CHAPITRE 9

INTRODUCTION AU LISP

Bien qu'ancien car créé en 1960 par Mac Carthy, et fondé sur le λ-calcul de Church, le Lisp est toujours le plus répandu, le plus simple des langages fonctionnels et le plus utilisé en intelligence artificielle. Pratiquement tous les problèmes évoqués dans les deux chapitres précédents sont résolus de façon magistrale en Lisp.

Fonctionnel signifie que tout ce qui est décrit, l'est sous forme de fonctions retournant un résultat calculé suivant les valeurs de ses arguments en entrées. C'est de plus un langage de manipulations symboliques : par exemple, dans la liste formée par (+ 3 2), les trois éléments jouent des rôles tout à fait analogues, on peut ne pas faire la coupure habituelle entre programme et données, néanmoins la liste pourra être évaluée à 5 si besoin est.

Il est parfaitement adapté à tous les problèmes de programmation approfondie, car d'un principe et d'une syntaxe très simple, le programmeur n'épuise pas toute son énergie en détails secondaires; le Lisp permet à des débutants d'accéder rapidement à l'essentiel des grands problèmes. Il favorise la programmation interactive, tout ce qui entre au clavier est systématiquement évalué, l'algorithme général d'un interpréteur Lisp se traduisant par la boucle : lecture - évaluation - impression.

Lisp possède de très nombreuses versions dont certaines sont compilées, (il existe même des machine-Lisp où celui-ci est en quelque sorte le langage du processeur) mais on se contentera ici d'une quinzaine de mots communs à toutes les versions, en évitant les problèmes d'entrée-sortie et de modification physique des objets. On peut parfaitement laisser de côté ces aspects de la programmation en Lisp, au début.
La notion de programme est assez différente de celle d'un langage classique, on définit en Lisp des fonctions de manière vraiment modulaire, qui viennent enrichir au fur et à mesure l'arsenal des fonctions prédéfinies. Un programme est une collection de fonctions que l'on met au point séparément. C'est l'idée de coprogramme.
Le Lisp étant particulièrement bien adapté à la récursivité, il faut pour bien programmer en Lisp abandonner les habitudes algorithmiques besogneuses : éliminer les affectations pour penser en terme de passage de paramètres, remplacer les itérations par des appels récursifs, les débranchements par des appels de fonctions, et la programmation séquentielle en composition de fonctions. Il est toujours possible de suivre ces recommandations en dédoublant les fonctions et en multipliant les paramètres.
Le lecteur s'habituera peu à peu à la description récursive des algorithmes en étudiant les premiers exemples. On introduira alors la notion de récursivité terminale.

Les objets du Lisp

Il n'y a pas de type en Lisp, la mémoire est simplement partagée entre deux zones réservées aux atomes et aux listes.

Les atomes représentent grosso-modo des objets insécables, deux d'entre eux jouent un rôle particulier :

nil (du latin "rien") encore noté comme une liste vide () symbolise en outre le faux, c'est le seul objet qui soit à la fois une liste et un atome.
t est l'atome symbolisant le vrai (true), il faut d'ailleurs noter que dans un test, tout ce qui n'est pas nil, peut jouer le rôle du vrai, notion adoptée par le C++.
Les autre atomes sont les valeurs numériques dont l'évaluation est elle-même, et les chaînes de caractères.

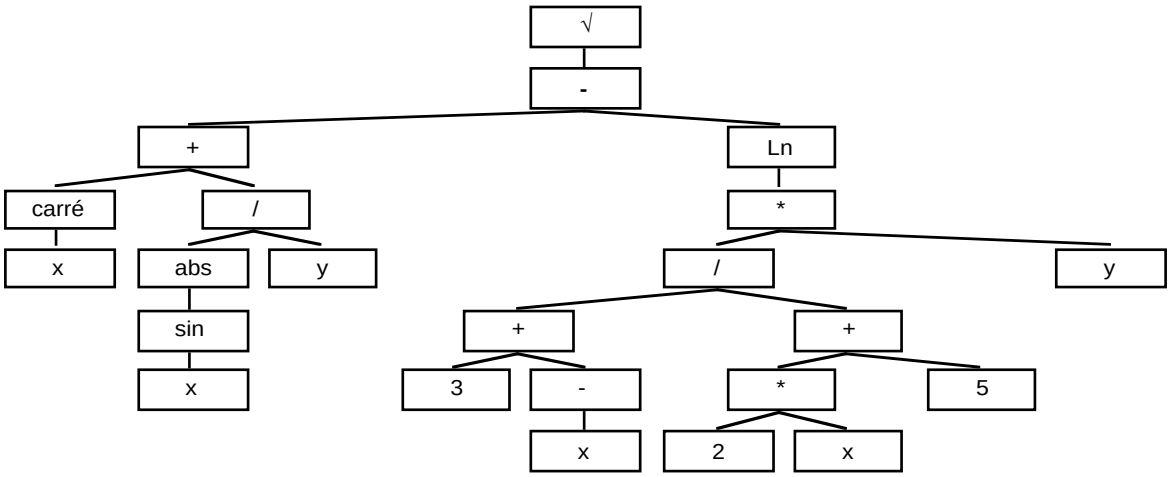
Les listes sont constituées par le plus petit ensemble contenant nil et stable par la fonction "cons" définie plus loin, elles sont notées avec des parenthèses.
Par exemple (a b c), (x), ((a) b (a b) ((c)) d) sont des listes.
Les listes sont en quelque sorte des fichiers (limités) à accès séquentiel, c'est-à-dire que pour accéder à un élément quelconque d'une liste, il faut avoir parcouru cette liste depuis le début jusqu'à la position voulue.

Notation préfixée : c'est la notation en vigueur pour le Lisp.

L'expression E est ici écrite suivant la façon traditionnelle, ou notation infixe, ce qui signifie par exemple que le signe + se trouve entre les deux termes de l'addition.

$$E = \sqrt{x^2 + \frac{|\sin(x)|}{y} - \log\left(\frac{3 + (-x)}{2x + 5}\, y\right)}$$

En fait, une telle expression correspond à l'arborescence suivante, et la notation infixe résulte d'une lecture gauche - racine - droite, de l'arbre.



La notation suffixée ou polonaise, (utilisée en Forth), résulte, elle, d'une lecture gauche - droite - racine , ce qui donne par exemple ici :

$$E_s = x\ \text{carré}\ x\ \sin\ \text{abs}\ y\ /\ +\ 3\ x\ \text{opp}\ +\ 2\ x\ *\ 5\ +\ /\ y\ *\ \text{Ln}\ -\ \sqrt{}$$

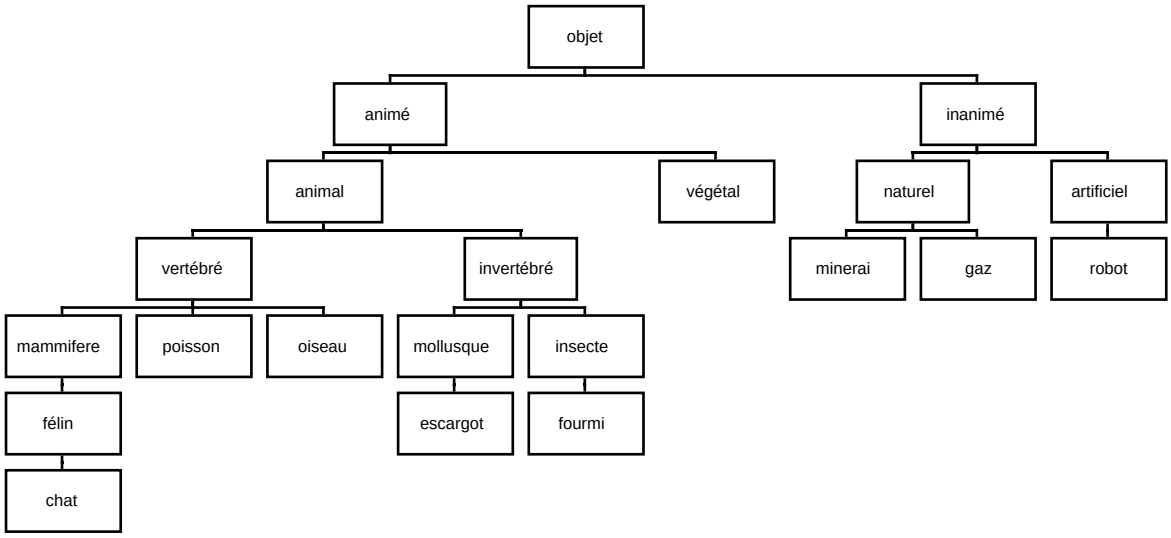
On a noté - la soustraction de deux arguments et opp la fonction "opposé" à un seul argument.

La notation préfixée (racine-gauche-droite) du Lisp consiste à écrire au contraire, la fonction avant ses opérandes ce qui donne ici:

E_p = (√ (- (+ (carré x)(/ (abs (sin x)) y)) (Ln (* (/ (+ 3 (opp x)) (+ (* 2 x) 5)) y))))

Le signe séparateur est l'espace, ab+cd s'écrit donc : (+ (* a b) (* c d))

On remarquera le parenthésage systématique englobant toujours le nom de la fonction suivi de ses arguments; cela permet de lever des ambiguïtés comme celle du signe "moins" qui peut être à un ou deux arguments, ou bien dans l'écriture de termes tels que |xy|z|t| ; mais cela a par ailleurs une interpétation très intéressante : chaque parenthèse ouvrante correspond à une descente dans l'arbre et chaque parenthèse fermante à une remontée. Prenons une classification simple :



Cet arbre pourrait se représenter par la liste :

```
(objet      (animé      (animal     (vertébré   (mammifère   (félin (chat)))
                                                (poisson)
                                                (oiseau) )
                                    (invertébré (mollusque   (escargot))
                                                (insecte     (fourmi)))
                        (végétal)
            (inanimé    (naturel    (minerai)
                                    (gaz))
                        (artificiel (robot)) ))
```

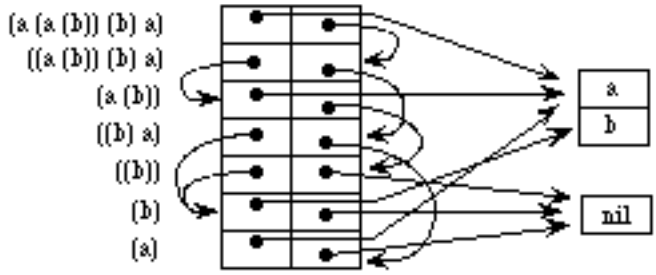
On constatera en lisant cette liste de gauche à droite, que chaque parenthèse ouvrante signifie bien une descente dans l'arbre , et chaque parenthèse fermante une remontée. (Dessiner l'ordre de parcours de l'arbre.)

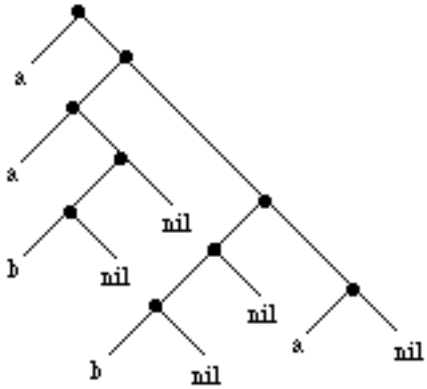
Représentation interne

Bien que le propos de ce livre ne soit pas d'étudier la structure interne du Lisp, il faut savoir que la représentation en mémoire des listes ne se fait pas du tout ainsi, mais suivant des "arbres binaires": chaque liste est repérée par un doublet d'adresses, celle de son premier élément (ce que l'on appelera son "car", qui peut lui-même être une liste), et celle de la sous-liste formée par ce qui suit ce premier élément (la queue ou le "cdr").

Une liste est donc représentée sous forme d'un d'arbre binaire c'est à dire informatiquement une liste chaînée de doublets de pointeurs. Le premier de ces pointeurs (le "car") pointe sur le premier élément (ci-dessous, sur l'atome "a" dans la zone mémoire réservée aux atomes), le second (le "cdr") pointant toujours dans la zone liste sur la sous-liste formée par la queue (ci-dessous le doublet placé en dessous). Naturellement le "car" peut lui même pointer sur une liste, c'est ce qui se passe pour la liste représentée au deuxième doublet, son "car" est le troisième, son "cdr" est le quatrième. La liste vide est un atome bien particulier c'est à dire une adresse unique en mémoire. "car" et "cdr" signifient depuis l'origine, contenus des régistres adresse et "à décrémentation".

Exemple de la liste (a (a (b)) (b) a), on a figuré à gauche une tranche de la zone liste avec l'écriture lispienne de chacune d'entre elles en regard.





Dans la figure de droite, on a la représentation de la même liste sous forme d'arbre binaire où le car de chaque noeud est le fils gauche, et le cdr le fils droit.

Les premières fonctions primitives "car", "cdr" et "cons"

On peut programmer des problèmes très complexes avec seulement une quinzaine de fonctions de base (car, cdr, cons, quote, eval, de, cond, atom, eq...), les autres pouvant se réécrirent en Lisp (and, or, append, list, member, null, ...) ou d'un usage moins courant (nth, set, read, print, explode,...).
On donne ici neuf fonctions suivies de quelques exemples. A chaque fois que l'on tape une expression au clavier, suivie de "return", le Lisp renvoie un résultat (qui peut être un message d'erreur). Cet évenement sera symbolisé dans ce qui suit, par la flèche ⇨.
Le rôle de l'apostrophe est étudié dans le paragraphe suivant.

(car L) ⇨ premier élément de la liste L
(car '(a b c d e)) ⇨ a
(car '((a b c) (d e) (f) g h)) ⇨ (a b c)

(cdr L) ⇨ liste formée par ce qui suit le premier élément de L
(cdr '(a b c)) ⇨ (b c)
(cdr '(a)) ⇨ nil

car et cdr se combinent: par exemple (caar '((a b) c) ⇨ a
(cadar '((u v w) x y)) ⇨ v
(cddr '(a b c)) ⇨ (c)
(cddar '((a b c d) (e) f)) ⇨ (c d)

(cons X L) ⇨ liste L augmentée de l'élément X au début
(cons 'X '(Y Z)) ⇨ (X Y Z)

(list X Y Z ...) ⇨ liste formée par les éléments X Y Z...

(member X L) ⇨ première sous-liste de L débutant par X
(member 'X '(Y U X Z X)) ⇨ (X Z X) (member 'X '(Y U)) ⇨ ()

(append L M) ⇨ liste obtenue par concaténation de L et M
(append '(a b) '(c d e)) ⇨ (a b c d e)

(nth N L) ⇨ N-ième élément de L (à partir de 0)
(nth 2 '(a b c d)) ⇨ c

(nthcdr N L) ⇨ N-ième "cdr" de L
(nthcdr 3 '(a b c d e)) ⇨ (d e)

(length L) ⇨ longueur de la liste L
(length '((a) (b c) ((d)) (a) b)) ⇨ 5

L'évaluation et la non-évaluation "eval" et "quote"

(eval X) ⇨ résultat de l'évaluation de X, par exemple : (eval '(+ 5 8)) ⇨ 13

(quote X) ⇨ X ou encore 'X ⇨ X (empêche l'évaluation , il existe un "macro-caractère" c'est-à-dire une abréviation pour cette fonction, c'est l'apostrophe.) Dans ce qui précède comme par exemple (car '(a b)), si l'apostrophe est oubliée, c'est la fonction "a" qui doit s'appliquer sur la valeur "b", ce qui provoque une erreur sauf si a et b ont été préalablement affectés.

Définition de fonction

Suivant les différents Lisp la définition d'une nouvelle fonction se fait au moyen de "define", "defun", "def" ou ici dans la version utilisée "lelisp" par "de".

(de F P L) ⇨ L permet de définir une fonction de nom F pour une liste de paramètres P, par une liste L, ainsi : (de cube (x) (* x x x)) pourra s'utiliser par (cube 3) ⇨ 27.
Remarque, dans la version Scheme, une définition se fait par (define (f x y z ...) < corps de la fonction>) ce qui est plus rationnel du point de vue des notations.

Prédicats "eq", "atom", "null"

En Lisp, les prédicats ou relations, sont représentés par leur fonction caractéristique, c'est-à-dire que leur évaluation renverra toujours le vrai ou le faux. En fait, lorsque la valeur renvoyée n'est pas le faux, elle peut toujours représenter le vrai. Ainsi par exemple la fonction "member" déjà rencontrée, peut aussi bien être utilisée pour connaître son résultat, que comme un simple test de présence d'un élément dans une liste.

(atom L) ⇨ T (vrai) si L est un atome, nil dans le cas contraire.

(eq X Y) ⇨ T si X et Y sont vraiment égaux

(equal X Y) ⇨ T si X et Y sont deux objets quelconques, mais ayant la même évaluation.

(eq A (cons (car A) (cdr A))) ⇨ nil

(equal A (cons (car A) (cdr A))) ⇨ T car l'évaluation du deuxième argument a eu pour effet de reconstruire un objet qui n'est autre que A.
On dispose naturellement de <> < <= > >= , comme d'ailleurs des fonctions arithmétiques.

(null L) ⇨ T si la liste L est vide (prédicat de vacuité)

(null (cdr '(a))) ⇨ T

(null '(nil)) ⇨ nil

on trouve aussi :

(zerop X) ⇨ T si X est 0 et nil si X n'est pas zéro.

(and L1 L2 L3...Ln) évalue séquentiellement les Li et renvoie nil dès que l'une est nil, et renvoie (eval Ln) sinon. (conjonction logique)

(and (< 2 5) (= 3 3)) ⇨ T

(or L1 L2 L3...Ln) évalue les Li jusqu'à ce que l'une d'entre elles soit différente de nil et retourne celle-ci, renvoie nil dans le cas contraire. (disjonction logique)

(or (< 4 8) (= 4 7) (> 1 -2)) ⇨ T (en fait l'expression (< 4 8) renvoie 4)

La condition "cond", c'est la "structure" la plus importante à acquérir, on lui donne en argument des "clauses" ayant généralement chacune 2 éléments qui sont en quelque sorte le test et l'action à effectuer en cas de succès du test (c'est en fait un peu plus général que cela).

```
(cond       (C1 L11 L12 L13....)
            (C2 L21 L22 L23....)
             .....................
            (Ck Lk1 Lk2 Lk3....) )
```

évalue les L_i qui suivent la première "condition" C_i différente de nil et retourne la dernière des évaluations de la liste débutant par ce C_i, cond retourne nil si toutes les C_i sont des expressions évaluées à nil.

(cond ((null nil) 'oui)) ⇨ oui (cond((> 2 3) 'non) (t 'oui)) ⇨ oui

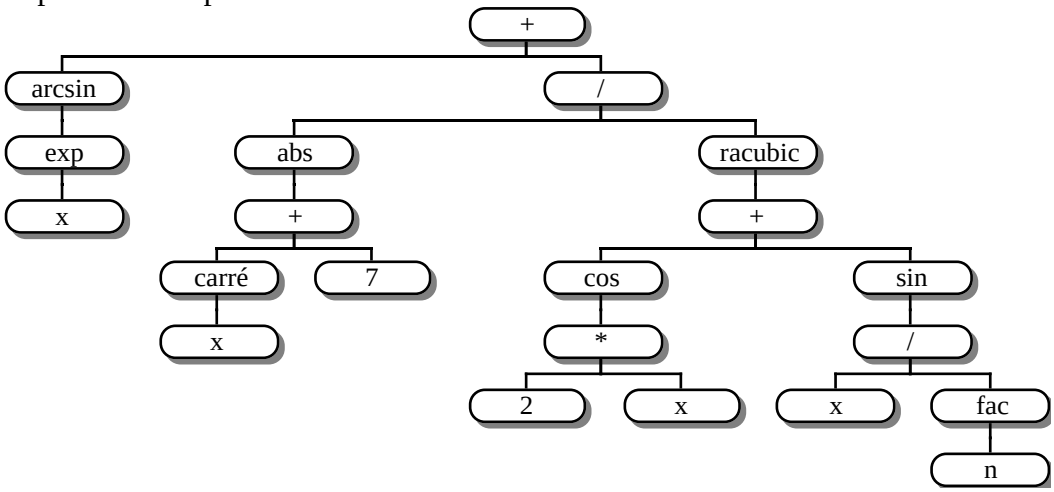
En effet le premier "test" (> 2 3) est évalué à faux, le second, qui est "t", est toujours vrai, écrire "t" dans la dernière clause permet de traduire tous les autres cas que ceux qui sont énumérés dans les clauses précédentes.
Cond correspond à une "instruction" de sélection, mais elle est bien plus puissante que les sélections des langages classiques et lorsqu'on est familiarisé avec son utilisation, on s'aperçoit que c'est la structure vraiment fondamentale pour la description claire d'un algorithme.
Deux autres possibilités existent pour sélectionner suivant des cas, tout d'abord le "if" lorsque l'on a que deux cas à examiner, sa structure est (if test alors sinon) dont l'évaluation est celle de "alors" quand "test" est différent de nil.

Exemple (if (< x 0) (- x) x) sera toujours évalué comme la valeur absolue de x.
L'autre façon est semblable au "case" du Pascal en alignant une série d'associations antécedent-images, par exemple :
(selectq x (0.5 'demi) (0.25 'quart) (0.333 'tiers)) donnera le mot correspondant à quelques valeurs de x.
Une abréviation "ifn" existe également pour la composée des fonctions "if" et "not".
Il existe bien d'autres fonctions prédéfinies, ainsi l'exemple du chapitre suivant sur les chiffres romains, utilise-t-il la fonction "explodech" renvoyant la liste des signes composant une chaîne de caractères. ("unpack" ou "explode" dans différentes versions de Lisp, la fonction contraire étant "implodech", "pack" ou "implode" suivant les cas.)

9-1° Mettre l'expression suivante sous forme suffixée et dessiner l'arborescence des calculs à effectuer pour son évaluation.

$$F_n(x) = \text{Arcsin}(e^x) + \frac{|\,x^2 + 7\,|}{\sqrt[3]{\cos(2x) + \sin(x/n!)}}$$

Réponses : x exp arcsin x carré 7 + abs 2 x * cos x n fac / sin + racubic / +



9-2° Mettre l'expression suivante sous forme préfixée, puis écrire l'expression-lisp parenthésée qui lui correspond.

$$\frac{37 * (473 + 12) - \dfrac{51 + 73}{103}}{-121 * 65}$$

Réponse : (/ (- (* 37 (+ 473 12)) (/ (+ 51 73) 103)) (* (- 121) 65))

9-3° Que rendent les évaluations des expressions suivantes (expliquez les intermédiaires) :
a) (car (cdr '((a b) (c d) (e) f)))
b) (car (cdr (car '((a b c) (d e f)))))
c) (car (cdr '(car ((a b c) (d e f)))))
d) (car '(cdr (car ((a b c) (d e f)))))
e) (eval (cons '+ (cons 4 (list 5))))
f) (cons (atom nil) (cons (atom (cdr '(nil nil))) (list (null '(nil)) (null (car '(nil))))))
g) (eval (cons (car (cdr '(3 + 4))) (cdr (list '(6 3 4)))))

Réponses : (c d), b, ((a b c) (d e f)), cdr, 9, (t nil nil t), 7.

9-4° Ecrire sous forme préfixée :
$\cos(a + b) + \cos(a - b)$
$3x + \text{Log}(5x + 4) - \sqrt{|x|}$
$4x^3 - 5x^2 + 3x + 1$

Réponses : + cos + a b cos - a b, puis + * 3 x - ln + * 5 x 4 √ abs x, enfin - * 4 cube x + * 5 carré x + * 3 x 1

9-5° Que donnent les quatre évaluations de :
(car (cdr (cons '(a b) (list '(c d) '(e) 'f))))
(cons (car '(a b c)) (cdr '(list e r t y u i o p)))
(eval (null nil))
(list (null t) (eval (eq 4 5)) (< 7 1) (atom '(q s d)) (atom nil) (atom (null nil)))

9-6° Comment traduire les fonctions ((x y) z) ⇨ x et ((x y) z) ⇨ y ?

Réponses (car (car X)) ou bien "caar", puis (car (cdr (car X))) ou "cadar"

9-7° Evaluer :
(- (* (+ 5 2) (- 3 4) (+ 1 2 3)))
(+ (length '(3 4 5)) (car (cddr '(5 2 6))))
(cadar (caar '((((3 7)) 5) (6 1) (2 0))))
(cadr (caddr '((10 9 8) (7 6 5 4) (3 2) (6 5))))

9-8° Evaluer :
(cons 'a (cddar '((c d e))))
(cons (cadddr '(g h k m n)) '(n e f))
(cons (list (car '(r f))) (cddr '(c b a t p)))
(eval (cons '+ (cons 4 (list 5))))
(eval (quote (quote x)))

9-9° Représenter par une arborescence de type arbre généalogique, puis par un arbre binaire la liste : (((a) b c d) e ((f) g) (((h))) (i (j)))

9-10° Construire grâce à "cond" les fonctions booléennes suivantes (a et b ne pouvant prendre que les valeurs vrai ou faux)

(a , b) ⇨ (non a) ou (non b)	(Incompatibilité de Sheffer)
(a , b) ⇨ (non a et b) ou (a et non b)	(exclusion réciproque)
(a , b) ⇨ (non a) et (non b)	(Négation connexe de Peirce)

9-11° Redéfinir en Lisp les fonctions "and", "or", "null".

9-12° Redéfinir en Lisp la fonction "member", telle que (member 'c '(a b c d e)) ⇨ (c d e) et (member 'c '(a b)) ⇨ ()

9-13° Reprendre le problème de Dijkstra : on dispose d'une liste contenant trois couleurs en désordre et en nombres indéterminés, il faut les ranger.

Supposons les couleurs de la Roumanie Bleu, Jaune, Rouge dans une liste L, nous créons trois listes B, J, R avec :

```
(de ranger (L B J R) (cond
                ((null L) (append B J R))
                ((eq (car L) 'bleu) (ranger (cdr L) (cons 'bleu B) J R))
                ((eq (car L) 'jaune) (ranger (cdr L) B (cons 'jaune J) R))
                ((eq (car L) 'rouge) (ranger (cdr L) B J (cons 'rouge R)))
```

Départ avec (ranger '(jaune bleu bleu rouge jaune rouge jaune bleu jaune .....) nil nil nil)

9-14° **La première machine à calculer de Pascal.** Cette machine qui se trouve au musée municipal de Clermont Ferrand possédait 5 roues destinées à additioner des montants exprimés en livres (1 livre = 20 sols), en sols (1 sol = 12 deniers), et en deniers. Simuler cette machine par une fonction "plus" telle que : (plus L1 S1 D1 L2 S2 D2) renvoie la liste (L S D) de la somme exprimée en livre, sols et deniers.

9-15° Ecrire la fonction N puissance P pour P entier relatif

```
(de puissance (N P) (cond
                ((eq P 0) 1)
                ((eq P 1) N)
                ((< P 0) (divide 1 (puissance N (- P))))
                (t (* N (puissance N (- P 1))))))
```

9-16° **Pgcd de deux entiers positifs N et P**

```
(de pgcd (N P) (cond
                ((< N P) (pgcd P N))
                ((eq N P) N)
                ((eq P 0) N)
                (t (pgcd (- N P) P))))
```

9-17° **Nombre de Stirling pour N P, donné par :**
0 si N = P = 0, 1 si N = P = 1, et (1 - N) sti (N - 1 , 1) si P=1,
et enfin sti(N - 1, P - 1) - (N - 1) sti (N - 1, P) dans les autres cas.

9-18° **Combinaisons en utilisant la relation de Pascal.**

```
(de comb (N P) (cond
                ((eq P 0) 1)
                ((eq N P) 1)
                (t (+ (comb (1- N) P) (comb (1- N) (1- P))))))
```

Cette formulation est evidemment très mauvaise puisque les mêmes calculs sont répétés plusieurs fois.

9-19° **Liste obtenue en retirant,**
a) ... la première occurrence d'un élément X d'une liste.
b) ... toutes les occurrences de X dans L.
c) ... un élément X à tous les niveaux de L.

9-20° Exécuter les petites fonctions de [D.Hofstadter]
(de f(n) (if (eq n 0) 1 (- n (g (f (1- n))))))
(de g(n) (if (eq n 0) 0 (- n (f (g (1- n))))))
(de q(n) (if (< n 3) 1 (+ (q (- n (q (1- n)))) (q (- n (q (- n 2)))))))
Etudier ces suites pour $0 \leq n \leq 12$

9-21° Rang d'un élément X dans une liste L.

```
(de rang (X L) (if (eq (car L) X) 1 (1+ (rang X (cdr L)))))
```

Cette fonction ne donne une réponse que si X est vraiment présent dans L, que faut-il faire pour tous les cas ?

9-22° Tête formée par les N premiers éléments d'une liste.

```
(de tete (N L) (cond
        ((null L) nil) ; cas d'une liste vide où  ses N premiers termes seront la liste vide
        ((eq N 0) nil) ; ces deux cas permettent l'arrêt des appels récursifs
        (t (cons (car L) (tete (- N 1) (cdr L))) )))
```

Cette fonction existe sous le nom de "firstn".

9-23° Concaténation de deux listes

```
(de append (L M) (if (null L) M (cons (car L) (append (cdr L) M))))
```

Cette fonction existe sous le nom de "append".

9-24° Intersection de deux listes.

```
(de inter (L M) (cond
        ((null L) nil)
        ((member (car L) M) (cons (car L) (inter (cdr L) M)))
        (t (inter (cdr L) M)) ))
```

9-25° Prédicat d'inclusion d'une liste dans une autre.

```
(de inclu (L M) (cond
        ((null L) t) ; le vide est inclu dans tout ensemble
        ((member (car L) M) (inclu (cdr L) M))
        (t nil) )) ; inutile d'aller voir plus loin
```

9-26° Union ensembliste.

```
(de union (L M) (cond
        ((null L)  M)
        ((member (car L) M) (union (cdr L) M))
        (t  (cons (car L) (union (cdr L) M))) ))
```

9-27° Sous-liste non consécutive exemple pour '(a e t y) et '(a z e r t y u i o p) ⇨ t

```
(de sliste (L M) (cond ; donne vrai ssi L suite extraite de M
        ((null L) t)
        ((null M) nil)
        ((eq (car L) (car M)) (sliste (cdr L) (cdr M)))
        (t (sliste L (cdr M)))))
```

9-28° Différence ensembliste, en déduire le prédicat d'égalité ensembliste.

```
(de diff (L M) (cond
                ((null L) nil)
                ((member (car L) M) (diff (cdr L) M))
                (t  (cons (car L) (diff (cdr L) M))) ))

(de egal (L M) (and (null (diff L M) (null (diff M L))))
```

ou bien : (not (or (diff L M) (diff M L)))
qui donnera le prédicat "ensembles égaux" avec ou sans répétitions.

9-29° Profondeur d'une liste ex. (((a)) (y) h (((g n))) j) ⇨ 4

```
(de prof (L) (if (atom L) 0 (max (1+ (prof (car L))) (prof (cdr L))))))
```

9-30° Union à toutes les profondeurs, ex. (a (s d) (t y (u) ((p)) k) m) ⇨ (a s d t y u p k m)

```
(de aplatir (L) (cond
                ((null L) nil)
                ((atom L) (list L))
                (t (append (aplatir (car L)) (aplatir (cdr L)) )) ))
```

Comparer avec :

```
(de xyz (L) (cond
                ((atom L) L)
                (t  (cons (xyz (car L)) (xyz (cdr L)))) ))
```

9-31° Elimination des répétitions ex. (a z g a s z d g z s) ⇨ (a z g s d)

```
(de elim (L) (cond ; c'est une fonction pouvant rendre service après l'union par exemple
                ((null L) nil)
                ((member (car L) (cdr L)) (elim (cdr L)))
                (t (cons (car L) (elim (cdr L)))) ))
```

9-32° Placer X à la position N dans L sans supprimer d'éléments.

```
(de placer (X N L) (append (tete (- N 1) L) (cons X (nthcdr (- N 1) L)))) ; cas d'impossibilités non examinés
```

9-33° Prédicat d'arborescences semblables ex. (a (f g) j ((y)) l) et (k (g n) f ((p)) h)

```
(de homo (L M) (cond
                ((atom L) (atom M))
                ((atom M) nil)
                ((homo (car L) (car M)) (homo (cdr L) (cdr M)))))
```

Remarque : le "t" est inutile dans la dernière clause, dans la mesure où "cond" retourne l'évaluation de la dernière expression de cette troisième clause au cas où les deux premières sont fausses.

9-34° Nombre de feuilles ex. (r ((t)) y (g h) (j m l) p) ⇨ 9

```
(de nbfeuilles (L) (cond  ((nullL L) 0)
                          ((atom L) 1)
                          (t (+ (nbfeuilles (car L)) (nbfeuilles (cdr L)))))) ; ici aussi le "t" est inutile
```

9-35° Nombres d'arcs

```
(de nbarcs (L) (if (atom L)  0 (+ 1 (nbarcs (car L)) (nbarcs (cdr L)))  ))
```

Essayer sur (a (d f g) (g h (k l m) n ((a)) b (c (d (e)) c)) (((f))) s)

9-36° Traduction des chiffres romains

Cet exemple est destiné à montrer qu'une seule utilisation de la conditionnelle "cond" peut traduire tous les cas de décryptage d'une inscription Z écrite en chiffres romains.

```
(de trad (Z) (cond                  ; 11 cas se présentent, les 7 symboles ...
                (( eq Z 'I) 1)
                (( eq Z 'V) 5)
                (( eq Z 'X) 10)
                (( eq Z 'L) 50)
                (( eq Z 'C) 100)
                (( eq Z 'D) 500)
                (( eq Z 'M) 1000)       ; ... qui sont les 7 valeurs de référence.
                ((atom Z) (trad (explodech Z)))
                ; le cas où on a au moins deux symboles, alors on va le traiter en liste
                ((null (cdr Z)) (trad (car Z)))
                ((< (trad (car Z)) (trad (cadr Z)))
                        (- (trad (cdr Z)) (trad (car Z))))
                ; c'est le cas de IV IX XC..., on décryptera aussi des formes non correctes comme IM pour 999
                (t (+ (trad (car Z)) (trad (cdr Z)) )) ))
```

Cette fonction (récursive) constitue un exemple de "programme", dont l'utilisation se fait en écrivant par exemple (trad 'MCMXCV) suivi de la touche "return". L'évaluation renverra 1995.

9-37° Traduction inverse chiffres arabes ⇨ romains

9-38° Fonction donnant la liste des nombres de billets ou pièces nécessaires pour une somme S payable en 500 200 100 50 20 10 5 2 1 0,5 0,2 0,1 0,05 F

9-39° **Un exemple de filtrage** Deux listes sont équivalentes à ? près, si elles donnent le même chose lorsqu'on leur retire les ? Ainsi (A ? ? B C D ?) et (? A B ? C ? ? D).

Pour créer ce nouveau prédicat, on va tester sur les premiers éléments des deux listes, si l'un d'entre eux est ? alors on avance la lecture d'un cran dans la liste où il se trouve; si les deux sont égaux, on peut avancer simultanément dans les deux listes, sinon elles ne sont pas équivalentes.
Cependant ce qui vient d'être dit dépend de l'existence de ces premiers éléments, c'est pourquoi il faut au préalable s'en assurer, en d'autres termes les tests d'arrêts d'appels récursifs doivent toujours se situer avant les autres.

```
(de equiv (L M) (cond
                ((null L)       (cond   ((null M) t)
                                        ((eq '? (car M)) (equiv L (cdr M)) )
                                        (t nil) ))
                ((null M)       (equiv  M L))   ; on utilise la symétrie
                ((eq '? (car L)) (equiv  (cdr L) M))
                ((eq '? (car M)) (equiv  L (cdr M)))
                ((eq (car L) (car M)) (equiv  (cdr L) (cdr M)) )
                (t nil) ))
```

9-40° Ecrire une autre fonction "filtre", qui pour deux listes L et M formées de lettres, mais où L peut comporter en plus les symboles ? et *, teste si les deux listes sont équivalentes lorsque ? est "unifiable" à tout symbole de M et * à toute sous liste non vide dans M. Exemple (a ? e f * h d ? * b) et (a b e f m n p h d k a z e r t y b) le sont car le premier ? filtre b, le second k, et le premier * filtre (m n p) alors que le second peut s'unifier à (a z e r t y).

```
(de filtre (L M) (cond
                ((null L) (null M)
                ((null M) nil)
                ((eq (car L) (car M)) (filtre (cdr L) (cdr M)))
                ((eq (car L) '?) (filtre (cdr L) (cdr M)))
                ((eq (car L) '*) (or (filtre (cdr L) (cdr M)) (filtre L (cdr M)))) ))
```

Il y a dans la dernière clause un regard en avant, c'est à dire qu'on va tester M jusqu'à son extrémité et revenir en arrière.

9-41° Fonction reverse restituant l'écriture à l'envers d'une liste.

Cet exemple est bien connu pour que nous anticipions sur le chapitre suivant en le faisant grâce à des appels récursifs terminaux (chaque appel constituant le dernier travail de l'appel précédent, il n'y a pas de nécessité de conserver une pile des appels attachée avec des valeurs correspondant à chaque appel, cette récursivité terminale débouche sur une occupation constante de la mémoire)

Prenons le cas de	debut =	(a b c)	et fin =	()
		(b c)		(a)
		(c)		(b a)
		()		(c b a)

On voit qu'en prenant le "car" de la première liste et en le "consant" à la seconde en trois étapes, on a une programmation très claire. Puis (de rev (L) (revbis L nil)) se contente de réaliser l'appel initial. :

```
(de revbis (D F) (if (null D) F (revbis (cdr D) (cons (car D) F))))
```

9-42° Calcul récursif terminal de Cnp

On utilise $C_{n,p} = nC_{n-1,p-1} / p$ quand c'est possible :

```
(de comb (n p r) (if (eq p 0) r (comb (1- n) (1- p) (div (* n r) p)) )) ; Le départ se faisant avec (comb n p 1)
```

9-43° Produire la liste des entiers de 1 à une valeur donnée, ex (jusqua 6) ⇨ (1 2 3 4 5 6).

```
(de jusqua (n) (jusquabis n nil))
(de jusquabis (n L) (if (eq n 0) nil (jusquabis (- n 1) (cons n L))))
```

Voici une version récursive terminale. Le départ se faisant avec (compte n nil)

```
(de compte (n L) (if (eq n 0) L (compte (- n 1) (cons n L)) ))
```

9-44° Répartition d'une somme S entre N personnes aux permutations près.

Pour avoir une formules de récurrence, on décompose en donnant au moins 1F à chacun, ou sinon un au moins obtient 0. Exemple 7 = 7+0+0 = 6+1+0 = 5+2+0 = 4+3+0 = 5+1+1 = 4+2+1 = 3+2+2 soit 7 possibilités.

```
(de rep (S N) (cond
                ((< S 0) 0)
                (( < S 2) 1)
                ((eq N 0) 0)
                (t  (+ (rep (- S N) N) (rep S (- N 1)))) ))
```

Remarque : les trois premières conditions peuvent être remplacées par les deux suivantes : N si S = 1, et 1 si S = 1

9-45° Nombre de surjections d'un ensemble de n éléments vers un ensemble de p éléments $S(n, p) = \sum(-1)^{p-i}.(i)^n C_p^i$ pour $0 \le i \le p$

```
(de comb (n p) (if (eq p 0) 1 (* (divide n p) (comb (1- n) (1- p)))))
(de surj (n p) (surj1 n p 1 0 p))      ; surj se contente d'appeler "surjbis", qui, elle, est récursive
(de surjbis (n p s res i) (if (eq i 0) res
                (surjbis n p (- s) (+ res (* s (pui i n) (comb p i))) (1- i))))
(de pui (x n) (if (< n 1) 1 (* x (pui x (- n 1.0)))))
```

Remarque : une formule de récurrence permet une autre écriture :
$S(n, p) = p[S(n-1, p-1) + S(n-1, p)]$ sauf si $n < p$ où c'est 0 et si $p = 1$ c'est 1, on a alors $S_{p, p} = p!$, mais l'arbre développé par l'utilisation de cette relation est exponentiel.

9-46° Nombre de partitions de n éléments en p parties non vides,
$part(n, p) = S(n, p) / p!$ sont les nombres de Stirling de seconde espèce.

9-47° Nombre de dérangements : substitutions de n éléments sans aucun point fixe $d_n = n!\sum(-1)^p / p!$ pour $0 \le p < n$

9-48° Simulation de la loi de Poisson de moyenne m, $P(X = k) = e^{-m} m^k / k!$ En considérant la fonction lisp (random a b) qui renvoie un entier réparti uniformément au hasard entre les entiers a et b-1, on peut simuler une loi de Poisson de moyenne m en admettant que le plus grand entier n tel que $\prod_{1 \le i \le n} X_i < \exp(-m)$ suit une loi P(m) si les X_i sont des variables aléatoires indépendantes de même loi uniforme dans [0, 1].

```
(de poisson (m) ; renvoie un entier aléatoire de loi de Poisson avec la moyenne m
                (poissonbis (exp (- m)) 0 (divide (random 0 100) 100)))

(de poissonbis (ex n x) ; vérifie s'il faut continuer à tirer des nombres au hasard
                (if (< x ex) n ; cas où le produit des x est parvenu à dépasser ex = exp (-m)
                (poissonbis ex (1+ n) (* x (divide (random 0 100) 100))) ))
```

9-49° Un Forth écrit en Lisp : on veut évaluer une formule mathématique écrite en notation suffixée comme par exemple 4*(3 + 7) - 2 qui serait (4 3 7 + * 2 -) à l'aide d'une pile de valeurs déjà calculées.

La lecture se faisant au fur et à mesure ou de gauche à droite en empilant les valeurs numériques, et en dépilant chaque fois que l'on pointe sur un opérateur. Pour (3 7 + 4 2 + *) on aura successivement :

pile	liste des entrées
()	(3 7 + 4 2 + *)
(3)	(7 + 4 2 + *)
(7 3)	(+ 4 2 + *)
(10)	(4 2 + *)
(4 10)	(2 + *)
(2 4 10)	(+ *)
(6 10)	(*)
(60)	()

```
(de calcul (P L) (cond     ;P est la pile, L, la liste constituée par la formule
                ((null L) (car P))
                ((member (car L) '(+ - * /))
                        (calcul (cons (eval (list (car L) (car P) (cadr P))) (cddr P)) (cdr L)))
                ((member (car L) '(sin cos ln exp))
                        (calcul (cons (eval (list (car L) (car P))) (cdr P)) (cdr L)))
                ((numberp (car L)) (calcul (cons (car L) P) (cdr L))) ))
```

9-50° Fonction d'Ackerman

f (x, y) = si x = 0 alors y + 1 si y = 0 alors f (x - 1, 1) sinon f (x - 1, f (x, y - 1))

9-51° Fonction de Mac Carthy

f (x, y) = si x > 100 alors si y = 1 alors $m^{(y)}$ (x), m composée y fois avec elle-même
sinon f(x - 10, y - 1)
sinon f(x + 11, y + 1)

Avec m (x) = si x > 100 alors x - 10 sinon f(f(x + 11))

Développer un tableau des valeurs de f pour $0 \leq x \leq 101$ et $1 \leq y \leq 11$ (toutes ne sont pas nécessaires).

Construire un programme itératif du calcul de f.

9-52° Le Jeu de Nim ou de Marienbad (Màriànské Làznê)

Initialement on donne un nombre n de rangées d'allumettes d'importances quelconques. A tour de rôle chacun des deux joueurs choisit une des rangées et retire une partie ou tout de cette rangée. Celui qui tire la dernière allumette a perdu. L'algorithme à programmer pour jouer est le suivant :

Si une seule rangée a plus d'une allumette il faut laisser un nombre impair de rangées d'une allumette.

S'il reste un nombre impair de rangées d'une allumette et pas d'autres rangées, on perd.

S'il reste un nombre pair de rangées d'une allumette, on gagne.

Sinon, on écrit le nombre binaire d'allumettes de chaque rangée et on somme chaque colonne modulo 2.

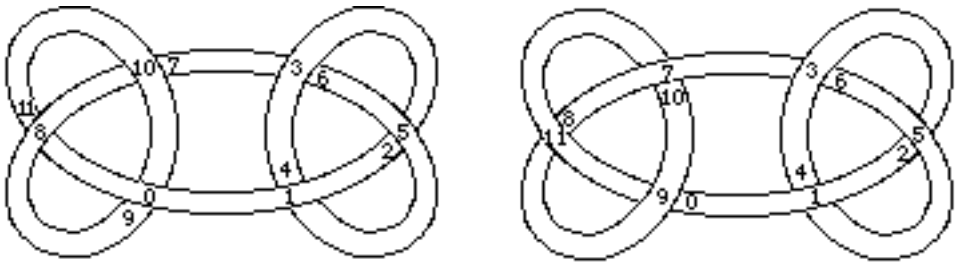
S'il n'y a que des 0 on va perdre sauf si l'adversaire fait une erreur.
Sinon on retranche dans une rangée un nombre tel que le résultat soit que des 0 (nombre pair en chaque colonne) car cela est toujours possible.

Si les p rangées sont de longueur l_1, ... , l_p et si $\otimes$ désigne le "ou" exclusif, on a par exemple $1101 \otimes 0100 \otimes 1100 = 0101$.

Posons $n = l_1 \otimes l_2 \otimes ... \otimes l_p$, il existe au moins un k tel que $l_{k'} = (l_k \otimes n) < l_k$, il suffit de prendre un k tel que l_k et $l_{k'}$ aient 1 au rang le plus élevé de $l_{k'}$. On va donc jouer dans la ligne k de façon à ramener l_k à $l_{k'}$ et alors : $l_1 \otimes ... \otimes l_{k'} \otimes ... \otimes l_p = l_1 \otimes ... (l_k \otimes n) ... \otimes l_p = n \otimes n = 0$ ce qui est une situation perdante pour l'adversaire.

9-53° Noeuds isotropes : si superposables.

Exemples avec 6 croisements, le noeud simple et le noeud de vache :



Construire une fonction indiquant si deux noeuds sont ou non isotropes. On pourra représenter un noeud par une liste de couples où l'abscisse est une marque sur la corde qui est en dessous de la marque de l'ordonnée. Ainsi ((9 0) (4 1) (2 5) (6 3) (7 10) (11 8)) à gauche et ((0 9) (4 1) (2 5) (6 3) (10 7) (8 11)) à droite.

9-54° **Temps de remontée d'un plongeur,** pour une profondeur x ≤ 100 m, sachant qu'il doit faire un palier de 100 secondes à 30m puis de 30m à 0, des paliers tous les 3m, le temps diminuant de 10s à chaque fois. Les paliers au-dessus de 30m étant par contre augmentés de 25s tous les 10m, ainsi par exemple 150s à 50m. La vitesse de remontée est supposée de 1 m/s

Définissons d'abord la "demisous"(traction), renvoyant 1 si X > Y, et 0 sinon.

```
(de demisous (X Y) (if (> X Y) 1 0))
```

E renvoie la partie entière de X, sauf si celui-ci est déjà entier auquel cas on renvoie X-1, cette fonction est utile pour prendre en compte les temps d'arrêt, il n'y a , en effet, pas de palier à 3 m si le départ est juste situé à 3 m.

```
(de E (X) (if (eq (truncate  X)  X) (-  X  1) (truncate  X))) ; truncate est en Lisp la partie entière d'un réel

(de temps1 (X) (+ X ; Temps1 résulte d'un calcul direct, temps2 est récursif.( "divide" est la division entière )
                  (* 25 (demisous X 30)  (-  (/ (*  (1+ (E (divide X 10)) ) (+ (E ( divide X 10))2 )) 2) 6 ))
                  (* 5 (E ( divide (min X 30) 3)) (1+  (E ( divide (min X 30) 3)) ))))
```

Temps1 est la somme des paliers calculée par la fameuse formule 1+2+3....=... , on peut constater son manque de lisibilité, temps2 fait par contre appel à la fonction palier qui elle, ne fait que transcrire les données.

```
(de palier (N) (cond      ((eq N 30) 100) ; Palier ne marche que pour .... 60 50 40 30 27 24 21 18 ..
                          ((< N 30) (- (palier (+ N 3)) 10)) ; Il est étrangement récursif, pourquoi ?
                          (t (+ (palier (- N 10))  25)) ) )


(de temps2 (X) (cond      ((< X 3) X)
                          ((< X 30) (cond ;          Rappelons que 1+ et 1- sont des fonctions prédéfinies
                                    ((eq (mul 3  X) X) (1+ (temps2 (1-  X))))
                                    (t (+ (modulo  X 3) (palier (mul 3 X)) (temps2 (mul 3 X))))))
                          ((eq (mul 10 X) X) (1+ (temps2 (1- X))))
                          (t  (+ (modulo X 10) (palier (mul 10 X)) (temps2 (mul 10 X))))))

(de mul (B X) (- X (modulo X B))) ; mul est le plus grand multiple de B inférieur à X
```

Exemples de résultats de temps suivant les profondeurs : 30/480 39/589 40/590 41/716 50/525 63/1063

9-55° Comparer expérimentalement les temps d'exécution de fonctions temps1 et temps2, pour des valeurs communes.

9-56° Exercice n'ayant rien à voir, en quelles écritures sont ces chiffres ?