CHAPITRE 11

COMPLEMENTS

Ce chapitre n'est pas nécessaire à la compréhension de la suite, il donne cependant des compléments sur certaines fonctions de Lisp destinées à des écritures itératives, à des affectations locales ou dans des listes de propriétés, mais aussi quelques curiosités fonctionnelles très difficiles sinon impossibles à réaliser dans les langages traditionnels.

L'affectation "set" (set 'X 4) ou bien (setq X 4) ⇨ 4 permet d'affecter la valeur 4 à X
(set X Y) ⇨ Y permet d'affecter la valeur de Y à la valeur de X contrairement à setq qui est une contraction de "set quote". Les commandes suivantes aident à comprendre ce qui se passe :

(setq X 'A) ⇨ A (X est affecté par A)
X ⇨ A (X suivi de "enter" renvoie la valeur de X)
(set X 5) ⇨ 5 (On affecte 5 à la valeur de X, c'est donc A qui vaudra 5)
X ⇨ A (si on demande la valeur de X, c'est A qui est donné comme réponse)
A ⇨ 5 (la valeur de A a en effet été modifiée)
(setq Y A) ⇨ 5 la valeur de Y est désormais 5
(setq A 7) ⇨ 7 la valeur de A est désormais 7, alors si on demande les valeurs de X Y A , on obtient respectivement A, 5, 7.

Setq permet de grouper les affectations en déclarant par exemple :
(setq X 1 Y (+ 4 5) Z (cube 5) L '(a b c) M '(d e) S1 's S2 'm) pour des initialisations diverses.

Définition de fonction n'évaluant pas ses arguments

Grâce à "df", une fonction ne calculera pas la valeur des arguments qu'on lui donne au début de son traitement, mais ne le fera que par nécessité, c'est à dire qu'au moment où ils apparaîtront dans le corps de la définition. Ceci peut être utile à la définition de fonction assez complexe comme on trouvera dans l'exemple sur les intégrales multiples.

Exemple, on veut définir un "si ... alors ... sinon ..." en supposant que seul la "cond" existe, on doit le faire par :

(df si (test alors sinon) (cond ((test alors) (t sinon)))

Les expressions "alors" et "sinon" figurant dans la liste des paramètres ne doivent absoluement pas être évaluées avant que le "test" ne le soit.

Itérations

Il existe dans les différents dialectes de Lisp tous les équivalents de "repeat", "while" etc, mais leur usage n'est pas nécessaire au début. On dispose de :

(while test éval$_1$ éval$_2$ ... éval$_n$)
(until test éval$_1$ ... éval$_n$)
(repeat n éval$_1$ éval$_2$ ... éval$_n$)

Ainsi :
(set 'x '(a b c d e f)) puis au choix :
(while x (set 'x (cdr x)) (print x))
(until (null x) (set 'x (cdr x)) (print x))
(repeat 5 (set 'x (cdr x)) (print x))

Fonctionnelles apply et mapcar

Pour itérer, il est plus intéressant de connaître deux "fonctionnelles" :
(mapcar 'fonction 'liste) ⇨ liste où la fonction s'est exercée sur tous les éléments. Par exemple, si on veut faire agir la fonction successeur qui est notée par les deux caractères accolés 1+ , sur une liste d'entiers, on aura:

(mapcar '1+ '(3 -2 1 0 4)) ⇨ (4 -1 2 1 5)

Soit maintenant la fonction "car" (premier élément) agissant sur tous les termes d'une liste de listes:

(mapcar 'car '((a b) (c) (a d e) (b d)) ⇨ (a c a b)

(apply 'fonction 'liste) ⇨ résultat de la fonction sur tous les éléments en bloc:

(apply '+ '(2 5 1 7)) ⇨ 15

(apply 'append '((a b) (f b c) (a c) (b))) ⇨ (a b f b c a c b)

On est donc assez loin des numérotations besogneuses.

Exemple du plus grand rapport dans une liste de nombres positifs :
(de amplitude (L) (/ (apply 'max L) (apply 'min L)))

Exemple de la profondeur d'une liste :
(de prof (S) (if (atom S) 0 (1+ (apply 'max (mapcar 'prof S)))))
On dispose en outre de "funcall" qui a le même effet que "apply" :
(apply '+ '(1 2 3 4)) et (funcall '+ 1 2 3 4) sont identiques.

Définitions de fonctions anonymes

(lambda P L) permet une définition anonyme, exemple: si on veut évaluer l'action de la fonction carré sur 5, mais en redéfinissant le carré uniquement pour cette occasion :

(eval ((lambda (x) (* x x)) 5)) ⇨ 25

Exemple (mapcar (lambda (x) (equal x 5)) '(0 5 3 2 5 7)) ⇨ (nil t nil nil t nil)
Exemple de la présence de X dans L à tous les niveaux :

```
(de present (X L) (cond
                  ((equal X L) t)
                  ((atom L) nil)
                  (t (apply 'or (mapcar (lambda (e) (present X e)) L)) )))
```

Exemple du compte du nombre d'occurrences de A dans L :

```
(de compte (X L) (cond
                  ((equal X L) 1)
                  ((atom L) nil)
                  (t (apply '+ (mapcar '(lambda (e) (compte X e)) L)) )))
```

Listes de propriétés

Des listes de propriétés peuvent être définies en lisp sous la forme (objet valeur attribut).
Exemple (Jean 32 age) ou encore (Irma (valérian laureline) enfants)
Les fonctions disponibles sont :
(plist ob) ⇨ (at_1 v_1 at_2 v_2 ... at_i v_i)
(plist ob '(at_1 v_1 ... at_i v_i)) permet l'affectation d'une liste de propriétés pour ob

(putprop ob v at) met la valeur v pour la propriété "at" de l'objet "ob" (et remplace l'ancienne valeur s'il y en avait une).
(getprop ob at) ⇨ valeur de l'attribut "at" pour l'objet "ob"
(addprop ob v at) rajoute en tête de la liste une nouvelle propriété
(remprop ob at) retire de la liste l'attribut at et ses valeurs

Exemple :
Si on définit toute une famille avec les attributs "pere" et "mere" pour chaque individu, on aura:

```
(de ancetres (x)
            (if x (cons x (append (ancetres (getprop x 'pere)) (ancetres (getprop x 'mere))))))
```

Un exemple très fort permet de voir maintenant un calcul comme l'attribution d'une valeur pour un attribut fonction à un objet qui ici est un nombre. Putprop fait un calcul et un stockage du résultat ce qui fait qu'une définition récursive peu intéressante peut le devenir.

```
(de fib (n) (or     ; fib sera le résultat de l'évaluation de la fonction "or" sur ses 3 arguments
                    (>= 1 n)          ; écriture dictée par le fait que (>= a b) renvoie a ou bien nil
                    (getprop 'fib n)          ; cas où fib n) a déjà été calculé une fois,
                                              ; il est donc restitué immédiatement
                    (putprop 'fib (+ (fib (1- n)) (fib (- n 2))) n)
                    ; car (putprop obj val att) renvoie toujours val
                    )) ; limité à (fib 22) = 28657 en entiers, ce qui n'enlève rien à la méthode
```

Variables locales

La fonction "let" permet de définir une liste de variables locales avec leurs initialisations de la façon suivante :

```
( let           ( (x 0) (y 1) (z 2) ....)       ; est une liste de couples (nom initialisation)
                ( .... ) ( .... ) ..... ; liste d'expressions à évaluer où on peut avoir notamment
                ( .... ) ( .... ) ..... ; des réaffectations comme (set 'x (+ x 2)) ...
                ( .... ) ( .... ) ( .... ) )
```

Let renvoie la valeur de sa dernière expression.

11-1° Evaluer successivement :

```
( * (apply  '*  '(3  2  1  6)) (apply  '+  '(9  7  10  3  4  1  0  2)) )
(apply  (quote +) (mapcar (quote length) (quote ((a b) (c) (d e f) () ))))
(mapcar  '+  (apply  'append '( (5  8) (6  2  1  3) (7  0  4))
```

11-2° Que fait (mapcar (lambda (x) (length (explodech x))) '(A la mer nous avons trempé crûment quelques gentilles Allemandes stupidement bouleversées)) ?
Explodech produisant la liste des lettres d'une chaîne.

Réponse : (1 2 3 4 5 6 7 8 9 10 11 12)

11-3° Réécrire la fonction ret2 (retirer toutes les occurrences d'un objet au premier niveau d'une liste) du chapitre 10 au moyen de mapcar.

```
(de ret2 (N L) (mapcar (lambda (X) (append (firstn (1- N) X) (nthcdr N L))) L))
```

11-4° Construire la loi binomiale $p(X = k) = C_n^k p^k (1-p)^{n-k}$ itérativement.

11-5° Comment définir le min et le max d'une liste de nombres au moyen de deux variables locales ?

11-6° Redéfinir la profondeur d'une liste grâce à "mapcar", "max" et "lambda".

11-7° Ecrire une fonction donnant la liste des décimales de 1/n pour 1 < n < 50, essayer de séparer les périodes de ce développement par des marques telles que /. [Wirth]

11-8° **Réaliser la division des polynômes suivant les puissances décroissantes**
$A = BQ + x^{k+1} R$ est toujours possible si B est de valuation 0 pour A, B, k fixés.

On représente les polynômes par la liste de leurs coefficients suivant les puissances croissantes.
Exemple : le polynôme $2 - 3x + 4x^2 - 5x^4$ divisé à l'ordre 3 par $2 + x - 4x^3 + x^4 - x^5$ donne un quotient égal à $1 - 2x + 3x^2 + 0{,}5x^3$ et un reste $x^4(-14{,}5 + 15x - 3x^2 + 2{,}5x^3 + 0{,}5x^4)$
ou encore 1 divisé par 1 - x à l'ordre 9 donne (1 1 1 1 1 1 1 1 1).

```
(de mulscal (P c) (mapcar (lambda (x) (* x c )) P)) ; est la multiplication scalaire
(de difpol (P Q) (cond                ; différence polynômiale
                  ((null Q) P)
                  ((null P) (mapcar '- Q))
                  (t (cons (- (car P) (car Q)) (difpol (cdr P) (cdr Q)))) ))

(de divbis (A B Q p k) (cond       ; la fonction principale
                  ((< k p) (list 'quotient (reverse Q) 'reste A 'fois 'x p)) ; stoppe dès que p dépasse l'ordre k
                  (t (divbis (cdr (difpol A (mulscal B (divide (car A) (car B)) ))) B
                        (cons (divide (car A) (car B)) Q) (+ 1 p) k))))

(de divpolcroiss (A B k) (divbis A B nil 0 k))
```

11-9° Construire le tri par extaction au moyen d'une boucle "until" en cherchant en même temps le min et le max d'une liste de nombres.

11-10° **Simuler une loi normale de moyenne m et d'écart-type s,** par la somme du tirage d'une dizaine (cela peut suffire) de variables aléatoires indépendantes de même loi (théorème central-limite).

Solution : on prendra n fois (random 0 100) de moyenne 50 et d'écart-type $50 / \sqrt{3}$, leur moyenne $M = \sum / n$ tend vers une loi normale de moyenne 50 et d'écart-type $50 / \sqrt{3n}$. C'est pourquoi, en prenant n = 12, on aura un écart-type 25 / 3, et donc s[3(M - 50) / 25 + m] qui se simplifie légèrement, suit la loi désirée. (On pourra réaliser la simulation de la loi du chi-deux à n degrés de liberté, en prenant la somme des carrés de n variables aléatoires indépendantes de loi normale centrée réduite.)

```
(de gauss (m s) ; renvoie un réel aléatoire de loi normale d'espérance m et d'écart-type s
                  (let ((som 0))     (repeat 12 (set 'som (+ som (random 0 100))))
                        (* s (+ (- m 6) (divide som 100))) ))
```

11-11° **Quel est le dernier résultat dans les séquences d'évaluations :**

```
(setq  x  4)  (eval  x)
(setq  x  5)  x
(setq  x  6)  (setq  y  'x)  y
(setq  x  7)  (setq  y  'x)  (set  y  8)  x
(setq  x  8)  (setq  y  'x)  (setq  z  'y)  (list  z  (eval  z) (eval (eval z) ) )
(setq  m  '(a b c))  (eq  m  '(a b c))
(setq  m '(a b c))  (eq  m  (cons (car m) (cdr m)))
(setq  m  '(a b c))  (equal  m  '(a b c))
(setq  m '(a b c))  (equal  m  (cons (car m) (cdr m)))
```

11-12° Quelle est la différence entre les fonctions définies par :
(de succ (N) (+ N 1)) et (de increm (N) (set N (+ N 1))) ?

Recommencez la lecture du chapitre si vous ne pouvez y répondre.

11-13° Donner les 100 plus petits entiers (dans l'ordre croissant) de l'ensemble M défini par : 1 est dans M et si n est dans M alors 2n+1 et 3n+1 le sont aussi, M ne contenant pas d'autre nombre. [Wirth]

11-14° Effectuer la transposition, la multiplication des matrices, et le produit scalaire des vecteurs. Une matrice étant la liste de ses colonnes, un vecteur étant une colonne seule.

```
(de transpo (A) (if (null (car A)) nil (cons (mapcar 'car A) (transpo (mapcar 'cdr A)))))

(de mult (A B) (multer (transpo A) B)) ;  produit de A par B si A a autant de colonnes que B a de lignes

(de multer (TA B) (multbis TA TA B nil nil))

(de multbis (TA RTA RB col prod) (cond
        ((null RB) (reverse prod))
        ((null RTA) (multbis TA TA (cdr RB) nil (cons (reverse col) prod)))
        (t (multbis TA (cdr RTA) RB (cons (scal (car RTA) (car RB)) col) prod))))

(de scal (V1 V2) (scalbis V1 V2 0))

(de scalbis (V1 V2 R) (cond
        ((null V1) R)
        ((null V2) R)
        (t (scalbis (cdr V1) (cdr V2) (+ R (* (car V1) (car V2)))))))

(mult '((0 2 1) (3 -2 0)) '((4 0) (1 -1) (3 2) (-1 0))) ; pour essayer
= ((0 8 4) (-3 4 1) (6 2 3) (0 -2 -1))
```

11-15° Créer la fonction intégrale par la somme de Darboux, et extension aux intégrales multiples, en utilisant "df".

a) Cas où on donne directement une expression (f) comportant une variable (x ou autre)

```
(de integrale (f x a b n) ; calcule l'intégrale de f pour la variable x en découpant en n
    (let        ((S 0) (h (divide (- b a) n)) (th a))        ; déclaration de 3 variables locales
                (until (> th b) (set 'S (+ S (eval (subst th x f)))) (set 'th (+ th h)))
                (* S h)))
```

Exemple (integrale '(* x x) 'x 0 3.14 100) donne 1.999861
(integrale '(sin (* 2 x)) 'x 0 1.57 100 donne 0.9999305

b) Solution plus générale : cas où on fait appel à une fonction déjà définie

```
(df integrale (f a b n) ; intégrale de la fonction f d'une variable, entre a et b en découpant en n
    (let        ((S 0) (h (divide (- b a) n)) (th a))
                (until (> th b) (set 'S (+ S (funcall f th))) (set 'th (+ th h)))
                (* S h)))
```

c) L'intérêt est de pouvoir calculer des intégrales doubles ou triples (l'appel par nécessité est alors essentiel), par exemple : (de f (x y) (+ (* 3 x) (* 5 y)))
On a par exemple l'intégrale double de f sur [3, 5]*[1, 2] par :
(integrale (lambda (y) (integrale (lambda (x) (f x y)) 3 5 20)) 1 2 20)
= 4.299749E1 (au lieu de 39 mais c'est une approximation sur un découpage assez grossier)
(integrale (lambda (x) (integrale (lambda (y) (f x y)) 1 2 20)) 3 5 20))
La réponse est identique conformément au théorème de Fubini.

11-16° Le recuit simulé [Kirkpartrick 83], [Aarts Korst 89], [Reeves 93]. Cette méthode consiste à obtenir par itérations successives le minimum absolu d'une fonction, elle est inspiré d'une technique de refroidissement consistant à accepter dans certains cas une remontée de la fonction (pour ne pas descendre trop vite vers un minimum local). En thermodynamique la probabilité d'avoir une augmentation d'énergie ΔE (la fonction que l'on veut minimiser suivant la loi de Boltzmann) est $e^{-\Delta E/\theta}$. On accepte un état voisin s_1 augmentant la fonction, dans la mesure où la probabilité ci-dessus est décroissante suivant ΔE. Le paramètre de contrôle (la température) va décroître, ce qui fait que pour un même ΔE, la probabilité d'accepter une remontée diminue suivant le refroidissement. Plus précisément :
a) Choisir un état initial s_0
b) Faire le "palier" consistant à répéter n fois
Chercher un voisin s_1 de s_0, on calcule $\Delta = f(s_1) - f(s_0)$
Si $\Delta < 0$ alors $s_0 \leftarrow s_1$
Si $\Delta > 0$ alors on accepte la même affectation de s0 avec une probabilité $e^{-\Delta/\theta}$
c) On abaisse la température θ et on réitère ces paliers jusqu'à un test d'arrêt défini par le problème.
La difficulté de cette méthode, qui par ailleurs donne de bons résultats, réside dans la détermination des paramètres. On peut montrer la convergence sous certaines conditions reliant n et la loi de décroissance de la température.
La température initiale peut être déterminée par la probabilité (ex. 1/2) d'accepter une hausse au cours du premier palier, si m est la valeur moyenne de ces hausses, alors $\theta_0 = m / \ln(2)$. Le nombre d'itérations sur chaque palier peut être de l'ordre de la centaine, la loi de décroissance de la température est généralement prise comme une suite géométrique telle que $\theta_{n+1} = 0{,}9\ \theta_n$

Solution :

```
(de refroidir (th) (* 0.9 th)) ; th sera la "température"
(de recuit (s phi n th eps) ; s point de départ pour la fonction phi, n : nb d'itérations, eps : seuil
        (recuitbis s (funcall phi s) (palier n s phi th) phi  n th eps))
(de recuitbis (s0 mini co phi n th eps) (prin co)
        ; co est le couple (s phi(s)) issu du palier partant de s0
        (if     (< (abs (- mini (cadr co))) eps) co
                (recuitbis (car co) (if (< (cadr co) mini) (cadr co) mini)
                        (palier n (car co) phi (refroidir th)) phi n (refroidir th) eps)))
(de palier (n s phi th) ; renvoie le nouvel état "voisin" de s lorsque la "température" th est constante
        (let ((co (list s (funcall phi s))) (s1 0) (m0 0) (m1 0)) ; palier renvoie le couple (s phi(s))
                (repeat n (set 's1 (voisin (car co))) (set 'm1 (funcall phi s1)) (set 'm0 (cadr co))
                        (set 'co (cond  ((< m1 m0) (list s1 m1))
                                        ((< (random 0 100) (* 100 (exp (- (divide (- m1 m0) th)))))
                                                (list s1 m1))
                                        ( t co))))  co))

(de voisin (x) (+ x (divide (- (random 0 100) 50) 1000)))
```

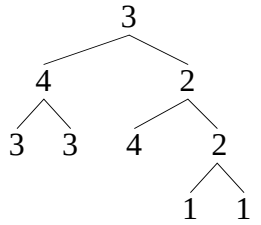
Pour tester, on prend une fonction polynôme de degré 6, calculée par Hörner, dont on connaît à l'avance les minimums. Cette fonction est particulièrement délicate à cause de ses minimums très proches -39,5833; -38,25 et -39,5839 respectivement en 1, 3 et 5.

```
(de psi (x) (* x (+ (- 120) (* x (+ 137 (* x (+ (- 75) (* x (+ 21.25 (* x (- (divide x 6) 3))))))))))
```

11-17° Méthode Tabou [Glover 86]. On part de la même façon d'une "solution" initiale quelconque x. A chaque point courant x du domaine de définition du problème, on cherche grâce à une discrétisation du problème, le meilleur voisin non tabou de x (ce sera le point suivant). La liste des "tabous" est simplement constituée d'états antérieurs sur lesquels on ne souhaite pas revenir. Cette liste doit avoir une taille fixe et chaque état n'y reste donc qu'un nombre fini d'itérations. On la gère en file : chaque mise-à-jour y place un nouveau venu et en retire le plus ancien.

11-18° Le mobile du Professeur Cornuejols : soit (p mg md) un mobile formé d'un poids p central et de deux "sous-mobile" mg et md à gauche et à droite. Construire une fonction renvoyant le poids total si le mobile est équilibré ou nil dans le cas contraire.

Exemple de mobile équilibré de poids 23 :




```
(de poids (L) (if (numberp L) L (poidsbis (car L) (poids (cadr L)) (poids (caddr L))) ))
(de poidsbis (m p q) (cond
                ((null p) nil)
                ((null q) nil)
                ((eq p q) (+ m p q))
                (t nil)))
```

On pourra toujours généraliser avec des mobiles ternaires où intervient la longueur des barres, l'équilibre n'étant assuré que si on a l'égalité des moments, on représentera M = (poids-central ((dist-gauche) mobile-gauche) ((dist-droite) mobile-droite))
Solution avec variable locale :

```
(de poids (L) (if (atom L) L
                (let    ((pg (poids (cadr L))))
                        (and    pg      ; notez l'utilisation astucieuse du "and"
                                (eq pg (poids (caddr L)))
                                (+ (car L) (* 2 pg))) )))
```

11-19° Composition de trois fonctions

```
(de comp (f g h) (list 'lambda '(x) (list f (list g (list h 'x)))))
```

11-20° Construire et représenter les fonctions polynômes convergent uniformément vers n'importe quelle fonction continue sur [a, b] (théorème de Stone-Weierstrass).
On construira pour cela la fonctionnelle Stone (f, a, b, n) $\rightarrow P_n$ défini sur [0, 1] par le polynôme $P_n(t) = \sum f(a + k(b-a)/n)B_{n,k}(t)$ pour $0 \leq k \leq n$ où $B_{n,k}$ est le polynôme de Bernstein $C_n^k t^k (1-t)^{n-k}$.

11-21° Une fonction dont le nombre d'arguments est variable. Si la version de Lisp utilisée possède la fonction "max" pour deux ou un nombre quelconque d'arguments, réécrire une fonction "maxi" ayant des arguments entiers, et retournant l'élément maximal.

```
(de maxi L (if   (null (cdr L)) (car L)     ; ici L désigne cette fois la liste des paramètres
                 (maxibis (car L) (cdr L))))
(de maxibis (x L) (cond  ((null L) x)
                         ((< x (car L)) (maxibis (car L) (cdr L)))
                         (t (maxibis x (cdr L)))))
```

Exemples (maxi 4 2 5 8 6 3) $\rightarrow$ 8 et (maxi 4 2) $\rightarrow$ 4

11-22° Redéfinir la fonction "list" avec une liste d'arguments en nombre variable.

```
(de list L (ifn (null L) (cons (car L) (apply 'list (cdr L)))))
```

11-23° Redéfinir la fonction "mapcar" grâce à "eval".

```
(de mapmoi (f L) (if (null L) L (cons (eval (list f (list 'quote (car L)))) (mapmoi f (cdr L)))))
(mapmoi 'car '((1 2 3) (4 5 6) (7 8 9) (2 3) (5 6) (8 9) (3 0) (6 0) (9 0))) = (1 4 7 2 5 8 3 6 9)
```

11-24° Redéfinir la fonction "apply"

11-25° Recréer la boucle "for" avec un incrément facultatif

```
(de substituer (X Y L) (cond ; réalise une substitution de X par Y à tous les niveaux de L
                ((null L) L)
                ((eq X L) Y)
                ((atom L) L)
                (t (cons (substituer X Y (car L)) (substituer X Y (cdr L))))))

(de pour L  (cond          ; L de la forme (identificateur départ arrivée incrément expression)
                ((< (caddr L) (cadr L)) nil)
                (t (eval (substituer (car L) (cadr L) (car (last L))))
                        (apply 'pour (mcons (car L) (+ (cadr L) (if (null (nth 4 L)) 1 (cadddr L))) (cddr L))) )))
```

(pour 'x 1 10 3 '(print x '--)) ; donne 1--4--7--10-- sur 4 lignes
(pour 'x 1 5 '(print '"carré(" x '")= " (* x x))) ; donne la liste des carrés de 1 à 5

11-26° Un nombre variable de boucles (mais où va-t-il chercher tout ça ?) Il s'agit de définir une fonction "boucles" qui va évaluer son premier argument P (un programme) pour un ensemble d'indices I (en nombre variable) chacun allant d'un début à une fin qui lui sont propres.

```
(de boucles (P I D F) (cond ;  P est une expression à évaluer, I une liste d'indices,
                ; D la liste de leurs valeurs initiales, F leurs valeurs finales
                ((null I) (eval P)) ; fin de la boucle supérieure
                ((> (car D) (car F)) nil); première boucle achevée
                (t      (boucles (substituer (car I) (car D) P) (cdr I) (cdr D) (cdr F))
                        (boucles P I (cons (1+ (car D)) (cdr D)) F) )))
```

(boucles '(prin a b c '/) '(a b c) '(0 0 0) '(3 3 3)) (print) ; donne les nombres à 3 chiffres < 4
000/001/002/003/010/011/012/013/020/021/022/023/030/031/032/033/100/101/102/103/110/111/112/113/120/121/122/123/130/131/132/133/200/201/202/203/210/211/212/213/220/221/222/223/230/231/232/233/300/301/302/303/310/311/312/313/320/321/322/323/330/331/332/333/

On rajoute un compteur x
(set 'x 0) (boucles '(prin (set 'x (1+ x))'-> a b c d '/) '(a b c d) '(0 0 0 0) '(1 2 1 2)) (print)

1->0000/2->0001/3->0002/4->0010/5->0011/6->0012/7->0100/8->0101/9->0102/10->0110/11->0111/12->0112/13->0200/14->0201/15->0202/16->0210/17->0211/18->0212/19->1000/20->1001/21->1002/22->1010/23->1011/24->1012/25->1100/26->1101/27->1102/28->1110/29->1111/30->1112/31->1200/32->1201/33->1202/34->1210/35->1211/36->1212/

Les triadiques de Cantor limité à 4 décimales (il s'agit des nombres n'ayant que 0 ou 2 dans leur développement décimal en base 3

(boucles '(prin '"0." (* 2 a) (* 2 b) (* 2 c) (* 2 d)'/) '(a b c d) '(0 0 0 0) '(1 1 1 1)) (print)

0.0000/0.0002/0.0020/0.0022/0.0200/0.0202/0.0220/0.0222/0.2000/0.2002/0.2020/0.2022/0.2200/0.2202/0.2220/0.2222/

11-27° Graphe, construire le graphe d'une fonction f comme une fonction de f, sachant que l'écran est 500*250 et que l'on dispose de (line-to x y) et (move-to x y).

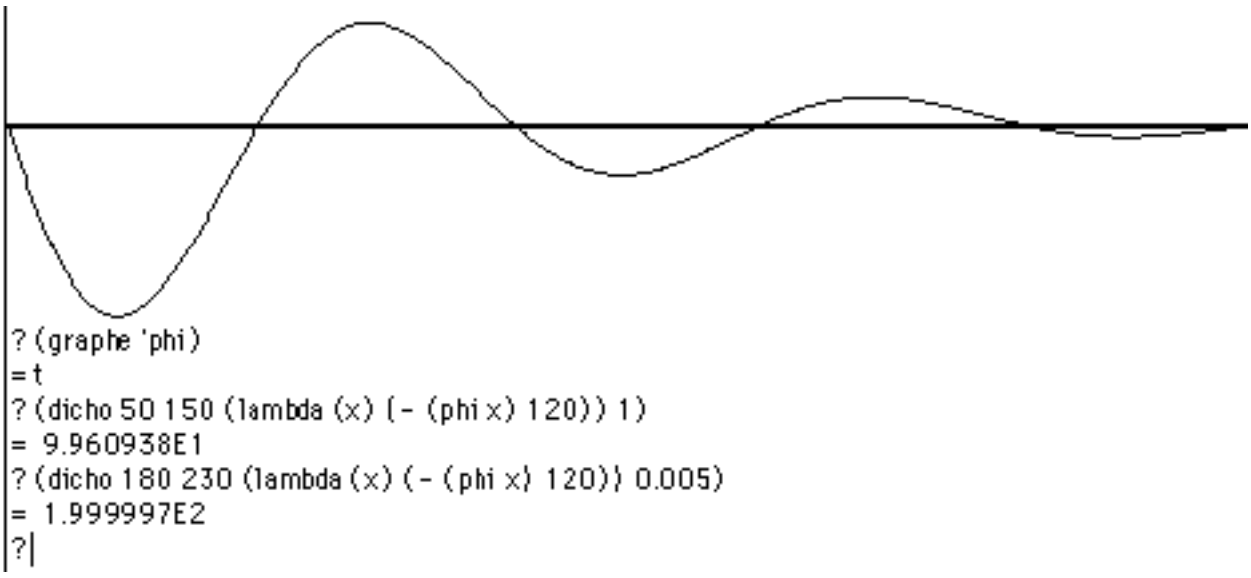
```
(de graphe (f)    (pen-size 2 2);    trace la fonction f pour c < y < d et a < x < b
                  (move-to 500 120) (line-to 0 120) (line-to 0 0) (line-to 0 250) (pen-size 1 1)
                  (let ((x 0)) (repeat 500 (line-to x (truncate (funcall f x))) (set 'x (1+ x))) ))
```

11-28° Construire la fonction calculant la racine d'une fonction entre a et b par dichotomie

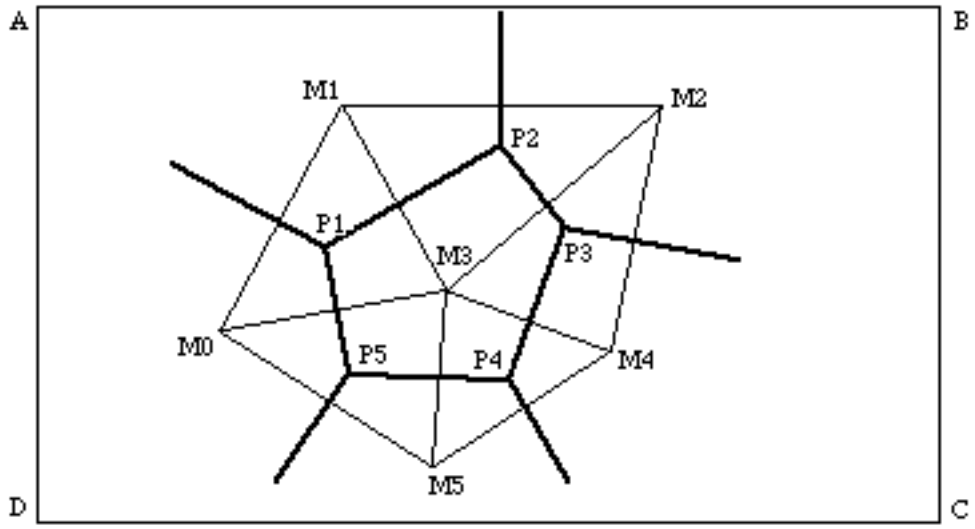
On peut, ce qui est impossible en Pascal, passer la fonction en argument, ce que l'on fera, soit avec son nom comme "sin par exemple ou bien comme sur le listing ci-dessous par une définition créée pour l'occasion.

```
(de dicho (a b phi eps) ; donne l'unique c entre a et b à eps près tel que phi soit nul
                  (dichobis a b (divide (+ a b) 2) phi eps))

(de dichobis (a b c phi eps) (cond
                  ((< (abs (- b a)) eps) c)
                  ((<= (* (funcall phi a) (funcall phi c)) 0) (dichobis a c (divide (+ a c) 2) phi eps))
                  (t (dichobis c b (divide (+ b c) 2) phi eps))))

(de phi (u) (truncate (+ 120 (* 100 (exp (divide u -150))    ; facteur pour la sinusoide amortie
                        (sin (* 0.0314159 u) )))))
```

Cette fonction $100e^{-u/150}$ sin (πu / 100) s'annule pour u multiple de 100, on retrouve quasiment 100 et 200 dans les deux intervalles ci-dessous.



11-29° Triangulation de Delaunay [Georges, Hermeline 89]. Etant donnés n points M_1, ... , M_n dans le plan (dans un espace de dimension d) grâce aux médiatrices (hyperplans médiateurs) on peut définir $V_i = \{M$ / pour tout j, $MM_i \leq MM_j\}$, chaque V_i est l'intersection de demi-plans fermés donc est un polyèdre convexe dont les arêtes (des portions de médiatrices) constituent le "maillage de Voronoï" (en gras).



Aux sommets P_1, ... , P_m de ce polyèdre, on associe l'enveloppe convexe des points Mi dont le polyèdre de Voronoï associé admet Pi comme sommet. c'est alors un polytope (polyèdre convexe borné) réduit la plupart du temps à un simplexe (enveloppe convexe de d+1 points affinement indépendants). En dimension 2, ce sera des triangles dans le cas où on a aucune relation de cocyclicité pour plus de 3 points. Cette triangulation est indépendante de l'ordre des points. L'algorithme à programmer est le suivant (on suppose qu'il n'y aura pas de points cocycliques autres que les triangles formés avec l'ensemble des points M_0, ... , M_n) :
a) Rajouter 4 points formant un rectangle contenant tous les points de la famille (on prendra pour A, B, C, D las points (0, 0) (0, 300) (500, 0) et (500, 300)) et soit D_0 l'ensemble de triangles formé initialement par (A, B, C) et (A, C, D).
b) Si A, B, C, D, M_0, ..., M_i est déjà triangulé par un ensemble D_i de triangles (tous orientés positivement), soit LS l'ensemble de ces triangles dont le cercle circonscrit contient le nouveau point courant M. Alors si F_1, ... F_p sont les arêtes (faces) externes (c'est à dire non communes à 2 éléments de LS) des triangles (simplexes) de LS, on aura :

$D_{i+1} = (D_i - LS) \cup \{S_j / 1 \leq j \leq p\}$ où Sj est le triangle orienté positivement formé avec M_{i+1} et F_j.
c) On retire tous les triangles comprenant l'un des 4 points rajoutés artificiellement au départ.
d) En utilisant les procedures "move-to" et "line-to", visualiser des essais.

Solution :
Les produits par 1.0 ne sont présents que pour modifier le type des résultats en réels.

```
(de determ (a b c d e f g h i) ; calcule le déterminant 3*3
                (- (+ (* a e i 1.) (* b f g 1.) (* c d h 1.)) (* c e g 1.) (* b d i 1.) (* a f h 1.)))

(de circons (M A B C)
                (circonsbis (car M) (cadr M) (car A) (cadr A) (car B) (cadr B) (car C) (cadr C)))

(de circonsbis (x y xa ya xb yb xc yc)
    (> (determ  (- (+ (* xa xa 1.) (* ya ya 1.)) (* x x 1.) (* y y 1.)) (- xa x) (- ya y)
                (- (+ (* xb xb 1.) (* yb yb 1.)) (* xa xa 1.) (* ya ya 1.)) (- xb xa) (- yb ya)
                (- (+ (* xc xc 1.) (* yc yc 1.)) (* xa xa 1.) (* ya ya 1.)) (- xc xa) (- yc ya) )  0))

(de tricirc (M LT) (cond ; renvoie la liste des triangles de LT qui sont circonscrits au point M
                ((null LT) nil)
                ((circons M (caar LT) (cadar LT) (caddar LT)) (cons (car LT) (tricirc M (cdr LT))))
                (t (tricirc M (cdr LT)))))
```

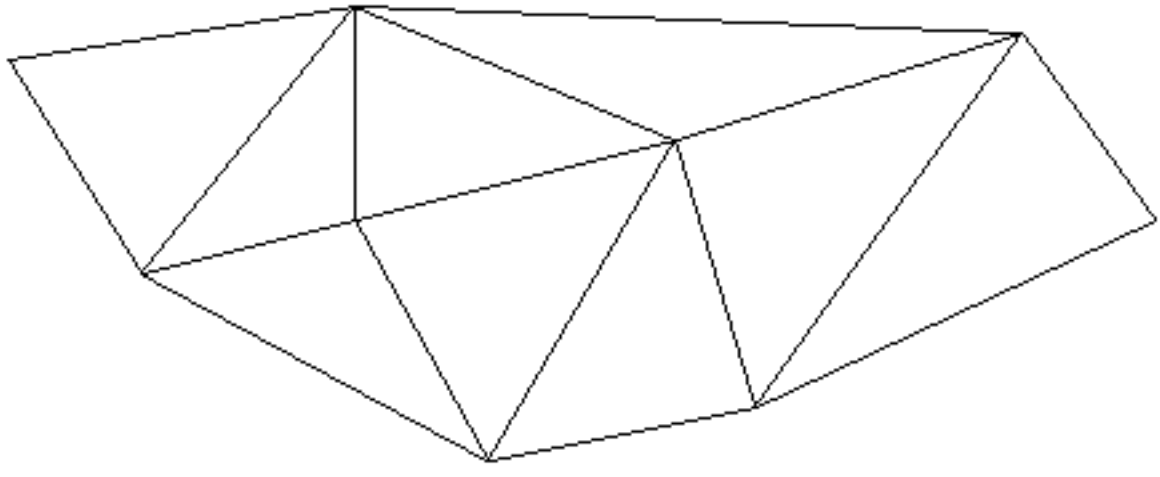
```
(de faces (LT) ; renvoie la liste de toutes les faces des triangles de LT
      (ifn   (null LT)     (mcons      (list (caar LT) (cadar LT))
                                       (list (cadar LT) (caddar LT))
                                       (list (caddar LT) (caar LT)) (faces (cdr LT)))))

(de opp (C) (list (cadr C) ( car C)))
(de reduire (LF) (cond ; renvoie la liste de faces LF où les faces opposées ont été retirées
                    ((null LF) nil)
                    ((member (opp (car LF)) (cdr LF)) (reduire (retirer (opp (car LF)) (cdr LF))))
                    (t (cons (car LF) (reduire (cdr LF))))))
(de retirer (X L) (cond
                    ((null L) nil)
                    ((equal X (car L)) (cdr L))
                    (t (cons (car L) (retirer X (cdr L))))))

(de positif (TR) ; réoriente le triangle TR s'il n'est pas positif
      (if           (< 0 (determ 1 0 0 0 (- (caadr TR) (caar TR)) (- (cadadr TR) (cadar TR))
                              0 (- (caaddr TR) (caar TR)) (- (cadr (caddr TR)) (cadar TR))) ) TR
                    (list (car TR) (caddr TR) (cadr TR))))

(de nouvtri (M LF) ; renvoie la liste des triangles formés avec le sommet M et les faces de LF
                    (ifn (null LF) (cons (positif (cons M (car LF))) (nouvtri M (cdr LF)))))
(de retrait (M LT) (cond ; liste de triangles LT dans où on a retiré ceux ayant le point M
                    ((null LT) nil)
                    ((member M (car LT)) (retrait M (cdr LT)))
                    (t (cons (car LT) (retrait M (cdr LT))))))
(de diff (L M) (cond ; différence ensembliste
                    ((null L) nil)
                    ((member (car L) M) (diff (cdr L) M))
                    (t (cons (car L) (diff (cdr L) M)))))
```

Conformément à l'esprit de récursivité mutuelle, on écrira :

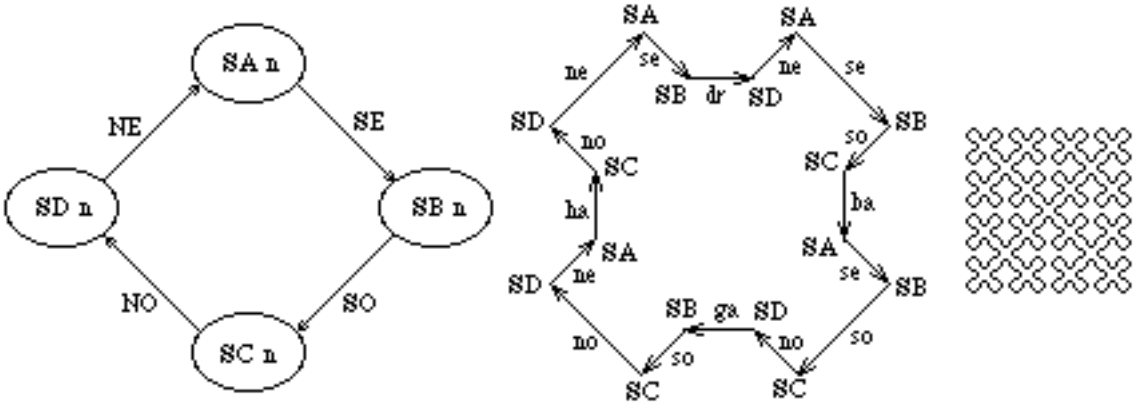
```
(de triang (LP) ; fournit la liste des triangles composée à partir de la liste de points LP
      (ifn (null LP) (triangbis '(((0 0) (500 0) (0 300)) ((0 300) (500 0) (500 300)))  LP)))
(de triangbis (D LP) ; D est la liste des triangles déjà formés, LP la liste des points restants
      (if           (null LP) (retrait '(0 0) (retrait '(500 0) (retrait '(0 300) (retrait '(500 300) D))))
                    (triangter D (car LP) (tricirc (car LP) D) (cdr LP))))
(de triangter (D M LS LP) ; LS liste des triangles de D circonscrit au point courant M
                    (triangbis (append (diff D LS) (nouvtri M (reduire (faces LS)))) LP))

(de vue (LT) (if (null LT) 'Voilà   ; visualise les triangles de la liste LT
                    (move-to (caaar LT) (cadaar LT)) (line-to (caadar LT) (cadr (cadar LT)))
                    (line-to (caadr (cdar LT)) (cadar (cddar LT))) (line-to (caaar LT) (cadaar LT)) (vue (cdr LT))))
```

Exemple :

```
(vue (triang '((20 70) (150 50) (150 130) (450 130) (400 60) (300 200) (70 150) (270 100) (200 220))))
```



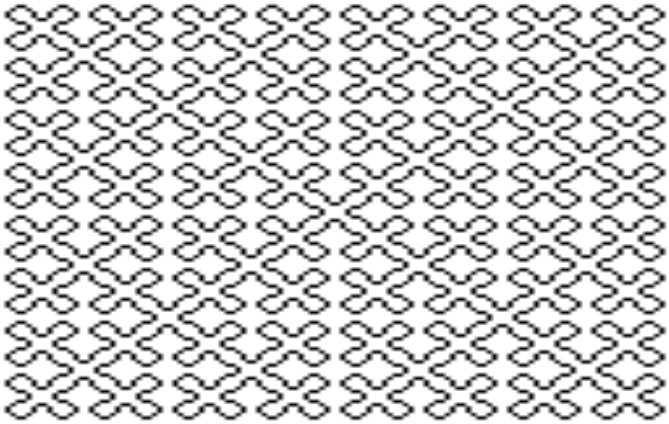
11-30° Courbe de hilbert (doit remplir un carré avec une courbe continue sans point double). Il s'agit, en partant d'un point (x, y), de suivre au niveau 1 le circuit de gauche où sa, sb, sc, sd sont des procédures de tracé décrites récursivement par la figure droite. [Breaud-Pouliquin 82] Les procédures modifiant (x, y) sont données par droite, gauche, haut, bas, nord-est, sud-est, sud-ouest, nord-ouest :




```
(de 0+ (x) x)
(de op (f g) (line-to x y) (set 'x (funcall f x)) (set 'y (funcall g y)))  ; est un appel de f et de g sur x, y (globales)
(de dr () (op '1+ '1+) (op '1+ '0+) (op '1+ '1-) (op '0+ '1-))
(de ga () (op '1- '1-) (op '1- '0+) (op '1- '1+) (op '0+ '1+))
(de ha () (op '1+ '1-) (op '0+ '1-) (op '1- '1-) (op '1- '0+))
(de ba () (op '1- '1+) (op '0+ '1+) (op '1+ '1+) (op '1+ '0+))
(de ne () (op '1+ '1-) (op '1+ '1-) (op '1+ '0+))
(de se () (op '1+ '1+) (op '1+ '1+) (op '0+ '1+))
(de so () (op '1- '1+) (op '1- '1+) (op '1- '0+))
(de no () (op '1- '1-) (op '1- '1-) (op '0+ '1-))
```

On peut utiliser un prédicat "zerop" testant si son argument est nul ou non.
La réponse est :

```
(de hilb (n) (set 'x 20) (set 'y 20) (move-to x y) (sa n) (se) (sb n) (so) (sc n) (no) (sd n) (ne))

(de sa (n) (if (zerop n) nil (sa (1- n)) (se) (sb (1- n)) (dr) (sd (1- n)) (ne) (sa (1- n))))
(de sb (n) (if (zerop n) nil (sb (1- n)) (so) (sc (1- n)) (ba) (sa (1- n)) (se) (sb (1- n))))
(de sc (n) (if (zerop n) nil (sc (1- n)) (no) (sd (1- n)) (ga) (sb (1- n)) (so) (sc (1- n))))
(de sd (n) (if (zerop n) nil (sd (1- n)) (ne) (sa (1- n)) (ha) (sc (1- n)) (no) (sd (1- n))))
```



11-31° Un système de maintien de la cohérence [Doyle 79, DeKleer 86]. En se plaçant dans la logique des propositions ordinaire avec (T=vrai, F=faux, I=incertain, TF=contradictoire), (il ne sera pas question de la négation c'est à dire qu'au lieu d'avoir systématiquement A et non (A) → TF, on aura seulement des règles particulières A et B → TF). Dans le fonctionnement du système, on met des axiomes (des faits ayant une justification vide), des hypothèses (des faits H ayant (H) comme justification), et une base de règles sous forme de clauses de Horn (si A1 et A2 et ... → A est une règle, alors cela signifie que (A1 A2 ...) sera une justification pour A.
Un environnement est une conjonction de propositions, il est dit consistant s'il n'entraîne pas de contradiction.
Une justification est une liste d'hypothèses.
Un label L(A) pour une proposition A est une disjonction de justifications (les différents "mondes" qui permettent de déduire A)
L(A) est minimal si aucune des justifications présentes n'est sur-ensemble d'une autre (ce sont les plus générales et il n'y a pas de redondance).
L(A) est consistant si chaque justification présente est consistante.
L(A) est complet si tout environnement consistant où A peut être déduit est un sur-ensemble d'une des justifications déjà présente. (aucune justification n'est oubliée)
L(A) est correct (ou sain ou bien construit) si chaque justification présente entraîne A.
Le rôle du système est de mettre à jour la liste de justification de chaque assertion, en en faisant un label sain, consistant minimal et complet, et de le recalculer chaque fois que l'on rajoute une justification c'est à dire une nouvelle règle (par exemple si on rajoute (A B) comme justification de TF, cela signifie que A et B ne peuvent plus coexister.
Construire le système à l'aide de p-listes et le faire vivre sur un exemple en rajoutant des justifications et des contradictions (qui ne sont que des justifications de TF).

Solution :

```
(de inclu (L M) (cond ; est la relation d'inclusion de la liste L dans la liste M
                      ((null L) t)
                      ((member (car L) M) (inclu (cdr L) M))
                      (t nil)))
(de ajout (nouv LL) (ajoutbis nouv LL nil)); ajoute conditionnellement "nouv" à LL déjà minimale

(de ajoutbis (ref  ll res) (cond ; on confronte la liste ref avec LL déjà minimisée, RES= résultat
                      ((null ll) (cons nouv res))
                      ((inclu ref (car ll)) (ajoutbis ref (cdr ll) res))
                      ((inclu (car ll) ref) (append res ll))
                      (t (ajoutbis ref (cdr ll) (cons (car ll) res)))))
(de minimise (LL) ; renvoie la liste de listes LL dans laquelle aucun élément n'est sur-ensemble d'une autre
                      (if (null ll) nil  (ajout (car LL) (minimise (cdr LL)))))
(de justif  (J F EX) ; permet de rajouter une justification J du fait F dans l'exemple EX
                      (putprop EX (ajout J (getprop EX F)) F) ; est une affectation
                      (vue (consist EX (plist EX)))) ; visualise aussitôt tous les labels

(de consistfait (EX F) ; modifie les justifications de F de telle sorte que toutes soient consistantes
                      (putprop EX (consistbis (getprop EX 'TF) nil (getprop EX  F)) F))
(de consistbis (LT LV LR) (cond ; LT est la liste des contradictions possibles
                      ((null LT) LR)
                      ((null LR) (consistbis (cdr LT) nil LV))
                      ((inclu (car LT) (car LR)) (consistbis LT LV (cdr LR)))
                      (t (consistbis LT (cons (car LR) LV) (cdr LR)))))

(de consist (EX PL) (cond ; rend consistant tous les labels, PL est la p-liste associée à l'objet EX
                      ((null PL) (plist EX))
                      ((eq (car PL) 'TF) (consist EX (cddr PL)))
                      (t (consistfait EX (car PL)) (consist EX (cddr PL))))) ; succession d'affectations

(de vue (L) (if (null L) 'voilà (vuebis (cadr L)) (print '--> (car L)) (vue (cddr L))))

(de vuebis (LL) (if (null (cdr LL)) (prin (car LL)) (prin (car LL) "' ou ") (vuebis (cdr LL))))
```

Exemple :

```
(plist 'ex1 nil)
= ()
(justif '(suede blond grand ybleu) 'nordique 'ex1)
(suede blond grand ybleu)-->nordique
= voilà
(justif '(norvege) 'nordique 'ex1)                    ; on a rajouté une autre justification
(norvege) ou (suede blond grand ybleu)-->nordique
= voilà
(justif '(suede grand blond) 'nordique 'ex1)
(suede grand blond) ou (norvege)-->nordique       ; la minimisation s'est effectuée
= voilà
(justif '(brun petit ynoir) 'medit 'ex1)
(brun petit ynoir)-->medit
(suede grand blond) ou (norvege)-->nordique
= voilà
(justif '(brun blond grand) 'nordique 'ex1)
(brun petit ynoir)-->medit
(brun blond grand) ou (norvege) ou (suede grand blond)-->nordique
= voilà
(justif '(petit grand) 'tf 'ex1)
(petit grand)-->tf
(brun petit ynoir)-->medit
(suede grand blond) ou (norvege) ou (brun blond grand)-->nordique
= voilà
(justif '(brun blond) 'tf 'ex1)                       ; brun et blond et grand devient impossible
(brun blond) ou (petit grand)-->tf
(brun petit ynoir)-->medit
(suede grand blond) ou (norvege)-->nordique
= voilà
(justif '(petit brun) 'medit 'ex1)
(brun blond) ou (petit grand)-->tf
(petit brun)-->medit ; minimisation
(suede grand blond) ou (norvege)-->nordique
= voilà
(justif '(grand blond) 'nordique 'ex1)
(brun blond) ou (petit grand)-->tf
(petit brun)-->medit
(grand blond) ou (norvege) -->nordique
= voilà
(justif '(suede) 'blond 'ex1)
(suede)-->blond
(brun blond) ou (petit grand)-->tf
(petit brun)-->medit
(grand blond) ou (norvege) -->nordique
= voilà
(justif '(brun petit) 'medit 'ex1)                    ; l'ordre ne compte pas dans une justification
(suede)-->blond
(brun blond) ou (petit grand)-->tf
(brun petit)-->medit
(grand blond) ou (norvege) -->nordique
= voilà
```

Comme on le voit une extension consisterait à retirer des justifications dont tous les faits sont justifiés par une autre justification, par exemple si (suede → blond) et (suede ou blond → nordique) la minimisation donnerait seulement (blond → nordique).

11-32° Problème du professeur Pantel : il s'agit d'énumérer toutes les équations $e_{ij,kl} + e_{kl,ij} = e_{ik,jl} + e_{jl,ik}$ dans lesquelles $e_{ij,kl} = \partial^2 f_{ij} / \partial x_k \partial x_l$, et sachant que $f_{ij} = f_{ji}$. A cause du théorème de Schwarz sur les dérivées partielles, on peut considérer les couples de paires avec répétitions, c'est à dire tous les quadruplets (i j k l) avec $i \leq j$ et $k \leq l$ et $i < k$ et $j \neq k$, en éliminant les images par la transformation (i j k l) tr → (i k j l). On se servira également de la transformation (i j k l) vs → (k l i j) qui donne la même équation.

Si n est le nombre d'indices :

```
(de tr (i j k l) (mcons i k (if (< l j) (list l j) (list j l))))
(de vs (i j k l) (list k l i j))
(de eqt (L) (list L '(apply 'vs L) (apply 'tr L) (apply 'vs (apply 'tr L))))
```

Cst0 vérifie que (i j k l) est un bon candidat en ce qui concerne les inégalités et qu'il ne provoque pas par "tr" un quadruplet déjà trouvé.

```
(de cst0 (i j k l n q) (cond
                ((< l k) (cst0 i j k (1+ l) n q))
                ((<= k i) (cst0 i j (1+ k) l n q))
                ((= j k) (cst1 i j (1+ k) l n q))
                ((member (tr i j k l) q) (cst1 i j k (1+ l) n q))
                (t (cst1 i j k (1+ l) n (cons (list i j k l) q))) ))

(de cst1 (i j k l n q) (cond            ; cst1 cherche le suivant de (i j k l) dans l'ordre alphabétique
                ((< n i) (mapcar 'print (mapcar 'eqt (reverse q))) nil)
                ((< n j) (cst1 (1+ i) (1+ i) (+ i  2) (+ i 2) n q))
                ((< n k) (cst1 i (1+ j) (1+ i) (1+ i) n q))
                ((< n l) (cst1 i j (1+ k) (1+ k) n q))
                (t (cst0 i j k l n q)) ))
(de cst (n) (cst0 1 1 2 2 n nil ))
```

Exemple :

```
(cst 2)
((1 1 2 2) (2 2 1 1) (1 2 1 2) (1 2 1 2)) ; une seule équation en fait
= ()

(cst 3)
((1 1 2 2) (2 2 1 1) (1 2 1 2) (1 2 1 2)) ; 6 équations
((1 1 2 3) (2 3 1 1) (1 2 1 3) ( 1 3 1 2))
((1 1 3 3) (3 3 1 1) (1 3 1 3) (1 3 1 3))
((1 2 3 3) (3 3 1 2) (1 3 2 3) (2 3 1 3))
((1 3 2 2) (2 2 1 3) (1 2 2 3) (2 3 1 2))
((2 2 3 3) (3 3 2 2) (2 3 2 3) (2 3 2 3))
= ()
```

11-33° Numération japonaise

On définit avec JAP, une liste de mots :
(set 'JAP '((nil nil giu hyaku sen man) itchi ni san shi go roku shitchi hatchi ku giu))
(On utilise aussi 4 = yo, 7 = nana, et 9 = kokono), puis une fonction "num" à un argument entier (qui se le repasse en liste):

```
(de num  (N) (cond
                ((atom N) (num (explodech N)))
                ((null (cdr N))     (ifn (eq (car N) 0) (list (nth (car N) JAP))))
                (t (specons (car N) (nth (length N) (car JAP)) (num (cdr N))))))

(de specons (X mot suite) (cond ; est un "cons" qui tient un compte spécial de 0 ou 1
                ((eq X 0) suite)
                ((eq X 1) (cons mot suite))
                (t (mcons (nth X JAP) mot suite))))
```

Exemples d'utilisations :

(num 289) ⇨ (ni hyaku hatchi giu ku)

(num 23456) ⇨ (ni man san sen shi hyaku go giu roku)

(num 10517) ⇨ (man go hyaku giu shitchi)

11-34° Reprendre la numération littérale en plusieurs langues, en s'efforçant de tout mettre dans une seule fonction.

Nous choisissons de représenter les données par une liste dont les éléments de rangs 1, 2, ... seront les noms pour 1, 2, 3, ...et l'élément de rang 0 étant la liste (100 1000 pour les numérations quinaires, suivi éventuellement de 20 pour les numérations vigésimales, et de 30, 40, 50, ... pour les autres. On a choisit, ce qui n'est pas le plus simple, de constituer un petit fichier séparé pour chaque langue, d'où l'utilité de la fonction "loa".
La solution proposée ici ne tient pas compte des pluriels et des tirets, elle met au contraire des tirets partout. En outre elle ne convient que jusqu'à 9999 sans le zéro.

Les petits fichiers :

```
(set 'KME '((roy pan) muoy pir bei buon pram () () () () dap)) ; cambodgien quinaire simplifié (sans exceptions)
(set 'JAP '((hyaku sen) itchi ni san shi go roku shitchi hatchi ku giu)) ; régulière
(set 'ESP '((cent mil) unu du tri kvar kvin ses sep ok nau dek)) ; l'espéranto aussi
(set 'QUE '((pachaj waranqa) juk iskay kimsa tawa pisqa soqta qanchis pusaq isqon chunka)) ; quéchua
(set 'BAS '((ehun mila hogoi) bat bi hiru lau bost sei zazpi zortzi bederatzi hamar hameka))
(set 'BRE '((kant mil ugent) unan daou tri pevar pemp chouech seizh eizh nao dek unnek daouzek trizek pevarzek pemzek chouezek seizek triouech naontek))
(set 'ALB '((qind mije) nji dy tri kater pese gjashte shtaa tete nande dhete njimbedhete dymbedhete trembedhete katermbedhete pesembedhete gjashtembedhete shtatembedhete tetembedhete nandembedhete)) ; albanais
(set 'ANG '((hundred thousand twenty thirty forty fifty sixty seventy eighty ninety) one two three four five six seven eight nine ten eleven twelve thirteen () fifteen () () eighteen))
(set 'ALL '((hundert tausend zwanzig dreissig vierzig fünfzig sechzig siebzig achtzig neunzig) ein zwei drei vier fünf sechs sieben acht neun zehn elf zwölf () () () sechszehn siebzehn))
(set 'SPA '((cento mil veinte treinte quaranta cinquanta sessenta settenta ochenta noventa) uno dos tres cuatro cinco seis siete ocho nueve diez once doce trece catorce quince))
(set 'SUI '((cent mille vingt trente quarante cinquante soixante septante octante nonante) un deux trois quatre cinq six sept huit neuf dix onze douze treize quatorze quinze seize))
(set 'BEL (subst 'quatre-vingt 'octante SUI))
(set 'FRA (subst () 'nonante BEL))
```

Cette solution de prendre une grosse fonction où interviennent des cas particuliers de langues précises n'est pas plus satisfaisante que celle de prendre une fonction par langue. L'idéal serait d'adjoindre à chacun des fichiers une ou plusieurs petites fonctions portant le même nom (par exemple "moinsde100 (D U)" donnant l'expression pour une dizaine et une unité) que la fonction principale prendrait en compte.

Petites fonctions :

```
(de mod (N P) (if (> P N) N (mod (- N P) P))) ; la fonction modulo dans tous les cas
(de loa (X) (cond ;réalise un chargement conditionnel du fichier
                  ((member X '(FRA BEL)) (loa 'SUI) X)
                  ((boundp X) X) ; "boundp" indique si son argument est un nom de variable réellement affecté
                  (t (implode (tet (explode (loadfile X)))))))
(de tet (L) ; donne les 3 premiers éléments d'une liste
                  (list (car L) (cadr L) (caddr L)))
(de cdl (L) (ifn (null (cdr L)) (cons (car L) (cdl (cdr L))))) ; tout sauf le dernier de L
(de ct L
; est une concaténation des éléments de L, en en sautant 3 après un nil et les 2 derniers si le dernier est nil
(cond             ((null (car (last L))) (apply 'ct (cdl (cdl L))))
                  ((null (car L)) (apply 'ct (nthcdr 4 L)))
                  ((member nil L) (concat (car L) (apply 'ct (cdr L))))
                  (t (apply 'concat L))))
```

Noter encore qu'en lisp la liste des arguments (ici L dans "ct") peut parfaitement être passée sous forme d'un paramètre donc de longueur tout à fait variable.

La fonction principale (ce n'est pas un bon exemple à cause de sa longueur démesurée, mais un exercice de style) :

```
(de num (N L) (cond
((null N) (prin '"Quel nombre ou liste de nombres sous forme (départ arrivée incrément) ? ") (num (read) L))
((null L) (prin '"Quelle langue ou liste de langues ? ") (num N (loa (read))))
((listp N) (let ((X (car N)) (I (ifn (null (cddr N))(caddr N) 1))) ; cas d'une série de nombres
                (until (> X (cadr N)) (print X '= (num X L)) (set 'X (+ X I))) 'voilà))
((listp L) ; cas de plusieurs langues
                (mapcar 'loa L) (print N) (mapcar (lambda (X) (print X '= (num N X))) L) 'voilà)
((eq N 0) nil) ; cas de N=0
((and (< N 20) (null (nth N (eval L)))) (cond ; N<20 et pas dans les données
                ((eq L 'ALL) (ct (nth (- N 10) ALL) 'zehn)) ; 4 cas pas très jolis
                ((eq L 'ANG) (ct (nth (- N 10) ANG) 'teen))
                ((eq L 'SPA) (ct 'diez-y- (nth (- N 10) SPA)))
                ((eq L 'FIN) (ct (nth (- N 10) FIN) 'toista))
                ((and (null (nth 6 (eval L))) (< N 10)) ; numération quinaire
                        (ct (nth 5 (eval L)) '- (nth (- N 5) (eval L))))
                (t (ct (nth 10 (eval L)) '- (num (- N 10) L))))) ; cas général régulier N<20
((< N 20) (nth N (eval L))) ; N<20 et dans le dico
((eq N 100) (caar (eval L))) ; N=100
((and (< N 1000) (null (cddar (eval L)))) ; numération régulières
                (ct (num (div N 100) L) '- (caar (eval L)) '- (num (div (mod N 100) 10) L) '- (nth 10 (eval L)) '-
                        (num (mod N 10) L)))
((eq N 20) (nth 2 (car (eval L)))) ; N=20 non régulier
((and (< N 200) (null (cdddar(eval L)))) (cond ; numérations vigésimales
                ((eq L 'BRE) (cond
                        ((eq N 30) 'tregont)
                        ((eq N 50) 'hanter-kant)
                        ((member (div N 10) '(2 3 5 10))
                                (ct (num (mod N 10) L) '- 'ha '- (num (* (div N 10) 10) L)))
                        (t (ct (num (mod N 20) L) '- 'ha '- (num (div N 20) L) '-ugent))))
                (t (ct (nth (div N 20) (eval L)) '- (caddr (car (eval L))) '- (num (mod N 20) L)))))
((< N 100) (cond ;les survivances vigésimales en français
                ((and (eq N 81) (member L '(FRA BEL))) 'quatre-vingt-un)
                ((and (eq L 'FRA) (eq N 91)) 'quatre-vingt-onze)
                ((and (eq L 'FRA) (eq N 71)) 'soixante-et-onze)
                ((and (eq L 'FRA) (or (eq (div N 10) 7) (eq (div N 10) 9)))
                        (ct (nth (1- (div N 10)) (car FRA)) '- (num (mod N 20) L)))
                ((eq L 'ALL) (ct (num (mod N 10) L)'- 'und '- (nth (div N 10) (car ALL))))
                        ; l'inversion allemande
                ((and (eq (mod N 10) 1) (member L '(FRA BEL SUI))) ; le cas du "et" un
                        (ct (nth (div N 10) (car SUI)) '-et-un))
                (t (ct (nth (div N 10) (car (eval L))) '- (num (mod N 10) L))))); cas général N<100
((and (eq N 500) (eq L 'SPA)) 'quinientos)
((eq N 1000) (cadar (eval L)))
((and (< N 200) (member L '(SUI FRA BEL SPA ITA POR ROU))); cent au lieu de "un cent"
                (ct (caar (eval L)) '- (num (- N 100) L)))
((< N 1000) (ct (num (div N 100) L) '- (caar (eval L)) '- (num (mod N 100) L))); cas général N<1000
((and (< N 2000) (member L '(SUI FRA BEL SPA ITA POR ROU))); mille au lieu de "un mille"
                (ct (cadar (eval L)) '- (num (- N 1000) L)))
(t (ct (num (div N 1000) L) '- (cadar (eval L)) '- (num (mod N 1000) L)))))
```

Expérimentation, de 67 à 97 de 6 en 6 en plusieurs langues :

```
(num '(67 105 6) '(fra bel sui spa ang all))
```

67	fra=soixante-sept	bel=soixante-sept	sui=soixante-sept
	spa=sessenta-siete	ang=sixty-seven	all=sieben-und-sechzig
73	fra=soixante-treize	bel=septante-trois	sui=septante-trois
	spa=settenta-tres	ang=seventy-three	all=drei-und-siebzig
79	fra=soixante-dix-neuf	bel=septante-neuf	sui=septante-neuf
	spa=settenta-nueve	ang=seventy-nine	all=neun-und-siebzig
85	fra=quatre-vingt-cinq	bel=quatre-vingt-cinq	sui=octante-cinq
	spa=ochenta-cinco	ang=eighty-five	all=funf-und-achtzig
91	fra=quatre-vingt-onze	bel=nonante-et-un	sui=nonante-et-un
	spa=noventa-uno	ang=ninety-one	all=ein-und-neunzig
97	fra=quatre-vingt-dix-sept	bel=nonante-sept	sui=nonante-sept
	spa=noventa-siete	ang=ninety-seven	all=sieben-und-neunzig

```
(num '(1 19) 'kme)
1=muoy              2=pir              3=bei              4=buon          5=pram
6=pram-muoy         7=pram-pir         8=pram-bei         9=pram-buon     10=dap
11=dap-muoy         12=dap-pir         13=dap-bei         14=dap-buon     15=dap-pram
16=dap-pram-muoy    17=dap-pram-pir    18=dap-pram-bei    19=dap-pram-buon
= voilà

(num '(19 90 7) '(bre bas))
19    bre=naontek              bas=hamar-bederatzi
26    bre=chouech-ha-ugent     bas=bat-hogoi-sei
33    bre=tri-ha-tregont       bas=bat-hogoi-hamar-hiru
40    bre=daou-ugent           bas=bi-hogoi
47    bre=seiz-ha-daou-ugent   bas=bi-hogoi-zazpi
54    bre=pevar-ha-hanter-kant bas=bi-hogoi-hamar-lau
61    bre=unan-ha-tri-ugent    bas=hiru-hogoi-bat
68    bre=eiz-ha-tri-ugent     bas=hiru-hogoi-zortzi
75    bre=pemzek-ha-tri-ugent  bas=hiru-hogoi-hamar-bost
82    bre=daou-ha-pevar-ugent  bas=lau-hogoi-bi
89    bre=nao-ha-pevar-ugent   bas=lau-hogoi-bederatzi
= voilà

(num 1532 'que) = juk-waranqa-pisqa-pachaj-kimsa-chunka-iskay (une sinistre date pour les Quichuas)

(num '(1 120 11) 'jap)
1=itchi                 12=giu-ni                23=ni-giu-san
34=san-giu-shi          45=shi-giu-go            56=go-giu-roku
67=roku-giu-shitchi     78=shitchi-giu-hatchi    89=hatchi-giu-ku
100=hyaku               111=itchi-hyaku-itchi-giu-itchi
= voilà

(num '(1 85 9) 'esp)    ; espéranto
1=unu           10=dek          19=dek-nau      28=du-dek-ok    37=tri-dek-sep
46=kvar-dek-ses 55=kvin-dek-kvin 64=ses-dek-kvar 73=sep-dek-tri 82=ok-dek-du
```

11-35° Retour au problème du plus court chemin dans le métro. On se contentera d'une représentation de chaque ligne comme la liste de ses stations en négligeant les cas particuliers (boucles et fourches). On ne cherchera pas à optimiser, c'est à dire que les correspondances seront recherchées à chaque fois et leurs rangs dans les différentes lignes recalculées.

```
(set 'RP '(L1 L2 L4 L6 L8 L10 RER-A ))
(set 'L1 '(La-Défense Pont-de-Neuilly Les-Sablons Porte-Maillot Argentine  Etoile Georges-V Franklin-Roosevelt Champs-Elysées Concorde Tuileries Palais-Royal Louvre Châtelet Hôtel-de-ville St-Paul Bastille Gare-de-Lyon Reuilly-Diderot Nation Porte-de-Vincennes St-Mandé Bérault Château-de-Vincennes))
(set 'L2 '(Porte-Dauphine Victor-Hugo Etoile Ternes Courcelles Monceau Villiers Rome Place-de-Clichy Blanche Pigalle Anvers Barbès-Rochechouart La-chapelle Stalingrad Jaurès Colonel-Fabien Belleville Couronnes Ménilmontant Père-Lachaise Philippe-Auguste Alexandre-Dumas Avron Nation))
(set 'L4  '(Porte-d-Orléans Alésia Mouton-Duvernet Denfert-Rochereau Raspail Vavin Montparnasse St-Placide St-Sulpice St-Germain-de-prés Odéon St-Michel Cité Châtelet Les-Halles Etienne-Marcel Réaumur-Sébastopol Strasbourg-St-Denis Chateau-d-eau Gare-de-l-est Gare-du-nord Barbès-Rochechouard Chateau-rouge Marcadet-Poissonniers Simplon Porte-de-Clignancourt))
(set 'L6 '(Nation Picpus Bel-air Daumesnil Dugommier Bercy Quai-de-la-gare Chevaleret Nationale Place-d-Italie Corvisart Glacière St-Jacques Denfert-Rochereau Raspail Edgar-Quinet Montparnasse Pasteur Sevres-Lecourbe Cambronne La-Motte-Picquet Dupleix Bir-Hakeim-Grenelle Passy Trocadéro Boissière Kléber Etoile))
(set 'L8 '(Balard Lourmel Boucicault Félix-Faure Commerce La-Motte-Picquet-grenelle Ecole-Militaire Latour-Maubourg Invalides Concorde Madeleine Opéra Richeulieu-Drouot  Rue-Montmartre Bonne-Nouvelle Strasbourg-St-Denis République Filles-du-Calvaire St-Sébastien-Froissart Chemin-Vert Bastille Ledru-Rolin Faidherbe-Chaligny Reuilly-Diderot Montgallet Daumesnil Michel-Bizot Porte-dorée Porte-de-Charenton))
(set 'L10 '(Michel-Ange Chardon-Lagache Mirabeau javel Charles Michel Emile-Zola  La-Motte-Picquet-Grenelle Ségur Duroc Vanneau Sèvres-Babylone Mabillon Odéon Cluny Maubert Cardinal-Lemoine Jussieu Gare-d-Austerlitz))
(set 'RER-A '(La-Défense Etoile Auber Châtelet Gare-de-Lyon Nation Vincennes)) ; etc... etc ...
```

```
(de inter (L M) (cond      ; intersection de deux listes
                ((null L) nil)
                ((member (car L) M) (cons (car L) (inter (cdr L) M)))
                (t (inter (cdr L) M))))
```

On s'oriente sur une programmation détaillée avec 0, 1 ou 2 changements car on considère que 2 lignes ne se coupent qu'au maximum en 2 points et que l'on peut aller d'un point à un autre avec un maximum de 2 changements.

```
(de C1 (L M) (cond        ; renvoie la première correspondance entre les lignes L et M
                ((and (atom L) (atom M)) (C1 (eval L) (eval M)))
                ((null L) nil)
                ((null (member (car L) M)) (C1 (cdr L) M))
                (t (car L)) ))

(de C2 (L M) (ifn         ; renvoie la seconde correspondance s'il y en a une
                (C1 L M) (C1 (cdr (member (C1 L M) L)) M) ))

(de lgn (A R) (cond       ; donne la liste des lignes du réseau R passant en A
                ((null R) nil)
                ((member A (eval (car R))) (cons (car R) (lgn A (cdr R))))
                (t (lgn A (cdr R))) ))
```

Un chemin est codé par une liste de stations allant de la station de départ à celle d'arrivée, entre lesquelles sont intercalées les noms de lignes à prendre.
Une fonction "vue" donnera en clair l'explication de l'itinéraire avec le temps approximatif du parcours.

```
(de ch1 (A B R) (ifn              ; donne la liste des chemins directs de A à B s'il y en a
                (null R)          ; R sera toujours la liste des lignes communes de A et de B
                (cons (list A (car R) B) (ch1 A B (cdr R)))))

(de ch2 (A B R1 R2) (cond         ; donne la liste des chemins avec un changement
                ((null R1) nil)
                ((atom R1) (cond                ; tous les cas où R1 est une ligne
                        ((null R2) nil)
                        ((atom R2) (cond
                        ((eq R1 R2) nil)        ; après R1 et R2 sont deux lignes distinctes
                        ((null (C1 R1 R2)) nil)
                        ((null (C2 R1 R2)) (list (list A R1 (C1 R1 R2) R2 B)))
                        (t  (list (list A R1 (C1 R1 R2) R2 B) (list A R1 (C2 R1 R2) R2  B)))))
                        (t (append (ch2 A B R1 (car R2)) (ch2 A B R1 (cdr R2)))) ))
                (t (append (ch2  A B (car R1) R2) (ch2 A B (cdr R1) R2))) ))

(de ch3 (A B R1 R2 R) (cond       ; donne la liste des chemins avec 2 changements
((null R1) nil)                   ; des lignes de R1 vers les lignes de R2 en passant par les lignes de R
((atom R1) (cond
                ((null R2) nil)
                ((atom R2) (cond
                        ((null R) nil)
                        ((atom R) (cond ; ici on examine la ligne R1 coupant la ligne R, qui coupe la ligne R2
                                ((or (eq R1 R) (eq R2 R) (eq R1 R2) (null (C1 R1 R)) (null (C1 R2 R))) nil)
                                        ((and (null (C2 R1 R)) (null (C2 R2 R)))
                                        (list (list A R1  (C1 R1 R) R (C1 R2 R) R2 B)))
                                ((null (C2 R1 R)) (list (list A R1 (C1 R1 R) R (C1 R2 R) R2 B)
                                        (list A R1 (C1 R1 R) R (C2 R2 R) R2 B)))
                                ((null (C2 R2 R)) (list (list A R1 (C1 R1 R) R (C1 R2 R) R2 B)
                                        (list A R1 (C2 R1 R) R (C1 R2 R) R2 B)))
                                (t (list    (list A R1 (C1  R1 R) R (C1 R2 R) R2 B)
                                        (list A R1 (C1 R1 R) R (C2 R2 R) R2 B)
                                        (list A R1 (C2 R1 R) R (C1 R2 R) R2 B)
                                        (list A R1 (C2 R1 R) R (C2 R2 R) R2 B))) ))
                        (t (append (ch3 A B R1 R2 (car R)) (ch3 A B R1 R2 (cdr R)))) ))
                (t (append (ch3 A B R1 (car R2) R) (ch3 A B R1 (cdr R2) R)))))
(t (append (ch3 A B (car R1) R2 R) (ch3 A B (cdr R1) R2 R)))))
```

```
(de vue (ch) (cond
                ((numberp (car ch))
                (print '"Solution au départ de : " (cadr ch) '" Temps en minutes : " (car ch)) (vue (cdr ch)))
                ((null (cdddr ch)) ; explications pour le premier morceau
                        (print '" Prendre la ligne " (cadr ch) '" en direction de : "
                                (if (> (rang (caddr ch) (eval (cadr ch))) (rang (car ch) (eval (cadr ch))))
                        (car (last (eval (cadr ch)))) (car (eval (cadr ch)))) '" descendre à : " (caddr ch)))
                (t (vue (firstn 3 ch)) (vue (cddr ch))) ))

(de trajets (A B R); fonction principale renvoyant la liste triée des trajets possibles de A à B
                (trajetbis A B (lgn A R) (lgn B R) R))

(de trajetbis (A B LA LB R)
(mapcar 'vue  (tri (mapcar (lambda (c) (cons (quo (temps c) 60) c))
                        (append (ch1 A B (inter LA LB)) (ch2 A B LA LB) (ch3 A B LA LB R))))) nil )

(de tm (L) (cond ; temps moyen d'une station sur une ligne
                ((eq L 'corresp) 300)
                ((member L '(RER-A RER-B RER-C)) 240)      ; une station en RER
                ((member L '(L1 L4 L6 L11)) 70)            ; lignes sur pneus
                (t 80)))                                   ; autres lignes

(de rang (X L) ; rang de X dans L s'il est présent, nil sinon
                (rangbis (member X L) (length L)))
(de rangbis (M N) (if M (- N (length M))))

(de temps (ch) ; donne le temps de ce chemin
                (if     (null (cdddr ch))
                (* (tm (cadr ch)) (abs (- (rang (caddr ch) (eval (cadr ch)))  (rang (car ch) (eval (cadr ch))))))
                        (+ (tm 'corresp) (temps (firstn 3 ch)) (temps (cddr ch)) ) ))

(de fusion (L1R L2R LF) (cond           ; où chaque élément de LR est de la forme (clé ... ...)
                ((null L1R) (if (null L2R) (reverse LF) (fusion nil (cdr L2R) (cons (car L2R) LF))))
                ((null L2R) (fusion nil (cdr L1R) (cons (car L1R) LF)))
                ((< (caar L1R) (caar L2R)) (fusion (cdr L1R) L2R (cons (car L1R) LF)))
                (t (fusion L1R (cdr L2R) (cons (car L2R) LF))) ))

(de sep (pair LP LI LR)                 ; liste des éléments de rangs pairs, impairs et restant à séparer
                (if LR (if pair (sep nil (cons (car LR) LP) LI (cdr LR)) (sep t LP (cons (car LR) LI) (cdr LR)))
                        (cons LP LI)) )  ; renvoie ((a0 a2 a4 ...) a1 a3 a5 ....)

(de tri (L) (if (null (cdr L)) L (tribis (sep t nil nil L)))) ; mon tri par fusion préféré

(de tribis (LC) (fusion (tri (car LC)) (tri (cdr LC)) nil))

Essais :
(trajets 'Alésia 'Lourmel RP)
Solution au départ de : alésia  Temps en minutes : 33
                Prendre la ligne L4 en direction de : porte-de-clignancourt descendre à : odéon
                Prendre la ligne L10 en direction de : michel-ange descendre à : la-motte-picquet-grenelle
                Prendre la ligne L8 en direction de : balard descendre à : lourmel
Solution au départ de : alésia  Temps en minutes : 39
                Prendre la ligne L4 en direction de : porte-de-clignancourt descendre à : châtelet
                Prendre la ligne L1 en direction de : la-défense descendre à : concorde
                Prendre la ligne L8 en direction de : balard descendre à : lourmel
Solution au départ de : alésia  Temps en minutes : 42
                Prendre la ligne L4 en direction de : porte-de-clignancourt descendre à : strasbourg-st-denis
                Prendre la ligne L8 en direction de : balard descendre à : lourmel
Solution au départ de : alésia  Temps en minutes : 56
                Prendre la ligne L4 en direction de : porte-de-clignancourt descendre à : denfert-rochereau
                Prendre la ligne L6 en direction de : nation descendre à : daumesnil
                Prendre la ligne L8 en direction de : balard descendre à : lourmel
= ()
```