suite nous devons établir un pointeur vers ce doublet pointé actuellement par **SUIVANT**.

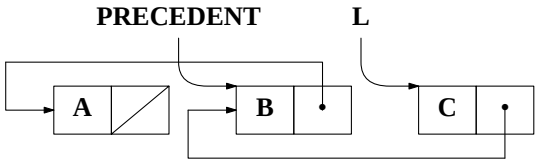
La solution qui s’impose sera donc d’introduire un pointeur supplémentaire, nommons-le **PRECEDENT**, et que **L** pointera a‘ tout instant sur la nouvelle teˆte de liste, **PRECEDENT** pointera vers le doublet qui précédait dans la liste originale l’élément pointé par **L**. Le pointeur **SUIVANT** 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 **FRERVERSE** :

```
(DE FREVERSE (L)
  (LET ((L L) (PRECEDENT))
       (IF L (SELF (CDR L) (RPLACD L PRECEDENT))
           PRECEDENT)))
```

Dans cette version nous n'avons pas besoin d'un pointeur **SUIVANT** explicite puisque la liaison de **L** au **CDR** de **L**, donc a‘ l'adresse du doublet suivant se fait en parallèle a‘ la destruction de ce meˆme champ **CDR** !

Voila‘ donc une fonction qui inverse une liste sans faire le moindre **CONS**, c'est-a‘-dire : une fonction qui n'utilise aucun doublet supplémentaire. **FREVERSE** sera donc sensiblement plus rapide que la fonction **REVERSE** que nous avons vu au chapitre 6.

Mais n'oublions jamais que **FREVERSE** est *destructive*, et qu'il faut prendre les meˆmes précautions qu'avec la fonction **FOO** ci-dessus. Cela signifie que si nous avons, par exemple, la fonction :

```
(DE 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 :

```
(DE LAST (L) (IF (ATOM (CDR L)) L (LAST (CDR L))))

(DE NCONC (LISTE1 LISTE2)
   (RPLACD (LAST LISTE1) LISTE2) LISTE1)
```

Nous pouvons construire une fonction **ATTACH** qui insère un nouvel élément en teˆte d'une liste, de manière telle que le premier doublet de la liste originale soit également le premier doublet de la liste résultat :

```
(DE 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 :

```
(DE 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 meˆme structure et si elles sont composées des meˆmes atomes dans le meˆme 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 *meˆme* atome. Pour prendre un autre exemple, chaque occurrence de l'atome **CONS** a‘ l'intérieur d'un programme se réfère précisément au meˆme 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 : effectivement, les deux atomes **A** dans l'expression ci-dessus sont des références a‘ un seul et unique objet.

Sachant que **EQ** se réduit a‘ une comparaison d'adresses, et sachant que des listes sont également accessibles a‘ 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))
```

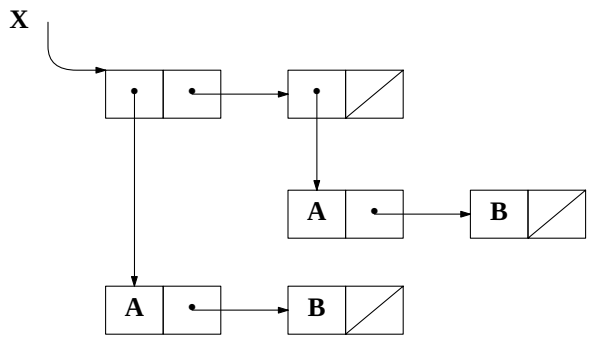
et la deuxième, **Y**, par l'appel :

```
(LET ((X '(A B))) (LIST X X))
```

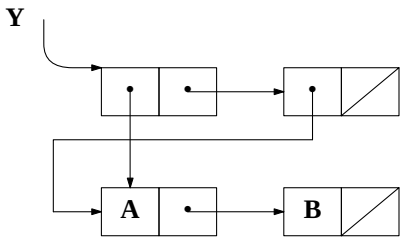
Regardez bien la différence entre ces deux listes ! Les deux listes s'écrivent de façon externe :

```
((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 meˆme liste : la fonction **EQ** livre donc **T**.

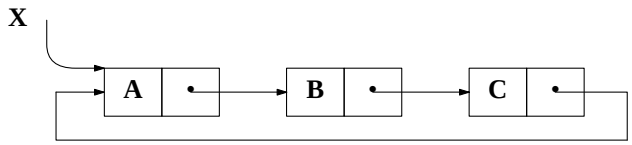
On dit que de telles listes sont *partagées*. Au niveau de l'impression standard, aucun moyen n'existe pour distinguer des listes partagées des listes non-partagées. Naturellement, des listes partagées sont toujours issues d'opérations sur des listes (c.a'.d.: elles ne peuvent pas eˆtre données tel quel 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 concatène deux listes : nous concaténons donc a' la liste **X**, c'est-a'-dire : **(A B C)**, la liste **X** elle-meˆme, 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 eˆtre 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** a‘ l'intérieur de cette liste **X**, l'appel

(MEMQ 'D X)

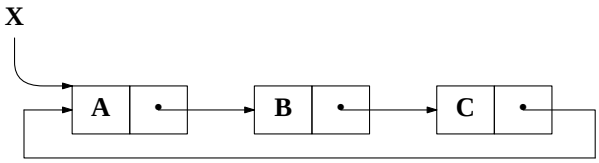
va boucler éternellement, puisque la fonction **MEMQ** compare le premier élément de la liste avec l'argument cherché, et si ce n'est pas le meˆme, elle recommence l'opération sur le **CDR** de la liste, ceci jusqu'a‘ ce qu'elle arrive soit a‘ une occurrence de l'élément cherché soit a‘ la fin de la 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 également.

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 '**EQ**ual' a‘ la liste entière et ainsi d'éviter de répeter la recherche a‘ l'infinie.

Ci-après la définition du prédicat C-MEMQ testant l'occurrence d'un élément a‘ l'intérieur d'une liste circulaire :

```
(DE C-MEMQ (OBJ L)
    (LET ((LL L)) (COND
        ((ATOM LL) (EQ LL OBJ))    ; le test d'égalité ;
        ((EQ L (CDR LL)) ())       ; le test d'arreˆt !! ;
        (T (OR (SELF (CAR LL)) (SELF (CDR LL)))))))
```

Si on appelle cette fonction avec l'atome **D** comme premier argument, et comme deuxième argument la liste circulaire :



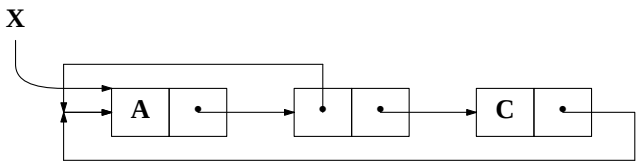
le résultat est **NIL**.

Naturellement, si le premier argument se trouve a‘ 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** :

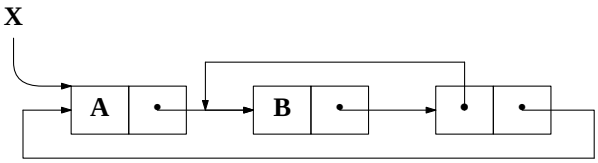


C-MEMQ, la fonction de recherche d'un élément dans une liste circulaire ne marche plus sur cette liste, car aucun test n'est effectué vérifiant la circularité sur le **CAR** de la liste.

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** :

```
(DE C-MEMQ (OBJ L)
   (IF (EQ (CAR L) OBJ) T
      (LET ((LL (CDR L))) (COND
            ((ATOM LL) (EQ LL OBJ))
            ((EQ L LL) ())
            (T (OR (SELF (CAR LL)) (SELF (CDR LL))))))))
```

Finalement, nous pouvons généraliser notre approche a‘ des listes a‘ circularités arbitraires, c'est-a‘-dire a‘ 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' a‘ 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 :[3]

```
(DE C-MEMQ (OBJ L)
   (LET ((L L) (AUX NIL)) (COND
         ((ATOM L) (EQ OBJ L))
         ((MEMQ L AUX) ())
         (T (OR (SELF (CAR L) (CONS L AUX))
                (SELF (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'a‘ maintenant, l'unique moyen que nous connaissons pour donner des valeurs a‘ des variables est la *liaison dynamique*, c'est-a‘-dire : la liaison des valeurs a‘ des variables pendant les appels de fonctions utilisateurs. Cette liaison dynamique se comporte de manière telle, qu'a‘ l'appel d'une fonction les variables prennent les nouvelles valeurs, déterminées par les arguments de l'appel, et qu'a‘ la sortie de la fonction les anciennes valeurs de ces variables soient restituées.

En LISP, il existe deux autres manières pour donner des valeurs a‘ des variables en utilisant les fonctions d'*affectation* **SETQ** et **SET**. La fonction **SETQ** prend - au minimum - deux arguments, le premier doit eˆtre 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 a‘ 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 valeurs de X
= 1
? (SETQ X (+ X 3))
= 4
? X                   ; examinons la nouvelle valeur de X
= 4
```

et voici une manière élégante d'échanger les valeurs numérique de **X** et **Y** sans passer par une variable intermédiare :

```
? (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 globales a‘ un ensemble de fonctions, elle permet de donner a‘ l'extérieur de toute fonction des valeurs a‘ des variables.

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**)** $\rightarrow$ *valeur(*val_n*)*
et 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 a' **SETQ**, mise a' 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 *indirection6 1 Tf 15 1 Tf 4.74j /F4 1 Tf 66F4 1 Tf0 TD ())Tj s nF6 1 K 4.74j /F 0 0 ple,*

```
(DF SETQ X
    (LET ((VARIABLE (CAR X)) (VALEUR (EVAL (CADR X))) (REST (CDDR X)))
        (SET VARIABLE VALEUR)
        (IF REST (EVAL (CONS 'SETQ REST)) VALEUR)))
```

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** a‘ deux arguments **ELE** et **LISTE**, qui enlève physiquement toutes les occurrences de l'élément **ELE** a‘ 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. 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** définit comme suit :

(NEXTL *variable***)** $\rightarrow$ **(CAR** *(valeur (variable))***)** et *variable* prend comme nouvelle valeur **(CDR** *variable***)**.

Avec cette fonction, la fonction **PREVERSE** peut eˆtre redéfinie comme suit :

```
(DF PREVERSE (L RES)
    (LET ((LISTE (EVAL (CAR L))))
        (IF (NULL LISTE) (SET (CAR L) RES)
            (SETQ RES (CONS (NEXTL LISTE) RES))
            (SELF LISTE))))
```

Definissez alors cette fonction **NEXTL**.

6. Definissez 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**.

16. 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 tre‘s souvent dans un programme le quatrie‘me 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 :

(DE 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 proble‘me avec cette manie‘re 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, liai-salteusn dsi **uuntypaetlde** cauise d (cettedichotomnielen‘re la lisibilitée plue)]TJ T* .0641 Tw [(remsn
so- fonctions :

(DM *nom variable corps***)** → *nom*
définie une fonction de type *macro* où la variable *variable* est liée à l'appel entier.

Ainsi, si nous définissons en VLISP une macro-fonction **FOO** comme :

(DM FOO (X) . . .)

cette même macro-fonction se définit en LE_LISP comme :

(DM FOO X . . .)

Ce sont donc des définitions où la liste de variables est réduite à un atome. Par un mécanisme similaire à celui de la liaison des NEXPRs cet atome est lié à la forme de l'appel de la macro.

Allons pas-à-pas et prenons d'abord un exemple tre‘s simple. Voici la macro-fonction **EX1** :

```
(DM EX1 (L)
    (PRINT "L =" L)
    '(1+  99))
```

Cette macro ne fait rien d'autre qu'éditer son argument et de ramener dans un premier temps la liste **(1+ 99)**. Cette liste est, dans un deuxie‘me temps, réévaluée, ce qui donne pour tout appel de **EX1** le résultat **100**. Essayons :

```
? (EX1)        ; l'appel de la macro-fonction
  L = (EX1)    ; l'impression de l'argument (qui est bien l'appel entier !)
= 100          ; le résultat de l'évaluation de (1+ 99)
```

Mais revenons vers notre fonction **4-IEME** et définissons une macro résolvant le proble‘me. Voici la premie‘re version :

(DM 4-IEME (L) (LIST 'CADR (LIST 'CDDR (CADR L))))

ou, en LE_LISP :

(DM 4-IEME L (LIST 'CADR (LIST 'CDDR (CADR L))))

Notons que le corps de cette macro construit une expression à évaluer, laquelle exprime le calcul du **CADR** du **CDDR** de l'argument donné. Evidemment, en utilisant le macro-caracte‘re *back-quote* (‘), nous pouvons écrire le corps plus simplement comme :

‘(CADR (CDDR ,(CADR L)))

Si nous appelons cette macro comme suit :

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

le résultat de la premie‘re 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**, 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 :

```
(DM MCONS (L)
    (IF (NULL (CDDR L)) (CADR L)
        '(CONS ,(CADR L) (MCONS ,@(CDDR L)))))
```

Voici une trace d'un appel de **MCONS** :

```
? (MCONS 1 2 3 4 5 NIL)                    ; l'appel initial ;
  ---> MCONS (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 (MCONS 2 3 4 5 NIL)           ; le premier appel récursif ;
  <--- MCONS (CONS 2 (MCONS 3 4 5 NIL))    ; son résultat
  ---> MCONS (MCONS 3 4 5 NIL)
  <--- MCONS (CONS 3 (MCONS 4 5 NIL))
  ---> MCONS (MCONS 4 5 NIL)
  <--- MCONS (CONS 4 (MCONS 5 NIL))
  ---> MCONS (MCONS 5 NIL)
  <--- MCONS (CONS 5 (MCONS NIL))
  ---> MCONS (MCONS NIL)
  <--- MCONS NIL
= (1 2 3 4 5)
```

A chaque appel, la variable est liée à cet appel en entier et son évaluation s'éxécute en deux temps : chaque appel récursif est effectué *apre's* la sortie de l'appel précédent. Ce n'est pas réellement une exécution récursive !

Toutefois, bien que nous ayons introduit les macro-fonctions en argumentant sur les temps d'exécution, nous ne pouvons pas réellement dire 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. Revenons alors à notre définition de **DM** : nous disions que la variable d'une macro-fonction est liée à *l'appel même* de cette macro-fonction. C'est là le point important ! Ceci nous permet de modifier physiquement l'appel de cette macro, à la premie're évaluation, par la suite d'instruction qui définit le corps de la macro. Ainsi, les appels de macro-fonctions se détruisent eux-mêmes, assurant par là l'élimination de ces appels.

Revenons vers notre macro **4-IEME** et modifions-la de manie're à remplacer son appel par le corps de la macro même :

```
(DM 4-IEME (L)
    (RPLACB L '(CADR (CDDR ,(CADR L)))))
```

La fonction **RPLACB** est évidemment celle que nous avons définie à la fin du chapitre précédent : la fonction **DISPLACE** de LE_LISP.

Afin de voir clairement l'effet de cette macro, construisons une fonction **FOO** qui calcule la somme du premier et du quatrie'me élément de sa liste argument. Ce n'est pas difficile :

```
(DE FOO (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 **4**. Mais, si nous regardons maintenant la fonction, elle a l'allure suivante :

```
(DE FOO (LISTE)
    (+ (CAR LISTE) (CADR (CDDR LISTE))))
```

Tous les appels ultérieurs de la fonction **FOO** n'impliquerons 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.

Des telles macro-fonctions s'appellent des macros *destructives*, puisqu'elles détruisent la forme de leur appel.

Construisons alors la macro-fonction **DMD** qui nous permet de définir de telles macro-fonctions *sans* écrire chaque fois l'appel explicite de **RPLACB**. La voici :

```
(DM DMD (L)
    (LET ((NOM (CADR L))
          (VARIABLE (CADDR L))
          (CORPS (CDR (CDDR L))))
        '(DM ,NOM (,VARIABLE)
             (RPLACB ,VARIABLE ,@CORPS))))
```

La macro-fonction **DMD**, qui est une fonction standard en LE_LISP, construit et évalue un appel de la fonction **DM**. Avec cette fonction, la définition de la fonction **4-IEME** ci-dessus peut s'écrire :

```
(DMD 4-IEME L
     '(CADR (CDDR ,(CADR L))))
```

La fonction **NEXTL** que nous avons définie dans les exercices du chapitre précédent, peut maintenant être définie en tant que macro-fonction comme suit :

```
(DMD NEXTL L
     '(LET ((VAL ,(CADR L)))
        (SETQ ,(CADR L) (CDR VAL)) (CAR VAL)))
```

Avec cette définition de **NEXTL**, la fonction **PREVERSE** du fin du chapitre précédant devient, après une exécution :

```
(DF PREVERSE (L RES)
    (LET ((LISTE (EVAL (CAR L))))
        (IF (NULL LISTE) (SET (CAR L) RES)
            (SETQ RES
                (CONS
                    (LET ((VAL LISTE))
                        (SETQ LISTE (CDR VAL))
                        (CAR VAL))
                RES))
            (SELF LISTE))))
```

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.

16.1. EXERCICES

1. Définissez les macro-fonctions **INCR** et **DECR**

17. LES DIVERES FORMES DE REPETITION

L'unique manière de faire des répétitions que nous connaissions jusqu'à maintenant est la récursivité, i.e.: 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.

17.1. 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 (SETQ M (CONS (NEXTL L) M)) *))
(RPLACA (LAST E) E)
```

Nous avons ici construit une liste **E** circulaire de la forme :

```
(IF (NULL L) M (SETQ M (CONS (NEXTL L) M))
    (IF (NULL L) M (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.

Un autre exemple pour exprimer la structure de contrôle dans la structure de 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

[1] Cet exemple est duˆ à Daniel Goossens

[2] La fonction **NCONC1** peut être définie par :

```
(DM NCONC1 (L) (RPLACB L '(NCONC ,(CADR L) (LIST ,(CADDR L)))))
```

un appel de **(EVAL REVERSE)**.

La différence avec la fonction **E** consiste dans le fait qu'en **E** nous avons eu une liste circulaire, ici, en **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 de construire des programmes de cette manière, elle n'est exposée ici qu'à titre de curiosité (et, également, à titre d'exposition de la puissance de la représentation des programmes sous forme de listes !).

17.2. LES ARGUMENTS FONCTIONNELS

Mise 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)**. Même les appels de fonctions à travers la fonction **SELF** peuvent être considérés comme des appels explicites, car il ne font rien d'autre qu'appeler récursivement la fonction à l'intérieur de laquelle on se trouve. 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 une manière à travers l'utilisation des fonctions **EVAL** et **APPLY**. Dans le reste de ce chapitre nous examinerons d'autres manières de calculer les fonctions appelantes.

Comme nous l'avons dit au début de ce livre, LISP existe dans une multitude de dialectes. Normalement, les différences d'un dialecte à l'autre sont mineures, sauf en ce qui concerne la manière de traiter l'évaluation des appels de fonctions calculées. Là, nous pouvons distinguer deux types de dialectes LISP : les interprètes LISP s'adaptant aux normes de Common-LISP, dont LE_LISP est un excellent exemple, et les autres, comme, par exemple, VLISP. Dans la suite nous donnons pour chaque programme deux versions : une dans le style de LE_LISP et une dans le style de VLISP.

Dans la définition d'une *forme* au début de ce livre, tout ce que nous avons dit est que l'évaluateur LISP considère le premier élément d'une liste à évaluer comme la fonction, et le reste de la liste comme les arguments de cette fonction. En LE_LISP une erreur *fonction indéfinie* se produit si l'évaluateur, ne connait pas la fonction du début de la liste, i.e.: si elle n'est pas définie. En VLISP l'évaluateur cherche alors à évaluer le **CAR** de cette liste, pour regarder si *la valeur* du **CAR** est une fonction, ainsi de suite, jusqu'à ce que LISP trouve une erreur où rencontre une fonction. Ce sera alors cette fonction qui s'appliquera aux arguments de la liste originale.

Par exemple, si l'on tape :

(SETQ F '1+)

suivi de l'appel :

(F 3)

le résultat de l'évaluation sera **4** : VLISP a évalué l'atome **F**, puisque **F** n'est pas le nom d'une fonction, et a pris le résultat de l'évaluation de **F** comme fonction appliquée à l'argument **3**.

Pour pouvoir faire la même chose en LE_LISP, on doit explicitement appeler la fonction **FUNCALL**, une version spécial de **APPLY**. En LE_LISP, si la variable **F** est liée à **1+**, l'appel

(FUNCALL F 3)

applique la fonction **1+** à l'argument 3, et livre donc également le nombre **4**.

ensuite en LE_LISP :

```
(DE MAPCAR (L F)
    (AND L
        (CONS (FUNCALL F (CAR L)) (MAPCAR (CDR L) F))))
```

A l'aide de cette fonction **MAPCAR**,[3] les deux fonctions ci-dessus peuvent être écrites simplement comme :

```
(DE PLUS-UN (L) (MAPCAR L '1+))

(DE LIST-EACH (L) (MAPCAR L 'LIST))
```

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.

Revenons à présent aux divers types d'appel de fonctions, nous en connaissons deux : les appels explicites et les appels par évaluation d'une variable en position fonctionnelle. 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 :

VLISP

((CAR '(1+ 1-)) 3)	→	**4**
((LET ((F 'CAR)) F) '(A B C))	→	**A**
((TWICE 'CAR '((FACTORIELLE)(+)(-))) 6)	→	**720**
((LAMBDA (X) (LIST X X)) 2)	→	**(2 2)**

LE_LISP

(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 s'écrit de manière identique dans les deux types de LISP. Examinons le : il utilise une λ-expression. Une λ-expression est une fonction sans nom dont l'évaluation livre cette λ-expression elle-même. Il n'est donc pas nécessaire de 'quoter' une λ-expression : elle est auto-quotée.

```
(LAMBDA (X) (LIST X X))
```

est une fonction - sans nom - à un argument, nommé **X**, et avec le corps **(LIST X X)**. Son appel s'effectue simplement en mettant cette λ-expression en position fonctionnelle. C'est très similaire à notre fonction **LET** qui combine la définition d'une λ-expression avec son appel. D'ailleurs, **LET** peut être définie en termes de **LAMBDA** comme suit :

```
(DMD LET L
     '((LAMBDA ,(MAPCAR (CADR L) 'CAR) ,@(CDDR L))
       ,@(MAPCAR (CADR L) 'CADR)))
```

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** :

---

[3] Les arguments de la fonction **MAPCAR** *standard* de LE_LISP sont dans l'ordre inverse : d'abord la fonction, ensuite l'argument.

```
(DE ENL-A (L)
    (MAPCAR L '(LAMBDA (X) (DELQ 'A X))))
```

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 :

```
(DE ENL-A-SANS-REPETITIONS (L) (ENL-A-AUX (ENL-A L)))

(DE 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 :

```
(DE ENL-A-SANS-REPETITIONS (L)
   (LET ((AUX  ()))
     (MAPC (ENL-A L)
        '(LAMBDA (X)
           (IF (MEMBER X AUX) ()
                (SETQ AUX (CONS X AUX)))))
     AUX))
```

La fonction **MAPC** est une simplification de **MAPCAR**, définie en VLISP, par :

```
(DE MAPC (L F)
   (IF (NULL L) () (F (CAR L)) (MAPC (CDR L) F)))
```

et en LE_LISP :

```
(DE MAPC (L F)
   (IF (NULL L) () (FUNCALL F (CAR L)) (MAPC (CDR L) F)))
```

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 du style **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 :

```
(DE PERMUT (L) (PERMUT1 L NIL))

(DE PERMUT1 (L RES)
  (LET ((X NIL))
    (MAPC L
        '(LAMBDA (ELE)
            (SETQ X (DELQ ELE L))
            (IF X (PERMUT1 X (CONS ELE RES))
                (PRINT (CONS ELE RES)))))))
```

Un problème similaire à celui des permutations est le problème des 8 reines. Il s'agit de placer 8 reines sur un échiquer, de manière qu'aucune reine ne puisse prendre une autre.

Rappelons brièvement : un échiquier est un tableau de 64 cases répartis en 8 lignes et 8 colonnes. Une reine est une pièce du jeu dotée de la prérogative de pouvoir prendre toute pièce qui se trouve sur la même ligne, sur la même colonne, ou sur la même diagonale qu'elle.

Il est clair que toute solution comporte 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 :

R							
				R			
							R
					R		
		R					
						R	
	R						
			R				

Avec cette représentation, le problème revient à trouver toutes les permutations des nombres 1 à 8, satisfaisant quelques contraintes. Voici alors le programme résolvant le problème des 8 reines :[4]

[4] Une très bonne description de ce problème se trouve dans l'excellent livre *la construction de programmes structurés* de Jaques Arsac.

```
(DE REINE (LISTE) (REINE-AUX LISTE 1 NIL NIL NIL NIL))

(DE REINE-AUX (LISTE COLONNE RESUL DIAG1 DIAG2 AUX3)
   (MAPC L
      (LAMBDA (X)
         (SETQ AUX3 (DELQ X LISTE))
         (IF (AND (NULL (MEMQ (- X COLONNE) DIAG1))
                  (NULL (MEMQ (+ X COLONNE) DIAG2)))
            (IF (NULL AUX3) (PRINT (CONS X RESUL))
               (REINE-AUX AUX3
                     (1+  COLONNE)
                     (CONS X RESUL)
                     (CONS (- X COLONNE) DIAG1)
                     (CONS (+ X COLONNE) DIAG2)))))))
```

Un appel se fera alors par

```
(REINE '(1 2 3 4 5 6 7 8))
```

La fameuse fonction **LIT** est similaire à **MAPCAR**. Là voici d'abord en VLISP :

```
(DE LIT (LISTE FIN FONCTION)
   (IF LISTE
      (FONCTION (NEXTL LISTE) (LIT LISTE FIN FONCTION))
      FIN))
```

et ensuite en LE_LISP :

```
(DE 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 :

```
(DE APPEND (L1 L2) (LIT L1 L2 'CONS))
```

et la définition de la fonction **MAPCAR** devient en VLISP :

```
(DE MAPCAR (L F) (LIT L NIL (LAMBDA (X Y) (CONS (F X) Y))))
```

et en LE_LISP :

```
(DE 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 :

```
(DE MUL (L) (LIT L 1 '*))
```

17.3. QUELQUES ITERATEURS SUPPLEMENTAIRES

En plus des possibilités de répétition par la récursivité, LISP dispose de quelques *itérateurs* : les fonctions

WHILE, **UNTIL** et **DO**.

La fonction **WHILE** est définie comme suit :

(WHILE *test corps***)** → **NIL**
corps est répétitivement exécuté tant que l'évaluation de *test* donne une valeur différente de **NIL**.

Voici la fonction **REVERSE** écrit en utilisant **WHILE** :

```
(DE REVERSE (L)
   (LET ((RES NIL))
      (WHILE L (SETQ RES (CONS (NEXTL L) RES)))
      RES))
```

et la fonction **APPEND** peut s'écrire :

```
(DE 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**
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 :

```
(DE REVERSE (L)
   (LET ((RES NIL))
      (UNTIL (NULL L) (SETQ RES (CONS (NEXTL L) RES)))
      RES))
```

Naturellement, si la fonction **WHILE** ou **UNTIL** n'existe pas dans votre LISP, vous pouvez les définir comme des macros. Voici, par exemple, une définition de **WHILE** :

```
(DMD WHILE CALL
      '(LET ()
         (IF (NULL ,(CADR CALL)) ()
            ,@(CDDR CALL) (SELF)))))
```

Il est possible d'utiliser à l'intérieur des définitions de macro-fonctions la technique exposée au début du chapitre, i.e.: la répétition par évaluation séquentielle d'une liste circulaire. Ce qui donne pour cette même fonction **WHILE** la nouvelle définition ci-après :

```
(DMD WHILE CALL
      '(IF (NULL ,(CADR CALL)) ()
         ,@(CDDR CALL) ,CALL))
```

MACLISP, un dialecte de LISP devéloppé au M.I.T., a introduit l'itérateur **DO** qui est défini comme suit :

(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, indiquant la suite des instructions à évaluer à chaque itération.

Prenons comme exemple la fonction **COMPTE** qui imprime tous les nombres de 0 à **N** :

```
(DE COMPTE (N)
    (DO ((I 0 (1+ I)))
        ((= I N) (TERPRI) 'FINI)
        (PRINT I)))
```

La fonction **REVERSE** s'écrit, en utilisant **DO**, comme ci-dessous :

```
(DE REVERSE (L)
   (DO ((L L (CDR L)) (RES NIL (CONS (CAR L) RES)))
       ((NULL L) RES)))
```

Dans cette forme, la partie 'corps' du **DO** est vide : tout le travail se fait dans l'itération.

La définition de la macro **DO** ci-après utilise la fonction **PROGN**, une fonction standard qui évalue séquentiellement chacun de ses arguments et ramène en valeur le résultat de l'évaluation du dernier argument.

```
(DMD DO CALL
    `((LAMBDA ,(MAPCAR (CADR CALL) 'CAR)
        (IF ,(CAAR (CDDR CALL)) (PROGN ,@(CDAR (CDDR CALL)))
            ,@(CDDDR CALL)
            (SELF ,@(MAPCAR (CADR CALL) 'CADDR))))
      ,@(MAPCAR (CADR CALL) 'CADR)))
```

La fonction **REVERSE** définie en utilisant la macro **DO** se transforme donc, après une exécution, en :

```
(DE REVERSE (L)
   ((LAMBDA (L RES)
       (IF (NULL L)
           (PROGN RES)
           (SELF (CDR L) (CONS (CAR L) RES))))
     L NIL))
```

C'est quasiment la même fonction que celle que nous avons définie dans le chapitre 6.

17.4. 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**, aussi bien en utilisant la fonction **SELF** qu'avec une liste circulaire.
3. Ecrivez la fonction **FACTORIELLE** en utilisant l'itérateur **DO**
4. Ecrivez la macro-fonction **DO** en faisant l'itération avec la fonction **WHILE**.

18. 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 **$** indique un élément quelconque,
2. le signe **&** indique une séquence, éventuellement vide, d'éléments quelconques,
3. le signe **%** indique 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 de 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 non seulement intéressé 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 que les opérateurs de filtrage puissent 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 non seulement indiquer qu'une certaine donnée correspond au filtre, mais d'également indiquer 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. Deux possibilités existent : soit en le faisant comme LISP, c'est-à-dire en changeant les C-valeurs des variables (avec la fonction **SET** ou **SETQ**), soit en retournant une liste contenant textuellement les noms des variables suivis de leurs valeurs, donc une liste de la forme :

(($variable_1$ **.** $valeur_1$**) . . . (**$variable_n$ **.** $valeur_n$**))**

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** au début de la liste est l'indicateur de réussite du filtrage, et le deuxième élément de la liste est la A-liste exprimant 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 en **CAR** l'indicateur de réussite du filtrage **T** et contenant en **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 qu'un 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)))
Une 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 commençant et se terminant 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.

• (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. Rappelons le :

```
(DE MATCH (FILTRE DONNEE) (COND
      ((ATOM FILTRE) (EQ FILTRE DONNEE))
```

[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 *A-liste* **(LAMBDA (X) (SET (CAR X) (CDR X))))**

```
((ATOM DONNEE) NIL)
((EQ (CAR FILTRE) '$)
     (MATCH (CDR FILTRE) (CDR DONNEE)))
((EQ (CAR FILTRE) '&) (COND
     ((MATCH (CDR FILTRE) DONNEE))
     (DONNEE (MATCH FILTRE (CDR DONNEE)))))
((AND (CONSP (CAR FILTRE))(EQ (CAAR FILTRE) '%))
     (LET ((AUX (CDR (CAR FILTRE)))) (COND
          ((NULL AUX) NIL)
          ((MATCH (CAR AUX) (CAR DONNEE))
               (MATCH (CDR FILTRE) (CDR DONNEE)))
          (T (SELF (CDR AUX))))))
((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 taˆche sera divisée en deux : le travail proprement dit sera réalisé par une fonction auxiliaire, la fonction **MATCH** se contentant de lancer cette fonction auxiliaire et d'éditer le résultat.

```
(DE MATCH (FILTRE DONNEE)
     (LET ((ALISTE NIL))
          (IF (MATCH-AUX FILTRE DONNEE)
               (LIST T ALISTE) NIL)))
```

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 (**$** ou **&**) du nom de la variable qui suit. Si nous utilisons la fonction **EXPLODE** pour tester le premier caractère d'un atome, nous devons le faire avec *tous* les atomes contenus dans le filtre. C'est trop couˆteux en temps. Définissons donc les caractères **$**, **&** 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.

```
(DMC "$" () (CONS "$" (READ)))

(DMC "&" () (CONS "&" (READ)))

(DMC "%" () (CONS "%" (READ)))
```

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'a 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 (EQ (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 point d'exclamation 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 point d'exclamation 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 **ASSQ** qui cherche sur une A-liste si une variable donnée est déjà liée ou pas. Si elle est liée, **ASSQ** ramène le doublet **(***variable* **.** *valeur***)** :

```
(DE ASSQ (VARIABLE ALISTE) (COND
        ((NULL ALISTE) ())
        ((EQ (CAAR ALISTE) VARIABLE) (CAR ALISTE))
        (T (ASSQ VARIABLE (CDR ALISTE)))))
```

Cette fonction existe en tant que 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 ;
    ((ASSQ (CDAR FILTRE) ALISTE)
        (MATCH-AUX
            (CONS (CDR (ASSQ (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)) . . .

et récrivons l'avant dernière clause :

```
((SETQ AUX (ASSQ (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) "%")
    (LET ((AUX (CDAR FILTRE)) (ALIST2 ALIST)) (COND
          ((NULL AUX) (SETQ ALIST ALIST2) NIL)
          ((MATCH-AUX (CONS (NEXTL AUX) (CDR FILTRE)) DONNEE) T)
          (T (SELF (CDR AUX) ALIST2)))))
```

Vient ensuite le cas difficile des variables de segment, i.e.: 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 c'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.

Ok. Allons-y et écrivons cette clause difficile :

```
((EQUAL (CAAR FILTRE) "&") (COND
   ; exite-il une valeur ;
   ((SETQ AUX (ASSQ (CDAR FILTRE) ALISTE))
      ; oui, comparer alors la valeur ;
      (MATCH-AUX (APPEND (CDR AUX) (CDR FILTRE)) DONNEE))
   ; c'est la fin du filtre ? ;
   ((NULL (CDR FILTRE))
      ; succès ;
      (if (EQ (CDAR FILTRE) '-) ()
         (SETQ ALISTE (CONS (CONS (CDAR FILTRE) DONNEE) ALISTE)))
       T)
   (T (IF (SETQ AUX (EQ (CDAR FILTRE) '-)) ()
         (SETQ ALISTE (CONS (LIST (CDAR FILTRE)) ALISTE)))
      (LET ((ALIST2 ALISTE))
           (IF DONNEE
              ; faisons les comparaisons successives ;
              (IF (MATCH-AUX (CDR FILTRE) DONNEE) T
                 (SETQ ALISTE ALIST2)
                 (IF AUX (NEXTL DONNEE)
                       ; la liaison de la variable s'agrandit ;
                       (NCONC (CAR ALISTE) (LIST (NEXTL DONNEE))))
                 (SELF ALISTE))
              ; échec totale : restaurons la A-liste ;
              (IF AUX (NEXTL ALIST))
              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 des données.

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 petit 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", ou "rotor".[2]

Voici le début du programme :

[2] 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.

(DF PALINDROME (MOT) (PAL (EXPLODE (CAR MOT))))

Evidemment, ce programme ne fait qu'éclater le mot dans une suite 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.

Pour la comparaison du premier et du dernier caractère le filtre :

($X &MILIEU $X)

semble parfait : en plus, ce filtre récupère la suite de caractères qui reste si on enlève le caractère du début et de la fin :

```
(DE 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 de appeler récursivement **PAL** avec la valeur de **MILIEU**.

Nous pouvons le faire avec l'appel :

(PAL (CDR (ASSQ '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 :

```
(DM LETALL (CALL)
   (LET ((ALIST (EVAL (CADR CALL))))
      (RPLACB CALL
           '((LAMBDA ,(MAPCAR ALIST 'CAR) ,@(CDDR CALL))
             ,@(MAPCAR ALIST (LAMBDA (X) (LIST QUOTE (CDR X))))))))
```

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 : si la variable **L** contient la liste :

((X A B C) (Y . 1))

l'appel :

(LETALL L (PRINT X Y) (CONS Y X))

donne d'abord l'impression **(A B C) 1**, et livre ensuite la liste **(1 A B C)** et nous permet de compléter notre fonction **PAL** en insérant après **(IF (CAR AUX)** la commande :

(LETALL (CADR AUX) (PAL MILIEU))

Le filtrage, constituant de base de la plupart de 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'enoncé, et dès qu'un des filtres correspond, 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)
($-)	'(c est un peu court)
(&- je ne suis pas &l)	'(que vous fait pensez que vous n'êtes pas &l)
(&- suis &l)	'(pourquoi pensez-vous que vous êtes &l)
(&- je ne &-)	'(vous en êtes suˆr)
(&- non &-)	'(vous êtes très négatif)
(&- vous &-)	'(concentrez vous sur votre problème, svp)
(&- moi &-)	'(racontez-moi plus sur votre famille)
(&- je &-)	'(comment cela se manifeste-t-il)
(&- j ai &-)	'(depuis combien de temps)
(&- combien &-)	'(ce n est point la quantité qui compte)
($- $- $-)	'(hmmm)
(&-)	'(je ne vous suis pas très bien)

Graˆce à ce petit ensemble de règles, nous pouvons déjà faire des dialogues comme celui-ci (les impressions du programme sont en caractères **gras**, celle 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 suˆr)
(oui)
(c est un peu court)
(je ne suis pas court)
(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, afin de le rendre vraiment stimulant, d'acroître le nombre de couples <filtre, action> et d'améliorer le programme qui les utilise, dont voici un (à vous de l'implémenter, de l'améliorer et de le tester) :

```
(DE ELIZA ()
     (PRINT "bonjour, quel est votre problème ?")
     (WHILE (REPOND (ANALYSE (READ))))
     "au revoir")

(DE ANALYSE (PROBLEME)
    (LET ((PAT-REP LIST-PATTERN-REPONSE) (MATCH))
       (IF (CAR (SETQ MATCH (MATCH (CAAR PAT-REP) PROBLEME)))
            (REFLECHIT (CADR MATCH) (CDAR PAT-REP))
            (SELF (CDR PAT-REP) ()))))

(DE REFLECHIT (ENV MODELE)
      (EVAL '(LETALL ,ENV ,MODELE)))

(DE REPOND (REPONSE)
  (AND REPONSE (PRINT "PSY : " REPONSE))
```

La seule chose qui manque encore est de mettre la liste des filtres et réponses dans la variable globale **LIST-PATTERN-REPONSE** sous la forme :

```
(SETQ LIST-PATTERN-REPONSE
      '(  ((AU REVOIR) . ())
          ((STOP) . ())
          ((NON) . '(VOUS ETES BIEN NEGATIF))
          ((&- LISP &-) . '(ALORS VOUS SAVEZ PROGRAMMER))

          ; 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 être 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 (LE (LENGTH Z) 1)) &(Y (EQUAL X (REVERSE Y))))
```

Introduire cette modification supplémentaire n'est pas difficile : 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) ())
      ((EQ (CDAR FILTRE) '-)
            (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))
      ; regardons si la variable a une liaison ;
      ((SETQ AUX (ASSQ (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 (NULL (CDDAR FILTRE))
                    (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE))))
                (MATCH-AUX (CDR FILTRE) (CDR DONNEE))
                (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) :

```
(DE VAR (FILTRE)
      (IF (ATOM (CDR FILTRE)) (CDR FILTRE) (CADR FILTRE)))
```

Et la nouvelle clause pour la variable segment devient :

```
((EQUAL (CAAR FILTRE) "&") (COND
   ; existe-t-il une valeur ;
   ((SETQ AUX (ASSQ (CDAR FILTRE) ALISTE))
      ; oui, comparer alors la valeur ;
      (MATCH-AUX (APPEND (CDR AUX) (CDR FILTRE)) DONNEE))
   ; c'est la fin du filtre ? ;
   ((NULL (CDR FILTRE))
      ; succès ;
      (IF (EQ (CDAR FILTRE) '-) ()
          (SETQ ALISTE (CONS (CONS (VAR (CDAR FILTRE)) DONNEE) ALISTE)))
          (IF (OR (NULL (CDDAR FILTRE))
                  (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE)))) T
              (NEXTL ALIST) NIL))
   (T (IF (SETQ AUX (EQ (CDAR FILTRE) '-)) ()
          (SETQ ALISTE (CONS (LIST (VAR (CDAR FILTRE))) ALISTE)))
      (LET ((ALIST2 ALISTE))
           (IF DONNEE
               ; testons les contraintes ;
               (PROGN (IF (OR (NULL (CDDAR FILTRE))
                              (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE))))
                          ; faisons les comparaisons successives ;
                          (SETQ AUX1 (MATCH-AUX (CDR FILTRE) DONNEE)))
                      (IF AUX1 T
                          (SETQ ALISTE ALIST2)
                          (IF AUX (NEXTL DONNEE)
                              ; la liaison de la variable s'agrandit ;
                              (NCONC (CAR ALISTE) (LIST (NEXTL DONNEE))))
                          (SELF ALISTE)))
               ; échec total : restaurons la A-liste ;
               (IF AUX (NEXTL ALIST))
               NIL)))))
```

Avec cette petite modification nous pouvons comparer maintenant des structures avec des

contraintes sur les variables du filtre. Un ensemble de tels nouveaux filtres est présenté ci-dessous :

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 Y))))	**(1 2 A B 2 1)**	**(T ((Y 2 1) (X 1 2)))**
(&X &(Y (EQUAL Y (REVERSE Y))))	**(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 ramène un échec. Pourquoi ? Regardez : le problème dans cette fonction 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 liasons 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 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 réspectifs, qu'elle est incorrecte.

Manquant de structures de contrôle plus sophistiquées[3], 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 >>)

et de cette façon, la dernière clause de notre fonction de filtrage, et donc la clause 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 :

(DE MATCH (FILTRE DONNEE)

[3] Nous examinerons de telles structures de contrôle dans un deuxième livre sur LISP.

```
  (LET ((ALISTE NIL))
    (IF (MATCH-AUX (LINEAR FILTRE) (LINEAR DONNEE))
        (LIST T ALIST) NIL)))

(DE MATCH-AUX (FILTRE DONNEE)
  (LET ((AUX NIL) (ALIST2 NIL))
  (COND
    ((NULL FILTRE)                  ; test de fin de recursion
     (NULL DONNEE))
    ((AND DONNEE (ATOM DONNEE))     ; donnée atomique
     ())                            ; échec
    ((ATOM (CAR FILTRE))            ; constante
     (AND (EQ (CAR FILTRE) (CAR DONNEE))     ; si égalité
          (MATCH-AUX (CDR FILTRE) (CDR DONNEE))))  ; on continue, sinon échec
((EQUAL (CAAR FILTRE) "%")          ; choix multiple
      (LET ((AUX (CDAR FILTRE)) (ALIST2 ALIST)) (COND
            ((NULL AUX) (SETQ ALIST ALIST2) NIL)
            ((MATCH-AUX (CONS (NEXTL AUX) (CDR FILTRE)) DONNEE) T)
            (T (SELF (CDR AUX) ALIST2)))))
((EQUAL (CAAR FILTRE) "$") (COND    ; variable élément
       ((NULL DONNEE) ())           ; échec
       ((EQ (CDAR FILTRE) '-) (NEXTL-DONNEE)  ; pas de aliste
            (MATCH-AUX (CDR FILTRE) DONNEE))
       ; regardons si la variable a une liaison ;
       ((SETQ AUX (ASSQ (VAR (CAR FILTRE)) ALIST))  ; le doublet
            (MATCH-AUX
              (IF (ATOM (CDR AUX))  ; à cause de la linéarisation
                  (CONS (CDR AUX) (CDR FILTRE))
                  (APPEND (LINEAR (LIST (CDR AUX))) (CDR FILTRE)))
              DONNEE))
       ; il n'y a pas de liaison, faut créer une nouvelle liaison ;
       (T (SETQ ALIST (CONS (CONS (VAR (CAR FILTRE)) (NEXTL-DONNEE)) ALIST))
          (IF   ; y a-t-il des contraintes
            (AND (CONSP (CDAR FILTRE)) (CDDAR FILTRE))
                 ; alors faut les évaluer
            (IF (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE)))
                (MATCH-AUX (CDR FILTRE) DONNEE)  ; et continuer
                (NEXTL ALIST) NIL)       ; ça marche pas
            (MATCH-AUX (CDR FILTRE) DONNEE)))))  ; pas de contraintes
((EQUAL (CAAR FILTRE) "&") (COND    ; une variable segment
   ; existe-t-il une valeur ;
   ((SETQ AUX (ASSQ (VAR (CAR FILTRE)) ALIST))  ; le couple de la alist
      ; oui, comparer alors la valeur ;
      (MATCH-AUX (APPEND (LINEAR (CDR AUX)) (CDR FILTRE)) DONNEE))
   ; c'est la fin du filtre ? ;
   ((NULL (CDR FILTRE))       ; c'est la fin de la donnée
      ; alors succès ;
      (IF (EQ (CDAR FILTRE) '-) ()  ; rien à faire si anonyme
          ; sinon faut créer une liaison
          (SETQ ALIST (CONS
                        (CONS (VAR (CAR FILTRE)) (DELINEAR-DONNEE))
                        ALIST)))
          (IF          ; y a-t-il des contraintes ?
```

```
            (AND (CONSP (CDAR FILTRE))(CDDAR FILTRE))
            ; si oui faut les évaluer
            (IF (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE))) T
                (NEXTL ALIST) NIL) ; échec : contrainte non satisfait
            T))  ; pas de contraintes : tout va bien
    (T (IF (SETQ AUX (EQ (CDAR FILTRE) '-)) ()  ; variable anonyme
           (SETQ ALIST (CONS (LIST (VAR (CAR FILTRE))) ALIST)))
       (LET ((ALIST2 ALIST)) ; bouclons jusqu'au match du reste
            (IF DONNEE    ; y a encore des données
              ; testons les contraintes ;
              (PROGN
                (IF (AND (CONSP (CDAR FILTRE))(CDDAR FILTRE))
                  (IF (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE)))
                    ; faisons les comparaisons successives ;
                    (SETQ AUX1 (MATCH-AUX (CDR FILTRE) DONNEE))
                    NIL)  ; échec : contrainte non satisfaite
                  ; pas de contrainte : continuons
                  (SETQ AUX1 (MATCH-AUX (CDR FILTRE) DONNEE)))
                (IF AUX1 T  ; variable anonyme
                  (SETQ ALIST ALIST2) ; restaurons la alist
                  (IF AUX (NEXTL-DONNEE) ; avançons
                    ; la liaison de la variable s'agrandit ;
                    (NCONC (CAR ALIST) (LIST (NEXTL-DONNEE))))
                  (SELF ALIST)))  ; et continuons
              ; échec total : restaurons la A-liste ;
              (IF AUX (NEXTL ALIST))
              NIL))))))))

(DE VAR (FILTRE)       ; pour trouver la variable du filtre
  (IF (ATOM (CDR FILTRE)) (CDR FILTRE) (CADR FILTRE)))

(DE LINEAR (DONNEE)       ; pour linéariser
  (COND
    ((ATOM DONNEE)
     DONNEE)
    ((ATOM (CAR DONNEE))
     (CONS (CAR DONNEE) (LINEAR (CDR DONNEE))))
    ((MEMBER (CAAR DONNEE) '("$" "&" "%"))
     (CONS (CAR DONNEE) (LINEAR (CDR DONNEE))))
    (T
     (APPEND (CONS '<< (LINEAR (CAR DONNEE)))
          (CONS '>> (LINEAR (CDR DONNEE)))))))

(DE DELINEAR (DONNEE)       ; pour délinéariser
  (IF (ATOM DONNEE) DONNEE
    (DELINEAR-DONNEE)))

(DE DELINEAR-DONNEE ()       ; auxiliaire pour DELINEAR
  (COND
    ((ATOM DONNEE)
     DONNEE)
    ((EQ (CAR DONNEE) '<<)
     (NEXTL DONNEE)
```

```
     (CONS (DELINEAR-DONNEE) (DELINEAR-DONNEE)))
    ((EQ (CAR DONNEE) '>>)
     (NEXTL DONNEE)
     ())
    (T
     (CONS (NEXTL DONNEE) (DELINEAR-DONNEE)))))

(DE NEXTL-DONNEE ()       ; pour avancer dans les données linéarisées
  (IFN (EQ (CAR DONNEE) '<<) (NEXTL DONNEE)
    (NEXTL DONNEE)
    (DELINEAR-DONNEE)))


; les macro-caractères

(DMC "%" () (CONS '"%" (READ))) ; %X  -->  (% . X)

(DMC "$" () (CONS '"$" (READ))) ; $X  -->  ($ . X)

(DMC "&" () (CONS '"&" (READ))) ; &X  -->  (& . X)

; et finalement encore une fois la fameuse fonction LETALL

(DM LETALL (CALL)
  (LET ((ALIST (CADR CALL)))
    (RPLACB CALL
      '((LAMBDA ,(MAPCAR ALIST 'CAR) ,@(CDDR CALL))
           ,@(MAPCAR ALIST (LAMBDA (X) (LIST QUOTE (CDR X))))))))
```

Pour conclure ce livre, construisons un autre petit 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 taˆche de simplifier des expressions algébriques, un exemple de filtre peut être :

($X + 0)

et la partie action correspondant peut alors être **X**, c'est évidemment 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, i.e.: jusqu'à ce que plus aucune règle puissse s'appliquer.

Voici la boucle top-level de notre interprète :

```
(DE INTERPRETE (EXPR 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 parenthésées), l'ensemble des règles peut alors être la liste suivante :

```
(SETQ 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 correspondant et relancer l'ensemble des règles sur cette réécriture :

```
(DE SIMPL (EXPR REGL ALIST)(COND
    ((NULL EXPR)())
    ((ATOM EXPR) EXPR)
    ((CAR (SETQ ALIST (MATCH (CAAR REGL) EXPR)))
        (SIMPL (EVAL '(LETALL ,(CADR ALIST) ,(CADAR REGL))) REGLES))
    ((AND REGL (SIMPL EXPR (CDR REGL))))
    (T (CONS (SIMPL (CAR EXPR)  REGLES)
        (SIMPL (CDR EXPR)  REGLES)))))
```

C'est tout !

Voici une petite 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 suˆr, 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 une possibilité de grouper des règles afin de ne pas balayer l'ensemble de toutes les règles à chaque itération,
2. adjoindre une possibilité de modifier l'ordre des règles dépendant des données pour accélerer la recherche d'une règle qui s'applique.
3. et, finalement, adjoindre une possibilité permettant pour chaque règle plusieurs filtres ainsi que plusieurs actions.

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 l'activité de votre programmation personnelle, l'observation de vos erreurs ainsi que leurs corrections et votre interaction avec l'ordinateur, vous permettront de devenir un programmeur expert.

19. SOLUTIONS AUX EXERCICES

19.1. CHAPITRE 1

1. Détermination du type :

123	atome numérique
(EIN (SCHO%NES) 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 (SCHO%NES) BUCH)	3 éléments : **EIN, (SCHO%NES), BUCH**
(((((1) 2) 3) 4) 5)	2 éléments : **((((1) 2) 3) 4), 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** $(_2$**SCHO%NES**$)_2$ **BUCH**$)_1$	**SCHO%NES** 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

19.2. 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))**

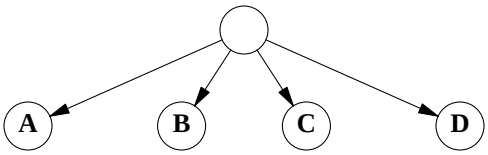
19.3. CHAPITRE 3

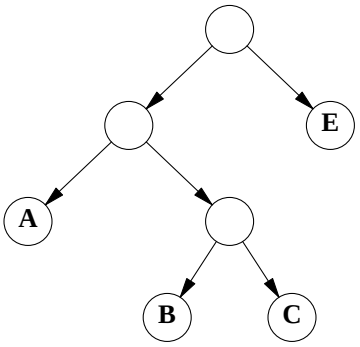
1. Combinaisons de **CAR** et **CDR** :

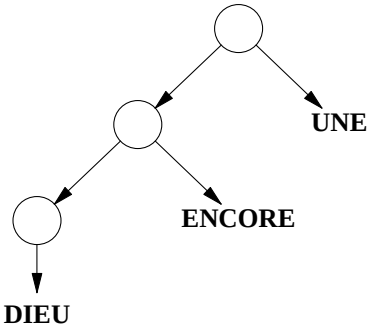
(CAR (CDR (CDR (CDR '(A B C D)))))	→	**D**
(CAR (CDR (CADR (CAR '((A (B C)) E)))))	→	**C**
(CAR (CAR (CAR '(((DIEU) ENCORE) UNE))))	→	**DIEU**

(CADR (CAR '(((DIEU) ENCORE) UNE))) → **ENCORE**

2. Traduction en forme arborescente :



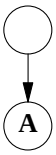




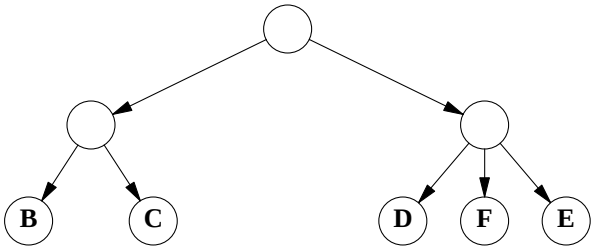
3. **SOL, ((C D)), ((HELLO) HOW ARE YOU), (JE JE JE BALBUTIE), (C EST SIMPLE)**

4. Calcul d'opérations sur des arbres :

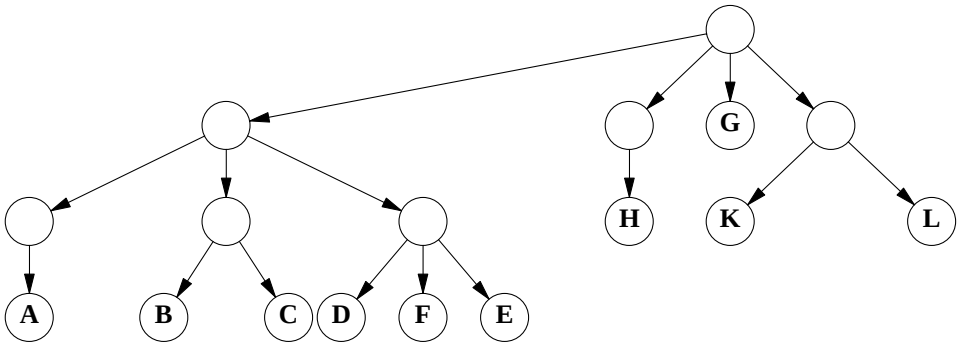
(A)



((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)))

19.4. 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 :

   (DE BONJOUR ()
   '(BONJOUR))

4. Quatre répétitions du même élément :

```
(DE 4FOIS (ARGUMENT)
   (CONS ARGUMENT
      (CONS ARGUMENT
         (CONS ARGUMENT
            (CONS ARGUMENT ())))))
```

5. Une fonction qui inverse une liste de 3 éléments :

```
(DE REVERSE3 (ARG1 ARG2 ARG3)
   (CONS ARG3 (CONS ARG2 (CONS ARG1 NIL))))
```

19.5. CHAPITRE 5

19.5.1. chapitre 5.1.1

1. Une fonction qui teste si le deuxième élément est un nombre :

```
(DE NUMBERP-CADR (L)
    (NUMBERP (CADR L)))
```

2. Une fonction qui teste si le premier élément est une liste :

```
(DE LISTE-CAR? (L)
   (CONS 'EST (CONS (CONSP (CAR L)) ())))
```

ou :

```
(DE 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**

19.5.2. chapitre 5.2.1

1. Une fonction qui teste si les 3 premiers éléments sont des nombres :

```
(DE 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 :

```
(DE 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)
```

Voici alors cette nouvelle fonction :

```
(DE MEMBER (ELE LISTE) (COND
    ((ATOM LISTE) NIL)
    ((EQUAL (CAR LISTE) ELE) LISTE)
    (T (MEMB1 ELE (MEMBER ELE (CAR LISTE)) (CDR LISTE)))))

(DE MEMB1 (ELE AUX LISTE)
     (IF AUX AUX (MEMBER ELE (CDR LISTE))))
```

6. Une fonction qui groupe les éléments successifs de deux listes :

```
(DE 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 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 l'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, mixés. 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 l'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 !

19.7. CHAPITRE 7

1. Pour la fonction **CNTH** 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 ?

```
(DE CNTH (NOMBRE LISTE) (COND
   ((NULL LISTE) NIL)
   ((LE N 1) (CAR LISTE))
   (T (CNTH (1- NOMBRE) (CDR LISTE)))))
```

2. L'écriture d'une fonction transformant des nombres décimaux en nombres octaux et hexadécimaux sera grandement facilitée si nous divisons cette taˆche en plusieurs sous-taˆches : 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**, traduisant 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.

```
(DE DEC-OCT (N)
 (IF (> N 0)
    (APPEND (DEC-OCT (/ N 8)) (APPEND (REM N 8) NIL)) NIL))

(DE DEC-HEX (N)
 (IF (> N 0)
    (APPEND (TRAD (DEC-HEXA (/ 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).

```
(DE TRAD (L)
 (IF (NULL L) ()
     (CONS
      (CNTH (1+ (CAR L)) '(0 1 2 3 4 5 6 7 8 9 A B C D E F)))
      (TRAD (CDR L)))))

(DE 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.

```
(DE OCT-DEC (N)
   (LET ((N N)(RES 0))
     (IF N (SELF (CDR N) (+ (* RES 8) (CAR N))) RES)))

(DE HEX-DEC (N)
   (LET ((N N)(RES 0))
     (IF N (SELF (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** :

```
(DE FIBONACCI (N M)
  (IF (LE N 1) 1
      (+ (FIBONACCI (1- N)) (FIBONACCI (- N 2)))))
```

Ci-dessous une version récursive terminale de cette même fonction :

```
(DE FIBONACCI (N)
   (IF (LE N 1) 1 (FIB 2 '(1 1) N)))

(DE FIB (COMPTEUR LAST N)
   (IF (= COMPTEUR N) (+ (CAR LAST)(CADR LAST))
       (FIB (1+ COMPTEUR)
            (CONS (CADR LAST)(CONS (+ (CAR LAST) (CADR LAST)) NIL))
            N)))
```

5. La fonction **QUOI** 'inverse´ 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 :

```
(DE 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. Seulement la première version est récursive terminale.

```
(DE LENGTH (L)
   (LET ((L L) (N 0))
        (IF (NULL L) N
            (SELF (CDR L) (1+ N)))))

(DE LENGTH (L)
    (IF (NULL L) 0 (1+ (LENGTH (CDR L)))))
```

8. 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 :

```
(DE NBATOM (L)
   (LET ((L L) (N 0))
        (IF (NULL L) N
          (IF (ATOM (CAR L)) (SELF (CDR L) (1+ N))
              (SELF (CDR L) (SELF (CAR L) N))))))

(DE NBATOM (L)
    (IF (NULL L) 0
       (IF (ATOM L) 1
           (+ (NBATOM (CAR L)) (NBATOM (CDR L))))))
```

19.8. CHAPITRE 8

1. Bien évidemment, afin de représenter ces diverses relations, il suffit de prendre les relations comme indicateurs sur les P-listes :

a. **(PUT ’GUY ’PERE ’PIERRE)**
   ou **(PUT ’PIERRE ’ENFANT ’(GUY))**
b. **(PUT ’MARIE ’PERE ’PIERRE)**
   ou **(PUT ’PIERRE ’ENFANT (CONS ’MARIE (GET ’PIERRE ENFANT)))**
c. **(PUT ’JACQUES ’PERE ’PIERRE)**
   ou **(PUT ’PIERRE ’ENFANT (CONS ’JACQUES (GET ’PIERRE ENFANT)))**
d. **(PUT ’PIERRE ’SEXE ’MASCULIN)**
e. **(PUT ’MARIE ’SEXE ’FEMININ)**
f. **(PUT ’GUY ’SEXE ’MASCULIN) (PUT ’JACQUES ’SEXE ’MASCULIN)**
g. **(PUT ’JUDY ’SEXE (GET ’MARIE ’SEXE))**
h. **(PUT ’JUDY ’AGE 22)**
i. **(PUT ’ANNE ’AGE 40)**
j. **(PUT ’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 taˆches, 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 :

(DE 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 :

(PUT ’A1 ’AUTEUR ’(KARL KRAUS))
(PUT ’A1 ’TITRE ’(DIE LETZTEN TAGE DER MENSCHHEIT))
(PUT ’A2 ’AUTEUR ’(KARL KRAUS))
(PUT ’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 :

```
(DE CHERCHEALL (PROP VAL) (CHERCHE PROP (BIBLIO)))

(DE CHERCHE (PROP BASE) (COND
    ((NULL BASE) NIL)
    ((EQUAL (GET (CAR BASE) PROP) VAL)
            (CONS (CAR BASE) (CHERCHE PROP (CDR BASE))))
    ((MEMBER VAL (GET (CAR BASE) PROP))
            (CONS (CAR BASE) (CHERCHE PROP (CDR BASE))))
    (T (CHERCHE PROP (CDR BASE)))))
```

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. Là voici :

```
(DE 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 :

```
(DE 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 :

```
(DE 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**.

```
(DE 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 (simple). A vous de le continuer !

3. Donnons d’abord les bonnes propriétés aux divers atomes :

```
(PUT ’ALBERT ’MERE ’MARTINE)     (PUT ’ALBERT ’PERE ’CLAUDE)
(PUT ’PIERRE ’MERE ’MARTINE)     (PUT ’PIERRE ’PERE ’JEAN)
(PUT ’KLAUS ’MERE ’EVE)          (PUT ’KLAUS ’PERE ’ALBERT)
(PUT ’GERARD ’MERE ’JULIE)       (PUT ’GERARD ’PERE ’PIERRE)
(PUT ’GUY ’MERE ’JULIE)          (PUT ’GUY ’PERE ’PIERRE)
(PUT ’MARIE ’MERE ’JULIE)        (PUT ’MARIE ’PERE ’PIERRE)
(PUT ’JACQUES ’MERE ’JULIE)      (PUT ’JACQUES ’PERE ’PIERRE)
(PUT ’ALAIN ’MERE ’ALICE)        (PUT ’ALAIN ’PERE ’KLAUS)
(PUT ’BRIAN ’MERE ’CATHERINE)    (PUT ’BRIAN ’PERE ’ALAIN)
(PUT ’ANNE ’MERE ’SARAH)         (PUT ’ANNE ’PERE ’GERARD)
(PUT ’CATHERINE ’MERE ’JANE)     (PUT ’CATHERINE ’PERE ’JACQUES)
(PUT ’JUDY ’MERE ’ANNE)          (PUT ’JUDY ’PERE ’PAUL)
```

et le sexe :

```
(DE PUT-SEXE (SEXE LISTE)
   (IF (NULL LISTE) NIL
      (PUT (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 nous calcule la liste de toutes les personnes :

```
(DE 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 :

```
(DE 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 ? Regardez : 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.

C;a 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 de trois fois écrire la même chose, j’ai décidé d’écrire une fonction auxiliaire

qui devra faire cela. Voici la petite fonction **FINDALL** :

```
(DE 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 légèrement améliorer cette fonction en sortant les arguments invariants de la boucle. Voici donc la deuxième version de la fonction **FINDALL** :

```
(DE FINDALL (INDICATEUR VALEUR LISTE)
  (LET ((LISTE LISTE)) (COND
      ((NULL LISTE) ())
      ((EQUAL (GET (CAR LISTE) INDICATEUR) VALEUR)
          (CONS (CAR LISTE) (SELF (CDR LISTE))))
      (T (SELF (CDR LISTE))))))
```

Mais continuons : une fonction qui trouve les soe;urs 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 :

```
(DE 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 rencontrerez dans un programme des fonctions presque identiques, vous pouvez être suˆr qu'il y a des optimisations à faire en construisant des fonctions auxiliaires qui captent la similarité. Ecrivons donc une fonction auxiliaire que nous nommerons **FINDALL-TROIS**, puisqu'elle doit trouver tous les éléments qui ont trois propriétés identiques :

```
(DE 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 :

```
(DE FRERES (X)
  (DELETE X (FINDALL-TROIS X 'PERE 'MERE 'SEXE 'MASCULIN)))

(DE SOEURS (X)
  (DELETE X (FINDALL-TROIS X 'PERE 'MERE 'SEXE 'FEMININ)))
```

et pour avoir à la fois les frères et les soe;urs, il suffit d'écrire :

```
(DE 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 :

```
(DE 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))))))))
```

19.9. CHAPITRE 9

1. Voici la fonction **FACTORIELLE** en tant que mémo-fonction :

```
(DE FACTORIELLE (N) (COND
    ((LE N 0) 1)
    ((GET 'FACTORIELLE N))
    (T (PUT '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 :

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

   Bien évidemment, ce programme peut encore se simplifier en :

```
(DE INVERSE (X) (- 0 X))
```

3. Voici un premier petit simplificateur algébrique :

```
(DE 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)))

(DE SIMPLIFY-PLUS (X Y) (COND
    ((AND (NUMBERP X) (NUMBERP Y)) (+ X Y))
    ((ZEROP X) Y)
    ((ZEROP Y) X)
    (T (CONS '+ (CONS X (CONS Y ()))))))

(DE 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 se sont évalués à 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 :

```
(DE 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 ()))))))
```

19.10. CHAPITRE 10

1. Si nous voulons généraliser ce petit programme de manière telle qu'il puisse conjuguer des verbes non seulement au présent, mais également en d'autres temps, la première chose à résoudre est : comment représenter les terminaisons supplémentaires ?

Actuellement, les terminaisons se touvent 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 cela que nous allons faire en créant deux atomes **ER** et **IR**. Sur les P-listes de ces deux atomes nous mettrons, sous des indicateurs correspondant au temps les listes de terminaisons. Ce qui donne :

```
(PUT    '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))
```

```
(PUT    '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 aˆmes aˆtes 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 aˆt aˆt 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** :

```
(DE 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 :

```
(DE VERBPRINT (PRONOM VERBE)
  (LET ((AUX (MEMQ TEMPS '(subjonctif-présent subjonctif-imparfait))))
    (IF AUX
       (IF (MEMQ PRONOM '(il elle ils elles)) (PRIN1 "qu'") (PRIN1 "que")))
    (PRINT PRONOM VERBE)))
```

et le tour est joué !

Néanmoins, comme vous avez suˆrement 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 quelques conjugaisons nécessaires du verbe *avoir*. Mettons-les sur la P-liste de l'atome **AVOIR** :

```
(PUT    '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 euˆmes euˆtes 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 euˆt euˆt eussions eussiez eussent eussent))
```

Ensuite construisons, comme précédemment pour le subjonctif, une fonction spéciale d'impression des temps composés :

```
(DE PR-COMPOS (PRON PARTICIPE)
  (LET ((AUX (GET 'AVOIR TEMPS))
        (PARTICIPE (CREE-MOT RACINE PARTICIPE))
        (PRON PRON))
    (IF (NULL AUX)("voila, c'est tout")
      (PRINT (IF (EQ (CAR PRON) 'je) "j'" (CAR PRON))
             (CAR AUX) PARTICIPE)
      (SELF (CDR AUX) PARTICIPE (CDR PRON)))))
```

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 qui sait faire 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 différent de **NIL**, et de lancer 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 associées au temps demandé :

```
(DE CONJ1 (RACINE TYP PRONOMS)
  (LET ((TERMINAISONS (GET TYP TEMPS)))
    (LET ((PRONOMS PRONOMS)(TERMINAISONS TERMINAISONS))
      (COND
        ((NULL PRONOMS) "voila, c'est tout")
        ((ATOM TERMINAISONS)
          (PR-COMPOS PRONOMS (IF (EQ TYP 'IR) 'i 'é)))
        (T (VERBPRINT (CAR PRONOMS)
                      (CREE-MOT RACINE (CAR TERMINAISONS)))
           (SELF (CDR PRONOMS)(CDR TERMINAISONS)))))))
```

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 :

```
(DE HANOI (N D A I AID)
   (WHEN (> N 0)
      (IF AID (HANOI (1- N) D I A) (HANOI N I A D N))
      (PRINT 'DISQUE N 'DE D 'VERS A)
      (IF (= N AID) (HAN (1- N) I D A) (HAN (1- N) I A D))))

(DE HAN (N D A I)
   (WHEN (> N 0)
      (HAN (1- N) D I A)
      (LET ((C 3))
        (WHEN (> C 0) (PRINT 'DISQUE N 'DE D 'VERS A) (SELF (1- C))))
      (HAN (1- N) I A D)))
```

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 différent de **NIL** la suite d'instructions qui compose le reste des arguments du **WHEN** est éxécutée et la valeur de la dernière expression est celle du **WHEN**, sinon, si 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 'depart 'arrivee 'intermediaire)
  DISQUE 1 DE intermediaire VERS arrivee
  DISQUE 1 DE depart VERS arrivee
= NIL
? (HANOI 2 'depart 'arrivee 'intermediaire)
  DISQUE 1 DE arrivee VERS depart
  DISQUE 1 DE intermediaire VERS depart
  DISQUE 2 DE intermediaire VERS arrivee
  DISQUE 1 DE depart VERS intermediaire
  DISQUE 1 DE depart VERS intermediaire
  DISQUE 1 DE depart VERS intermediaire
  DISQUE 2 DE depart VERS arrivee
  DISQUE 1 DE intermediaire VERS arrivee
  DISQUE 1 DE intermediaire VERS arrivee
  DISQUE 1 DE intermediaire VERS arrivee
= NIL
```

19.11. CHAPITRE 11

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)))))
```

   Visiblement, 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 très 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))))
```

19.12. CHAPITRE 12

1. La nouvelle version de la fonction **TIMES** :

```
(DE TIMES L
   (IF (NULL L) 1
       (* (CAR L) (APPLY 'TIMES (CDR L)))))
```

2. La nouvelle version de la fonction **SLIST** :

```
(DE SLIST L
   (IF (NULL (CDR L)) ()
       (CONS (CADR L) (APPLY 'SLIST (CDDR L)))))
```

3. La nouvelle version de la fonction **QSLIST** :

```
(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

4. Afin que notre petit traducteur mot à mot fonctionne au moins pour les quelques phrases exemples, nous avons besoin du vocabulaire suivant :

```
(DE CHAT () 'CAT)
(DE LE () 'THE)
(DE MANGE () 'EATS)
(DE SOURIS () 'MOUSE)
(DE A () 'HAS)
(DE VOLE () 'STOLEN)
(DE FROMAGE () 'CHEESE)
```

Ensuite nous avons besoin d'une fonction qui prend une suite quelconque de mots - ça sera donc une FEXPR - transforme chacun des 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 ;
```

19.13. CHAPITRE 13

1. Voici le macro-caractère '^' qui vous permet de charger un fichier simplement en écrivant

^nom-de-fichier

```
(DMC "^" () ['LIB (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.

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 :

```
(DMC ">" ()
  (LET ((X (PEEKCH)))
       (IF (NULL (EQUAL X "=")) '> (READCH) 'GE)))

(DMC "<" ()
  (LET ((X (PEEKCH)))
       (IF (NULL (EQUAL X "=")) '< (READCH) 'LE)))
```

3. D'abord le programme écrivant les nombres pairs de 2 à **N** dans le fichier **pair.nb** :

```
(DE PAIR (N)
   (OUTPUT "pair.nb")
   (LET ((X 2))(COND
         ((<= X N) (PRINT X) (SELF (+ X 2)))
          (T (PRINT "FIN") (OUTPUT) N))))
```

Ensuite faisons la même chose pour les nombres impairs :

```
(DE IMPAIR (N)
   (OUTPUT "impair.nb")
   (LET ((X 1))(COND
        ((<= X N) (PRINT X) (SELF (+ X 2)))
        (T (PRINT "FIN") (OUTPUT) N))))
```

Ensuite, il ne reste qu'à calculer la somme :

```
(DE SOMME ()
      (INPUT "impair.nb")
      (LET ((X (READ)) (IMPAIR))
            (IF (EQUAL X "FIN") (SOMM1 (REVERSE IMPAIR))
                (SELF (READ) (CONS X IMPAIR)))))

(DE SOMM1 (IMPAIR)
      (INPUT "pair.nb")
      (LET ((X (READ)) (PAIR))
            (IF (NULL (EQUAL X "FIN")) (SELF (READ) (CONS X PAIR))
                (INPUT)
                (SOMM2 IMPAIR (REVERSE PAIR)))))

(DE SOMM2 (IMPAIR PAIR)
   (OUTPUT "somme.nb")
   (LET ((IMPAIR IMPAIR) (PAIR PAIR))
   (COND
      ((AND (NULL IMPAIR) (NULL PAIR)) (OUTPUT))
      ((NULL IMPAIR) (PRINT (CAR PAIR)) (SELF NIL (CDR PAIR)))
      ((NULL PAIR) (PRINT (CAR IMPAIR)) (SELF (CDR IMPAIR) NIL))
      (T (PRINT (+ (CAR IMPAIR) (CAR PAIR)))
           (SELF (CDR IMPAIR) (CDR PAIR))))))
```

19.14. CHAPITRE 14

1. D'après ce que nous savons, tous les appels récursifs qui ne sont pas arguments d'une autre fonction, i.e.: 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* :

```
(DE MATCH (FILTRE DONNEE) (COND
        ((ATOM FILTRE) (EQ FILTRE DONNEE))
        ((ATOM DONNEE) NIL)
        ((EQ (CAR FILTRE) '$)
             (MATCH (CDR FILTRE) (CDR DONNEE)))
        ((EQ (CAR FILTRE) '&) (COND
             ((MATCH (CDR FILTRE) DONNEE))
             (DONNEE (MATCH FILTRE (CDR DONNEE)))))
        ((MATCH (CAR FILTRE) (CAR DONNEE))
             (MATCH (CDR FILTRE) (CDR DONNEE)))))
```

2. Afin de permettre le signe '**%**' en plus, 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 signe '**%**' particulier :

```
((AND (CONSP (CAR FILTRE)) (EQ (CAAR FILTRE) '%)) . . . )
```

Ensuite, similaire au 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 trouver 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. Seulement si aucun des filtres ainsi construits ne correspond au premier élément de la donnée nous devons signaler un échec.

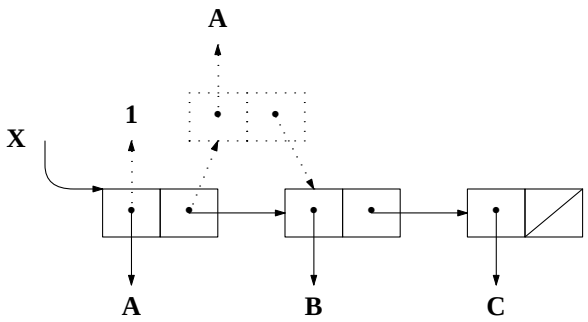
Voici la clause en entier :

```
((AND (CONSP (CAR FILTRE)) (EQ (CAAR FILTRE) '%))
      (LET ((AUX (CDR (CAR FILTRE)))) (COND
            ((NULL AUX) NIL)
            ((MATCH (CAR AUX) (CAR DONNEE))
                  (MATCH (CDR FILTRE) (CDR DONNEE)))
            (T (SELF (CDR AUX))))))
```

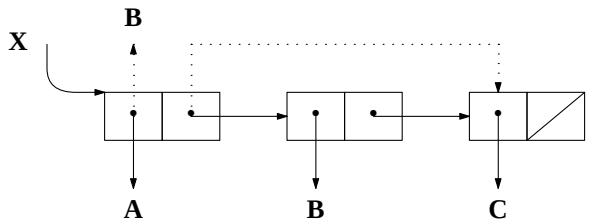
19.15. CHAPITRE 15

1. Dans les représentations graphiques suivantes, les listes originales sont dessinées avec des traits pleins, les modifications avec des traits en pointillés.

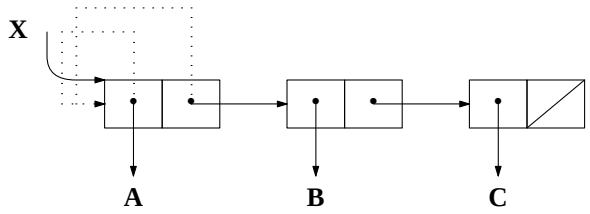
a. **(SET 'X '(A B C)) (ATTACH 1 X)**



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 assurons que tous les pointeurs vers la liste originale pointent après vers la liste modifiée.

```
(DE FDELQ (X Y)(COND
     ((NULL Y) ())
     ((EQUAL X (CAR Y)) (IF (CDR Y) (FDELQ X (SMASH Y)) ()))
     (T (DEL1 X Y))))

(DE DEL1 (X Y)(COND
     ((NULL Y) ())
     ((EQUAL X (CAR Y)) (DEL1 X (CDR Y)))
     ((CONSP (CAR Y))
          (RPLACA Y (DEL X (CAR Y))) (RPLACD Y (DEL1 X (CDR Y))))
     (T (RPLACD Y (DEL1 X (CDR Y))))))))
```

3. Evidemment, 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 :

```
(DE C-REVERSE (L)
     (AND L
          (LET ((C (CDR L)) (P L))
               (IF (NEQ L C) (SELF (CDR C) (RPLACD C P))
                    (RPLACD L P) P))))
```

4. Finalement la fonction **FFREVERSE** qui inverse physiquement une liste (sans circularités) sur tous les niveaux :

```
(DE FFREVERSE (L)
     (LET ((L L) (P))
          (IF L
               (IF (CONSP (CAR L))
                    (SELF (CDR L)
                          (RPLACD (RPLACA L (FFREVERSE (CAR L))) P))
                    (SELF (CDR L) (RPLACD L P)))
               P)))
```

5. Voici la définition de la fonction **NEXTL** :

```
(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 :

```
(DE RPLACB (OBJET LISTE)
    (RPLACA (RPLACD OBJET (CDR LISTE)) (CAR LISTE)))
```

Cette fonction s'appelle **DISPLACE** en LE_LISP.

19.16. CHAPITRE 16

1. Evidemment, les macro-fonctions **INCR** et **DECR** sont très similaires. Les voici :

```
(DM INCR (L)
    (RPLACB L '(SETQ ,(CADR L) (1+  ,(CADR L)))))
```

et en utilisant la fonction **DMD** :

```
(DMD DECR L
    '(SETQ ,(CADR L) (1-  ,(CADR L))))
```

2. et voici la fonction **SETNTH** :

```
(DMD SETNTH L
    '(RPLACA (NTH ,(CADDR L) ,(CADR L)) ,(CADR (CDDR L))))
```

19.17. CHAPITRE 17

1. Voici la fonction **MAPS** qui applique une fonction à tous les sous-structures d'une liste :

```
(DE MAPS (L F)(COND
        ((NULL L)())
        ((ATOM L)(F L))
        (T (F L)(MAPS (CAR L) F)(MAPS (CDR L) F)))
```

2. Evidemment, l'unique différence entre la fonction **WHILE** et la fonction **UNTIL** est le sens du test : il suffit de l'inverser. Ce qui donne :

```
(DMD UNTIL CALL
        '(LET ()
            (IF ,(CADR CALL) ()
                ,@(CDDR CALL) (SELF)))))
```

et la deuxième version :

```
(DMD UNTIL CALL
        '(IF ,(CADR CALL) ()
            ,@(CDDR CALL) ,CALL))
```

3. La fonction **FACTORIELLE** avec l'itérateur **DO** :

```
(DE FACTORIELLE (N)
    (DO ((N N (1- N))
         (R 1 (* N R)))
        ((LE N 1) R)))
```

4. Et finalement la macro-fonction **DO** en utilisant **WHILE** :

```
(DM DO (CALL)
    (LET ((INIT (CADR CALL)) (STOPTEST (CAAR (CDDR CALL)))
          (RETURN-VALUE (CDAR (CDDR (CALL))))
          (CORPS (CDDDR CALL)))
    (RPLACB CALL
          '((LAMBDA ,(MAPCAR INIT 'CAR)
              (WHILE (NULL ,STOPTEST)
                     ,@CORPS
                     ,@(MAPCAR INIT
                          (LAMBDA (X)
                             '(SETQ ,(CAR X) ,(CADDR X)))))
             ,@RETURN-VALUE)
            ,@(MAPCAR INIT 'CADR)))))
```

21. 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 - éventuellement - des informations supplémentaires sur le type des arguments. Nous distinguons trois types de fonctions standard : les SUBRs, les NSUBRs et les FSUBRs. Les SUBRs sont des fonctions à nombre d'arguments fixe où chaque argument est évalué. Les NSUBRs sont des fonctions standard à nombre d'arguments variable où chaque argument est évalué. Finalement, les FSUBR sont des fonctions standard à nombre d'arguments variable où les arguments ne sont pas obligatoirement évalués. Ces trois types de fonctions standard sont analogues aux trois types des fonctions utilisateurs : les EXPRs, des fonctions à nombre d'arguments fixe et évaluation de chaque argument, les NEXPRs, des fonctions à nombre d'arguments variable avec évaluation de chaque argument, et, finalement, les FEXPRs, qui admettent un nombre d'arguments variable et n'évaluent pas les arguments.

Le premier tableau concerne les fonctions **VLISP**, le deuxième donne les fonctions de **LE_LISP**.

VLISP

Fonction	Type	Arguments	Type
*	SUBR	2	nombres
+	SUBR	2	nombres
-	SUBR	2	nombres
/	SUBR	2	nombres
1+	SUBR	1	nombre
1-	SUBR	1	nombre
<	SUBR	2	nombres
=	SUBR	2	atomes
>	SUBR	2	nombres
ADDPROP	SUBR	3	atome, indicateur, valeur
AND	FSUBR	n	S-expressions
APPEND	SUBR	2	listes
APPLY	SUBR	2	fonction, liste d'arguments
ASSQ	SUBR	2	S-expression
ATOM	SUBR	1	S-expression
ATTACH	SUBR	2	S-expression, liste
CADDR	SUBR	1	liste
CADR	SUBR	1	liste
CAR	SUBR	1	liste
CDR	SUBR	1	liste
CNTH	SUBR	2	nombre, liste
COND	FSUBR	n	clauses
CONS	SUBR	2	S-expressions
DE	FSUBR	n	S-expressions
DECR	SUBR	1	nombres
DELETE	SUBR	2	S-expression, liste
DF	FSUBR	n	S-expressions

DM	FSUBR	n	S-expressions
DMC	FSUBR	n	S-expressions
DO	MACRO	n	S-expressions
EQ	SUBR	2	S-expressions
EQUAL	SUBR	2	S-expressions
EVAL	SUBR	1	S-expression
EVENP	SUBR	1	nombre
EXPLODE	SUBR	1	atome
GE	SUBR	2	nombres
GET	SUBR	2	S-expression, indicateur
IF	FSUBR	n	S-expressions
IMPLODE	SUBR	1	liste de caractères
INCLUDE	FSUBR	1	nom de fichier
INCR	SUBR	1	nombre
INPUT	SUBR	1	NIL ou nom de fichier
LAMBDA	FSUBR	n	S-expressions
LAST	SUBR	1	liste
LE	SUBR	2	nombres
LENGTH	SUBR	1	liste
LET	FSUBR	n	S-expressions
LIB	FSUBR	1	nom de fichier
LIST	NSUBR	n	S-expressions
LISTP	SUBR	1	S-expression
MAPC	SUBR	2	liste, fonction
MAPCAR	SUBR	2	liste, fonction
MCONS	NSUBR	n	S-expressions
MEMBER	SUBR	2	S-expression, liste
MEMQ	SUBR	2	atome, liste
NCONC	SUBR	2	liste, liste
NEXTL	FSUBR	1	atome
NULL	SUBR	1	S-expression
NUMBP	SUBR	1	S-expression
OBLIST	SUBR	0	
ODDP	SUBR	1	nombre
OR	FSUBR	n	S-expressions
OUTPUT	SUBR	1	NIL ou nom de fichier
PEEKCH	SUBR	0	
PLENGTH	SUBR	1	atome
PRIN1	NSUBR	n	S-expressions
PRINC	NSUBR	n	S-expressions
PRINCH	SUBR	2	S-expression, nombre
PRINT	NSUBR	n	S-expressions
PROBEF	SUBR	1	nom de fichier
PUT	SUBR	3	atome, indicateur, valeur
QUOTE	FSUBR	1	S-expression
READ	SUBR	0	
READCH	SUBR	0	
REM	SUBR	2	nombres
REMPROP	SUBR	2	atome, indicateur
REVERSE	SUBR	1	liste
RPLACA	SUBR	2	liste, S-expression
RPLACB	SUBR	2	liste, liste
RPLACD	SUBR	2	liste, S-expression

SELF	FSUBR	n	S-expression
SET	SUBR	2	S-expression, valeur
SETQ	FSUBR	2 ou 2*n	atome, valeur
SMASH	SUBR	1	liste
STRINGP	SUBR	1	S-expression
TERPRI	SUBR	0 ou 1	nombre
TYI	SUBR	0	
TYO	SUBR	1	nombre
TYPCH	SUBR	1	caractère
UNLESS	FSUBR	n	S-expressions
UNTIL	FSUBR	n	S-expressions
WHEN	FSUBR	n	S-expressions
WHILE	FSUBR	n	S-expressions
ZEROP	SUBR	1	nombre

LE_LISP

Fonction	**Type**	**Arguments**	**Type**
*	NSUBR	n	nombres
+	NSUBR	n	nombres
-	NSUBR	n	nombres
/	SUBR	2	nombres
1+	SUBR	1	nombre
1-	SUBR	1	nombre
<	SUBR	2	nombres
<=	SUBR	2	nombres
=	SUBR	2	atomes
>	SUBR	2	nombres
>=	SUBR	2	nombres
ADDPROP	SUBR	3	atome, valeur, indicateur
AND	FSUBR	n	S-expressions
APPEND	SUBR	2	listes
APPLY	SUBR	2	fonction, liste d’arguments
ASSQ	SUBR	2	S-expression
ATOM	SUBR	1	S-expression
ATOMP	SUBR	1	S-expression
CADDR	SUBR	1	liste
CADR	SUBR	1	liste
CAR	SUBR	1	liste
CDR	SUBR	1	liste
COND	FSUBR	n	clauses
CONS	SUBR	2	S-expressions
CONSP	SUBR	1	S-expression
DE	FSUBR	n	S-expressions
DECR	SUBR	1	nombres
DELETE	SUBR	2	S-expression, liste
DF	FSUBR	n	S-expressions
DISPLACE	SUBR	2	liste, liste
DM	FSUBR	n	S-expressions
DMC	FSUBR	n	S-expressions
DMD	FSUBR	n	S-expressions

DO	MACRO	n	S-expressions
EQ	SUBR	2	S-expressions
EQUAL	SUBR	2	S-expressions
EVAL	SUBR	1	S-expression
EVENP	SUBR	1	nombre
EXPLODECH	SUBR	1	atome
FUNCALL	SUBR	n	fonction, S-expressions
GETPROP	SUBR	2	S-expression, indicateur
IF	FSUBR	n	S-expressions
IMPLODECH	SUBR	1	liste de caractères
INCR	SUBR	1	nombre
INPUT	SUBR	1	NIL ou nom de fichier
LAMBDA	FSUBR	n	S-expressions
LAST	SUBR	1	liste
LENGTH	SUBR	1	liste
LET	FSUBR	n	S-expressions
LETN	FSUBR	n	S-expressions
LIST	NSUBR	n	S-expressions
LOAD	FSUBR	2	nom de fichier
MAPC	NSUBR	n	fonction, listes
MAPCAR	NSUBR	n	fonction, listes
MCONS	NSUBR	n	S-expressions
MEMBER	SUBR	2	S-expression, liste
MEMQ	SUBR	2	atome, liste
NCONC	NSUBR	n	listes
NEXTL	FSUBR	1	atome
NULL	SUBR	1	S-expression
NUMBERP	SUBR	1	S-expression
OBLIST	SUBR	2	
ODDP	SUBR	1	nombre
OR	FSUBR	n	S-expresisons
OUTPUT	SUBR	1	NIL ou nom de fichier
PEEKCH	SUBR	0	
PLENGTH	SUBR	1	atome
PLIST	SUBR	1 ou 2	atome, liste
PRIN	NSUBR	n	S-expressions
PRINCH	SUBR	2	S-expression, nombre
PRINFLUSH	NSUBR	n	S-expressions
PRINT	NSUBR	n	S-expressions
PUTPROP	SUBR	3	atome, valeur, indicateur
QUOTE	FSUBR	1	S-expression
READ	SUBR	0	
READCH	SUBR	0	
REM	SUBR	2	nombres
REMPROP	SUBR	2	atome, indicateur
REVERSE	SUBR	1	liste
RPLACA	SUBR	2	liste, S-expression
RPLACD	SUBR	2	liste, S-expression
SET	FSUBR	2 ou 2*n	S-expression, valeur
SETQ	FSUBR	2 ou 2*n	atome, valeur
STRINGP	SUBR	1	S-expression
SYMEVAL	SUBR	1	atome
TERPRI	SUBR	0 ou 1	nombre

TYI	SUBR	0	
TYO	NSUBR	n	nombres
TYPECH	SUBR	1 ou 2	caractère, nombre
UNLESS	FSUBR	n	S-expressions
UNTIL	FSUBR	n	S-expressions
WHEN	FSUBR	n	S-expressions
WHILE	FSUBR	n	S-expressions
ZEROP	SUBR	1	nombre

20. 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).

[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).

[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, New York (1984).

[Friedman74] D. Friedman, *The Litlle 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

[Henderson] P. Henderson, *Functionnel Programming Application and Implementation*, University of Newcastle upon Type, Prentice˜Hall international, ().

[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, Universita%t 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

[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).

[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 Einfu%hrung 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*, Masson, Paris New York (1985).

[Winston77] P. H. Winston, *Artificial Intelligence,* Addison Wesley (1977).

[Winston81] P. H. Winston et B. K. P. Horn, *LISP,* Addison Wesley (1981).