Chapitre 6

Applications graphiques

Ce chapitre présente quelques applications graphiques, les plus intéressantes provenant des programmes récursifs, notamment les courbes fractales.

Graphique

Les prédicats étant différents d'un Prolog à l'autre, nous donnons ceux de l'Open-Prolog, pour lequel une seule procédure *draw* permet plusieurs opérations, ainsi 1 pour l'ouverture d'une fenêtre graphique, 2 pour la fermeture, 4 pour dessiner un rectangle, etc., le troisième argument est le résultat, il ne sert pas ici.
Par exemple, la séquence suivante provoque les effets de bords du dessin.

draw(1, 'dessin'(50, 10, 400, 500), _).
draw(6, penSize(5,1),_).
draw(4, rect(170, 120, 230, 130),_).
draw(6, penSize(1,5),_).
draw(12, oval(100, 20, 300, 150),_).
draw(12, oval(150, 80, 180, 100),_).
draw(12, oval(220, 80, 250, 100),_).

Signalons *draw(9, color(0),_).* pour la couleur noire, 1 = jaune, 2 = magenta, 3 = rouge, 4 = cyan, 5 = vert, 6 = bleu, 7 = blanc.

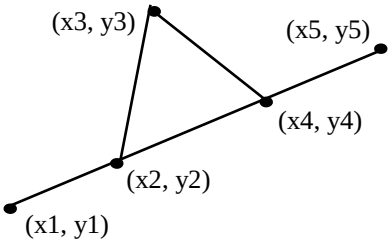
Pour plus de commodité, et afin de transposer à des Prolog n'ayant pas les mêmes mots réservés pour ces prédicats, nous réécrivons les quatre suivants dont le dernier suffit aux exercices qui suivent. Comme il n'est possible d'ouvrir

qu'une seule fenêtre graphique, autant appeler *ouvrir*, *effacer*, *fermer* les opérations de base qu'il ne sera pas difficile de retrouver dans un autre Prolog.

```
ouvrir :- draw(1, 'dessin'(50, 10, 400, 500), _).
effacer :- draw(11,_,_)
        % effacement de tous les tracés dans la fenêtre graphique
fermer :- draw(2,_,_). % fermeture de la fenêtre graphique
line(X1, Y1, X2, Y2) :- draw(3, line(X1, Y1, X2, Y2),_).
```

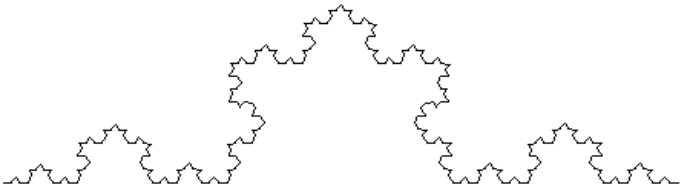
Le tracé d'un trait entre deux points est tout ce qu'il faut pour ce qui suit.

1 Fractales avec Von Koch (1904)
La courbe de Von Koch pour $n = 0$ est un segment orienté du point (x_1, y_1) vers (x_5, y_5). Sinon, à chaque étape, le segment est divisé en trois, (x_2, y_2), (x_4, y_4), le second étant remplacé à sa gauche par deux segments mis bout à bout, de sommet (x_3, y_3), formant une ligne brisée de 4 segments de même longueur.
A l'ordre n + 1, cette opération est répétée sur chacun des segments correspondant à l'ordre *n*.



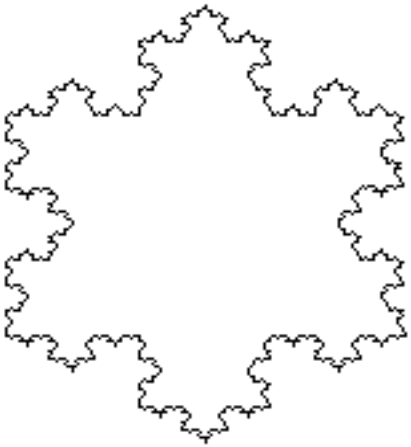

```
trig(0, X1, Y1, X2, Y2) :- line(X1, Y1, X2, Y2), !.
trig(N, X1, Y1, X5, Y5) :- M is N − 1, X2 is (2*X1 + X5)/3,
                           Y2 is (2*Y1 + Y5)/3, trig(M, X1, Y1, X2, Y2),
                           X3 is (X1 + X5)/2 + 2*(Y5 − Y1)/7,
                           Y3 is (Y1 + Y5)/2 −  2*(X5 − X1)/7,
                           trig(M, X2, Y2, X3, Y3), X4 is (X1 + 2*X5)/3,
                           Y4 is (Y1 + 2*Y5)/3, trig(M, X3, Y3, X4, Y4),
                           trig(M, X4, Y4, X5, Y5).
```

Exécution
trig(5, 10, 150, 450, 150).



En en enchaînant trois, l'exécution se demande par *etoile*(4).

```
etoile(N) :- trig(N, 10, 180, 85, 50),
        trig(N, 85, 50, 160, 180), trig(N, 160, 180, 10, 180).
```

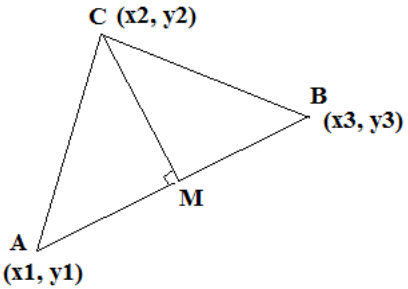


2 Dragon de Hilbert

Partant d'un vecteur joignant les points de coordonnées (x_1, y_1) et (x_3, y_3), on trace le triangle rectangle isocèle à gauche, son sommet est facile à trouver, ce sont les coordonnées (x_2, y_2) figurant dans le prédicat *hilbert*.
Hilbert, pour la valeur *n* sur ce segment consiste à tracer *hilbert* pour la valeur $n - 1$ sur les deux côtés de ce triangle. La récursivité s'arrête pour $n = 0$ où seul le segment considéré doit être tracé.

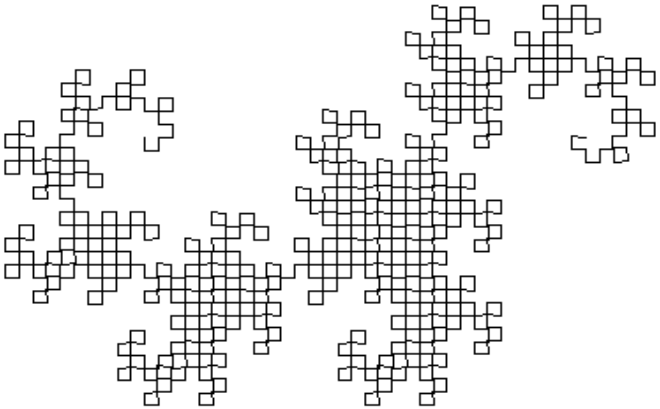
La question est de trouver les coordonnées (x_2, y_2) de C pour chaque segment AB connaissant les coordonnées (x_1, y_1) et (x_3, y_3) de A et de B.

Soit M le milieu du segment AB, la droite (MC) doit être orthogonale à (AB) et MC doit avoir la même distance que AM, de plus C doit être sur le côté gauche de l'axe AB.
On vérifie facilement que le vecteur de composantes ½($y_1 - y_3$) et ½($x_3 - x_1$) a un produit scalaire nul avec le vecteur AB et que sa longueur est ½AB, d'où les formules ci-dessous.




```
hilbert(0, X1, Y1, X2, Y2) :- line(X1, Y1, X2, Y2), !.
hilbert(N, X1, Y1, X3, Y3) :- M is N − 1,
                              X2 is (X1 + X3 + Y1 - Y3)/2,
                              Y2 is (X3 − X1 + Y1 + Y3)/2,
                              hilbert(M, X1, Y1, X2, Y2),
                              hilbert(M, X3, Y3, X2, Y2).
```

Exécution
hilbert(11, 100, 150, 350, 150).

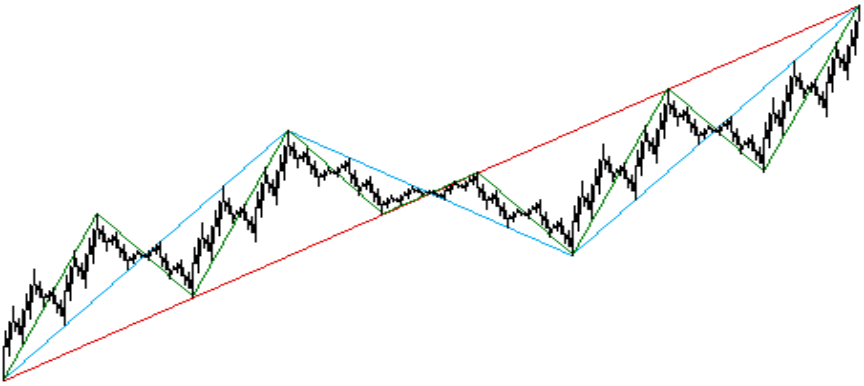


3 Escalier de Cantor
Il s'agit d'un contre-exemple célèbre de fonction de [0, 1] dans [0, 1] qui est continue en étant dérivable en aucun point.
Pour $N = 0$ c'est la diagonale, puis, pour $N = 1$, il faut découper celle-ci en trois en joignant les points d'abscisses et ordonnées respectives 1/3, 2/3 et 2/3, 1/3.

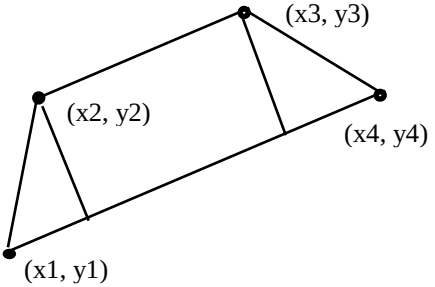
La diagonale et les trois courbes suivantes de niveau 1, 2 et 6 pour la plus foncée se voient sur le graphique.
Il est difficile d'apprécier à l'œil les escaliers suivants.

```
esc(0, X1, Y1, X2, Y2) :- !, line(X1, Y1, X2, Y2).
esc(N, X1, Y1, X4, Y4) :- M is N − 1,
                          X2 is (2*X1 + X4)/3, Y2 is (Y1 + 2*Y4)/3,
                          esc(M, X1, Y1, X2, Y2),
                          X3 is (X1 + 2*X4)/3, Y3 is (2*Y1 + Y4)/3,
                          esc(M, X2, Y2, X3, Y3),
                          esc(M, X3, Y3, X4, Y4).
```

```
ouvrir.
draw(9, color(3),_), esc(0, 10, 253, 458, 10). % rouge
draw(9, color(4),_), esc(1, 10, 253, 458, 10). % cyan
draw(9, color(5),_), esc(2, 10, 253, 458, 10). % vert
draw(9, color(0),_), esc(6, 10, 253, 458, 10). % noir
```

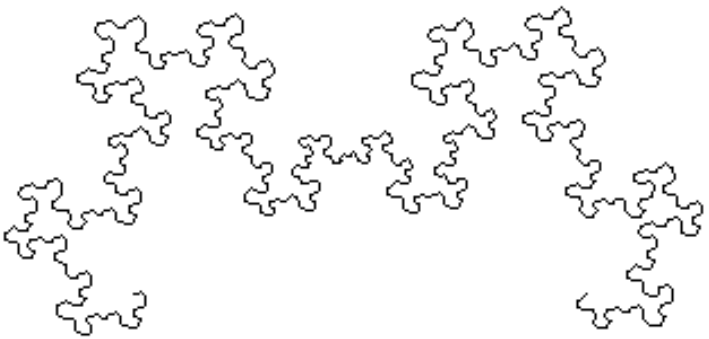


4 Courbe de Sierpinski (1915)
Cette fois chaque segment est remplacé par un trapèze : on part de 60° vers la gauche du point (x_1, y_1) pour arriver à (x_2, y_2) dont l'aplomb sur le segment $(x_1, y_1) - (x_4, y_4)$ tombe au quart, puis on trace un segment parallèle et moitié du segment initial pour arriver au point (x_3, y_3).
Une approximation de *sqrt*(3)/4 est faite par 443/1000.




```
sierp(0, X1, Y1, X2, Y2) :- line(X1, Y1, X2, Y2), !.
sierp(N, X1, Y1, X4, Y4) :- M is N − 1,
                            X2 is (X1 + (X4 - X1)/4 + (Y4 - Y1)/2),
                            Y2 is (Y1 + (Y4 − Y1)/4 − (X4 - X1)/2),
                            sierp(M, X1, Y1, X2, Y2)
                            X3 is (X2 + (X4 − X1)/2), Y3 is (Y2 + (Y4 − Y1)/2),
                            sierp(M, X3, Y3, X2, Y2), sierp(M, X3, Y3, X4, Y4).
```

```
ouvrir.
effacer.
sierp(6, 100, 250, 300, 250).
```

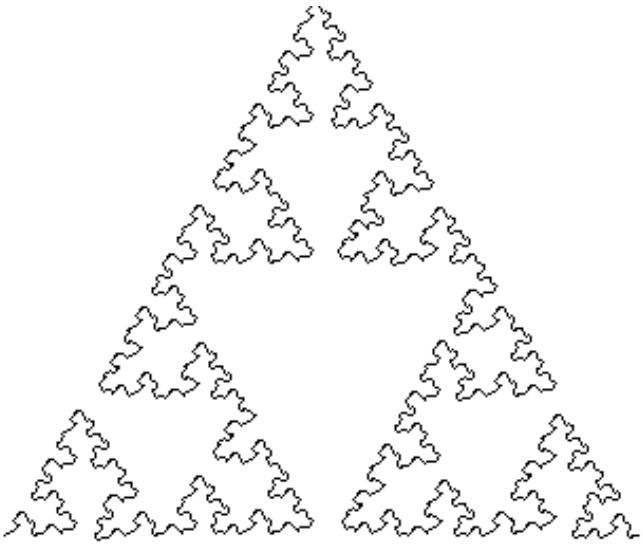


La véritable courbe consiste à partir vers la droite aux premier et troisième segments, il suffit d'inverser certains sens.

```
sierpbis(0, X1, Y1, X2, Y2) :- line(X1, Y1, X2, Y2), !. % cas N = 0

sierpbis(N, X1, Y1, X4, Y4) :- M is N − 1,
        X2 is (X1 + (X4 − X1)/4 + 433*(Y4 − Y1)/1000),
        Y2 is (Y1 + (Y4 − Y1)/4 − 433*(X4 − X1)/1000),
        sierpbis(M, X2, Y2, X1, Y1),
        X3 is (X2 + (X4 − X1)/2),
        Y3 is (Y2 + (Y4 − Y1)/2),
        sierpbis(M, X2, Y2, X3, Y3),
        sierpbis(M, X4, Y4, X3, Y3).
```

```
ouvrir.
effacer.
sierpbis(5, 100, 250, 400, 250).
```



Il est très facile de modifier les motifs, ainsi pour un simple cristal, on tracera au milieu de chaque segment un segment perpendiculaire de longueur égale au tiers du segment courant pour obtenir ceci :

```
cristal(0, X1, Y1, X2, Y2) :- line(X1, Y1, X2, Y2), !.

cristal(N, X1, Y1, X4, Y4) :- M is N − 1,
                              X2 is X1 + (X4 − X1)/2, Y2 is Y1 + (Y4 − Y1)/2,
                              X3 is X2 + (Y4 − Y1)/3, Y3 is Y2 − (X4 − X1)/3,
                              cristal(M, X1, Y1, X2, Y2), cristal(M, X2, Y2, X3, Y3),
                              cristal(M, X3, Y3, X2, Y2), cristal(M, X2, Y2, X4, Y4).
```

```
ouvrir.
effacer.
cristal(6, 50, 200, 400, 200).
```

Chapitre 7

Un Prolog écrit en Lisp

On présente dans ce chapitre quelques rudiments du langage Lisp qui fut longtemps le plus utilisé et reste le plus connu des langages fonctionnels. Sa très grande cohérence permet d'une part de le présenter brièvement et d'autre part de développer rapidement des applications qui, dans d'autres langages, seraient passablement ardues. Ainsi, pour écrire un programme réalisant un Prolog très simple, quelques pages suffiront.
Bien qu'ancien, car créé en 1960 par Mac Carthy, et fondé sur le λ-calcul de Church, le Lisp reste le plus simple des langages fonctionnels.
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.
Comme Prolog, le Lisp est particulièrement bien adapté à la récursivité, il faut, pour bien programmer en Lisp, abandonner les habitudes algorithmiques besogneuses : éliminer les affectations pour penser en termes 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.

Les objets du Lisp

Comme en Prolog, 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, l'un d'entre eux joue un rôle particulier :
nil 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. Pour simplifier encore les écritures, tout ce qui n'est pas *nil* peut être considéré comme la valeur de vérité vraie.
Les autres atomes sont les valeurs numériques dont l'évaluation est elle-même, et les chaînes de caractères. Le caractère *t* (true) est la valeur de vérité vraie.

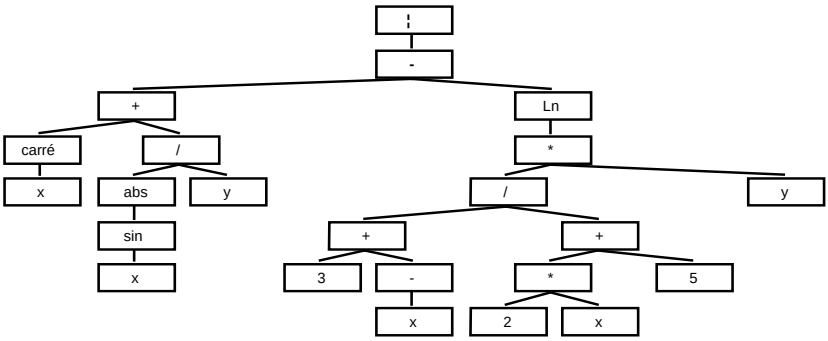
Les listes sont constituées par le plus petit ensemble contenant *nil* et stable par la fonction *cons* (équivalent du Prolog |) définie plus loin, elles sont notées avec des parenthèses. Les listes sont aussi représentées comme doublets d'adresse, celle de l'élément en tête de la liste, obtenue par la fonction *car*, et celle de la queue de la liste obtenue par la fonction *cdr*.
Par exemple (*a b c*), (*x*), ((*a*) *b* (*a b*) ((*c*)) *d*) sont des listes. Le *car* de la première est *a*, celui de la troisième est la liste (*a*). Le *cdr* de la première est la liste (*b c*), celui de la seconde est la liste vide ().

La notation en vigueur pour le Lisp est la notation préfixée et parenthésée. La notation préfixée donne une très grande cohérence au Lisp et les parenthèses sont un moyen extrêmement rationnel de vérification. En effet, chaque couple de parenthèses qui se correspondent délimite une expression évaluable.

Prenons par exemple une expression mathématique *E* 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* carré *x* sin abs *y* / + 3 *x* opp + 2 *x* * 5 + / *y* * Ln − √

On a noté « − » la soustraction de deux arguments et *opp* la fonction *opposé* à un seul argument.
Cette notation est de loin la plus rationnelle, car elle établit une correspondance entre l'ordre d'écriture de gauche à droite et l'ordre opératoire, ceci permettant de débuter les calculs avant même que l'expression soit terminée d'être donnée.

Néanmoins, c'est la notation préfixée (correspondant à la lecture racine-gauche-droite de l'arbre) 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 en Lisp est l'espace, *ab* + *cd* s'écrit donc (+ (* *a b*) (* *c d*)). La liste (1 2 3) possède donc trois éléments alors que (123) n'en possède qu'un.

Comme annoncé plus haut, les sous-expressions délimitées par des parenthèses sont à évaluer en profondeur d'abord. Ainsi le carré de *x*, le sinus de *x* puis la valeur absolue, etc.
Le parenthésage est 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 interprétation très intéressante : chaque parenthèse ouvrante correspond à une descente dans l'arbre représentant l'expression à évaluer et chaque parenthèse fermante à une remontée. Soulignons bien que les parenthèses qui se correspondent délimitent donc toutes les sous-expressions calculables par l'interpréteur Lisp.
Cet interpréteur du Lisp réalise toujours la même boucle, à savoir une lecture de l'expression qu'on lui donne, une évaluation de cette expression et l'affichage de la valeur obtenue.
Dans l'évaluation d'une expression parenthésée (*f a b* …), Lisp cherchera toujours à appliquer, si cela est possible, la fonction *f* sur les valeurs de *a*, *b*, … qui donc elles-mêmes doivent être évaluées au préalable. Ainsi :

(+ 3 4) → 7
(+ (* 2 5) (− 7 3)) → 40
(< (+ 3 4) 5) → nil
(3 + 4) → 3 n'est pas une fonction

Aussi, pour ne pas calculer, pour empêcher cette évaluation, c'est-à-dire prendre une expression telle quelle sans chercher sa valeur sémantique, une fonction bien particulière existe, c'est *quote*, qui, comme son nom l'indique, peut être écrite par une apostrophe sans les parenthèses. C'est la seule exception à la

règle d'écriture que nous ferons pour la raison que son utilisation est si courante que l'écriture de *quote* alourdirait énormément les exemples ci-dessous.
La fonction *eval* réalise le contraire, ainsi (*eval X*) → résultat de l'évaluation de *X*, par exemple : (*eval* '(+ 5 8)) → 13.

Les premières fonctions primitives *car, cdr* et *cons* sur les listes

On peut programmer des problèmes très complexes avec seulement une quinzaine de fonctions de base (*car, cdr, cons, quote, eval, cond, eq, atom...* et bien sûr la définition de fonction *defun*), les autres pouvant se réécrire en Lisp (*and, or, if, null, append, list, member*...) ou d'un usage moins courant (*nth, set, read, print, explode*...).
On donne ces quelques fonctions suivies de quelques exemples. A chaque fois que l'on tape une expression au clavier suivie de la touche *return*, le Lisp renvoie un résultat (qui peut être un message d'erreur). Cet événement sera symbolisé dans ce qui suit, comme dans nos exemples Prolog, par la flèche →.

(*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)
(car '(+ 2 3)) → + En effet c'est le premier élément d'une liste dont on ne demande pas l'évaluation grâce à l'apostrophe

(*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)
(cadr '((a b c d) (e h) f)) → e

(*cons X L*) → liste *L* augmentée de l'élément *X* au début

(cons '*X* '(*Y Z*)) → (*X Y Z*)
(cons (cadr '(3 + 4)) '(3 4)) → (+ 3 4)

Les prédicats *eq, atom, null*

En Lisp, les prédicats ou relations sont leur fonction caractéristique, c'est-à-dire que leur évaluation renverra toujours le vrai ou le faux. Mais en fait, lorsque la valeur renvoyée n'est pas le faux, elle peut toujours représenter le vrai. Ainsi par exemple la fonction *member* reconstruite plus loin sous le nom de *app* 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
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
et bien d'autres fonctions existent, signalons
(*list X Y Z* ...) → liste formée par les éléments *X Y Z*
(*member X L*) → première sous-liste de *L* débutant par *X*
(*append L M*) → liste obtenue par concaténation de *L* et *M*
(*nth N L*) → *N*-ième élément de *L* (à partir de 0)
(*firstn N L*) → liste *L* privée de ses *N* premiers éléments
(*nthcdr N L*) → *N*-ième « cdr » de *L*
(*length L*) → longueur de la liste *L*
(or *L*1 *L*2 *L*3... *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)
(*and L*1 *L*2 *L*3... *Ln*) évalue séquentiellement les *Li* et renvoie *nil* dès que l'une est *nil*, et renvoie (*eval Ln*) sinon. (conjonction logique)

> (list 'a 'b 'c (+ 3 4) (null 'a) (eq (* 2 5) 10)) → (a b c nil t)
> (member '*X* '(*Y U X Z X*)) → (*X Z X*)
> (member '*X* '(*Y U*)) → nil
> (append '(a b) '(c d e)) → (a b c d e)
> (nth 2 '(a b c d)) → c
> (nthcdr 3 '(a b c d e)) → (d e)

```
(length '((a) (b c) ((d)) (a) b) ) → 5
(or (< 4 8) (= 4 7) (> 1 −2) ) → t
(and (< 2 5) (= 3 3) )  → t
```

La fonction à trois arguments *if* évalue son premier argument, si celui-ci est *nil*, c'est-à-dire le faux, alors *if* renvoie l'évaluation de son troisième argument, et dans tous les autres cas, il renvoie l'évaluation de son second argument.

```
(if (eq (+ 2 3) 5 'a 'b) → a
(if (eq (* 2 5) (+ 7 2)) 'a 'b) → b
(if (< x 0) (− x) x) → x   c'est la définition de la valeur absolue
```

La condition *cond*

C'est la fonction 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....) )
```

Cette fonction 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       ((> 2 3) 'non)
            ((eq (* 2 3) 6) 'non)
            ((atom '(a b c)) 'non)
            ((null 'a) 'non)
            (t 'oui)) → oui
```

En effet le premier « test » (> 2 3) est évalué à faux, comme les autres, le dernier, qui 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.

Définition de fonction

La fonction *defun* est bien particulière, elle permet une affectation de fonction. Chaque appel de la fonction pour une liste de valeurs donnée ira chercher le corps de la fonction pour en faire une copie en laquelle les valeurs données sont substituées aux paramètres.
(*defun F P L*) → permet de définir une fonction de nom *F* pour une liste de paramètres *P*, par une liste *L*, ainsi : *(defun cube (x) (* x x x))* pourra s'utiliser par (cube 3) → 27.

Quelques exemples

L'appartenance
```
(defun app (x l) (cond ((null l) nil) ((eq (car l) x) t) (t (app x (cdr l)))))
```

La concaténation
```
(defun append (l m) (if (null l) m (cons (car l) (append (cdr l) m))))
```

Les *n* premiers éléments d'une liste *L*
```
(de tete (n l)      (cond
      ((null l) nil) ; les n premiers termes d'une liste vide 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 ».

Le *n*-ième élément d'une liste
```
(defun nth (n l) (if (eq n 0) (car l) (nth (- n 1) (cdr l))))
```

Naturellement, cette fonction ne s'exerce que si *l* n'est pas vide et que si *n* est compris entre 0 (le premier élément) et la longueur de la liste – 1.

Rang d'un élément dans une liste
```
(defun rg (E L) (if (eq E) (car L)) 0 (1 + (rg E (cdr L)))))
                    ; rang à partir de 0 d'un E dans L
```

Profondeur, longueur de la plus grande branche d'un arbre,
```
(de prof (l) (if (atom l) 0 (max (1 + (prof (car l))) (prof (cdr l))))))
```
(*prof ' (((a)) (y) h (((g n))) j))* → 4

Nombre de feuilles, exemple *(nbfeuilles '(r ((t)) y (g h) (j m l) p))* → 9

```
(defun nbfeuilles (l)     (cond
                          ((nullL l) 0)
                          ((atom l) 1)
                          ((+ (nbfeuilles (car l)) (nbfeuilles (cdr l))))))
```

Ces deux dernières fonctions sont remarquables, comme leurs équivalents Prolog, car dans un langage typé, elles nécessiteraient de long développements avant de pouvoir être mises au point.

Affectations

L'affectation peut naturellement se faire par une fonction particulière *set* pour une variable globale, et par la fonction *let* qui renvoie sa dernière expression calculée au moyen de son premier argument qui doit être une liste de couples d'associations entre noms de variables et valeurs d'initialisation. Par exemple :

(*let* ((*x* 2) (*y* 3) (*z* 1)) (+ (* *x y*) (* *x z*))) → 8

La fonctionnelle *mapcar* permet d'obtenir la liste des images par une fonction d'une liste de valeurs. La fonction est prédéfinie ou bien définie de façon anonyme grâce à lambda.
Ainsi, par exemple :

```
(mapcar 'length ' ((a) (b c) (d e f) (g h i j))) → (1 2 3 4)
(mapcar 'car ' ((a) (b c) (d e f) (g h I j))) → (a b d g)
(mapcar (lambda (x) (* 3 x)) '(1 2 3 4 5)) → (3 6 9 12 15)
(mapcar (lambda (x) (* x x)) '(1 2 3 4 5)) → (1 4 9 16 25)
```

Construction d'un petit Prolog en Lisp

En se limitant à un Prolog simple sans calcul sur des variables, les seuls mots réservés seront *impasse*, *sortie* et *arrêt*.

La fonction principale *muprolog* aura comme arguments une liste de règles *BR* et une question *Q* qui est sous la forme d'une liste de faits que l'on veut voir prouvés.

Elle utilise deux fonctions sans arguments qui demandent à l'utilisateur soit une base de règles *BR*, soit une question *Q*, soit les deux, puis lance la fonction *moteur1*.

```
(defun muprolog (BR Q) (cond ; la procédure principale
        ((null BR) (muprolog (inbase) (inquestion)))
        ((null Q) (muprolog BR (inquestion))))
        (t (moteur1 nil Q Q BR BR))

(defun xcons (X L) (append L (list X)))
        ; apposer un X à la fin d'une liste L

(defun cdl (L) (if (null (cdr L)) nil (cons (car L) (cdl (cdr L)))))
        ; tout sauf le dernier de L

(defun inbase (B) (if (null B)
        (print '"Entrez vos règles comme listes,
                la conclusion en dernier.")
        (print '"Exemple ((= A B) (= B C) (= A C))") )
        (let ((L (read)))
                (if (null L) B (inbase (xcons (append (last L) (cdl L)) B)) )))

(defun inquestion ()
        (print '"Posez une question de la forme ((fils JEAN X) (sortie X))")
        (read))

(defun gensym () (if (boundp 'CO) (concat 'X (incr CO)) (set 'CO 0) 'X0))
```

Cette fonction *gensym* est un générateur de symboles (que nous redéfinissons) qui construit une nouvelle variable à chaque appel, ce sont *X*0, *X*1, *X*2, ... *CO* étant un compteur de ces variables et donc lui-même une variable globale.
En se servant d'une fonction prédéfinie *explode* ou *unpack* suivant les Lisp, on peut convertir une chaîne de caractères en la liste de ses caractères.

```
(defun var (X) (cond
        ; reconnaissance d'une variable (lettres ou lettres suivies d'un chiffre)
        ((listp X) nil)
        ((null (cdr (explode X))) t)
        ((> 65 (cadr (explode X))) t)))
```

```
(defun occ (X L) (cond ; teste la présence de X à un niveau quelconque de L
        ((null L) nil)
        ((atom (car L)) (or (eq X (car L)) (occ X (cdr L))))
        (t (or (occ X (car L)) (occ X (cdr L))) )))
```

Afin d'éviter les conflits de noms de variables, une question posée *Q* comportant des variables aura celles-ci éventuellement modifiées si elles sont également présentes dans une règle *R*. Cette présence est testée par la fonction *occ* et le remplacement se fait grâce à *gensym*.

```
(defun alpha (Q R) (cond
        ; renvoie Q sans variable libre commune avec R (alpha-conversion)
        ((null R) Q)
        ((listp (car R)) (alpha Q (append (car R) (cdr R))))
        ((var (car R)) (if (occ (car R) Q)
                           (alpha (subst (gensym) (car R) Q) (cdr R))
                           (alpha Q (cdr R))))
        (t (alpha Q (cdr R))) ))
```

Nous arrivons maintenant à l'importante fonction d'unification, elle se décompose en *ufp* qui teste s'il y a unification possible entre les listes *E* et *C*, puis *eff* (effacement) et enfin *unif* qui donne le résultat.

```
(defun ufp (E C) (cond ; renvoie vrai ou faux
        ((null C) (if (null E) t))
        ((null E) (if (null C) t))
        ((eq (car E) (car C)) (ufp (cdr E) (cdr C)))
        ((var (car E)) (ufp (cdr E) (cdr C)))
        ((var (car C)) (ufp (cdr E) (cdr C))) ))

(defun eff (F H Q) (cond
        ; renvoie Q dans laquelle F est remplacée par les composantes de H
        ((null Q) nil)
        ((equal (car Q) F) (append H (eff F H (cdr Q))))
        (t (cons (car Q) (eff F H (cdr Q)))) ))
```

A chaque appel de *unif*, l'unification, *Q* et *R* sont alpha-invariants, *E* est le littéral de *Q* à tester et *RQ* est le reste des littéraux de *Q*.
La fonction *ufr* se contente d'appeler *unif* en décomposant la question pour ne pas avoir à redemander sans arrêt ses parties.
Enfin *uf*, grâce à « l'itération » *mapcar*, appelle l'unification *ufp* partout sur la liste *Q* des buts demandés.

```
(defun unif (Q RQ E C H R) (cond
        ; donne le nouvel état obtenu à partir de Q grâce à la  règle R
        ((null RQ) Q)
        ((ufp E C) (cond
                ((null E) (eff (car RQ) H Q))
                ((eq (car E) (car C)) (unif Q RQ (cdr E) (cdr C) H R))
                ((var (car E)) (unif    (subst (car C) (car E) Q)
                                        (subst (car C) (car E) RQ)
                                        (subst (car C) (car E) (cdr E))
                                        (cdr C) H R))
                ((var (car C)) (unif    (subst (car E) (car C) Q)
                                        (subst (car E) (car C) RQ)
                                        (cdr E)
                                        (subst (car E) (car C) (cdr C))
                                        (subst (car E) (car C) H) R))))
        (t (unif Q (cdr RQ) (cadr RQ) (car R) (cdr R) R))))

(defun ufr (Q R) (unif Q (cdl Q) (car Q) (car R) (cdr R) R))

(defun uf (Q R)
        (if (member t (mapcar (lambda (E) (ufp E (car R))) Q))
                (ufr (alpha Q R) R) ; s'il y a au moins une unification possible
                Q)) ; sinon Q n'est pas modifiée
```

La fonction *uf* sera appelée par le moteur, elle fait les substitutions de variables dans *Q* si nécessaire.
Quant aux vraies « fonctions principales » *moteur1* et *moteur2* ce sont celles qui réalisent le backtracking en se passant la main l'une l'autre.

```
(defun moteur1 (LQ Q QU B BR LR) (cond
        ; fait en gros la descente dans l'arbre
        ((eq Q QU) (moteur2 LQ Q B BR LR))
        ((member '(impasse) QU) (moteur2 LQ Q nil BR LR))
        ; on force la remontée
        ((eq (caar QU) 'sortie) (print 'réponse: (cadar QU))
                        (moteur2 LQ Q B BR LR))
        ((eq (caar QU 'arrêt) (print 'réponse: (cadar QU)) 'fini)
                        ; on stoppe l'exploration
        (t (print 'règle: (rg (car B) BR) '--> QU) ; édition de la règle
                (moteur1 (cons Q LQ) QU (uf QU (car BR))
                        (cdr BR) BR (cons (car B) LR) ))))
```

```
(defun moteur2 (LQ Q B BR LR) (if ; fait les remontées dans l'arbre
        (null B) (if (null LQ) 'fini ; toutes les branches sont épuisées
                     (moteur1 (cdr LQ) (car LQ) (uf (car LQ) (car LR))
                     (cdr (member (car LR) BR)) BR (cdr LR)))
                     ; descente relancée
        (moteur1 LQ Q (uf Q (car B)) (cdr B) BR LR)))
        ; cas où B n'est pas vide, descente avec règle suivante
```

1 Retour sur le coloriage d'une carte
On cherche à colorier une carte des pays d'Europe avec 4 couleurs en définissant une relation de voisinage qui est symétrique et antiréflexive, une couleur ne pouvant être voisine d'elle-même.

```
(set ' BC ' (
((bleu vois rouge))
((rouge vois vert))
((vert vois bleu))
((jaune vois bleu))
((rouge vois jaune))
((vert vois jaune))
((X  vois  X) (impasse))
((X vois Y) (Y vois X))
; pour la relation de voisinage, le seul prédicat utilisé est donc « voisin »,
))
```

Pour colorier huit pays avec les couleurs inconnues *A*, *B*, etc., on pose la longue question :

> (muprolog ' *BC* ' ((*F* vois *D*) (*F* vois *I*) (*A* vois *I*) (*H* vois *D*)
> (*B* vois *H*) (*S* vois *F*) (*A* vois *D*) (*D* vois *L*) (*D* vois *S*) (*I* vois *S*)
> (*L* vois *F*) (*B* vois *F*) (*S* vois *A*) (*D* vois *B*) (*B* vois *L*)
> (arret (*FRA : F* ALL *: D BEL : B NED : H LUX : L SUI : S ITA : I AUT: A*)))
> → (*FRA*: bleu *ALL*: rouge *BEL*: jaune *NED*: bleu *LUX*: vert *SUI*: vert *ITA*: rouge *AUT*: bleu)

2 Retour sur un logigram
Le flic nourrit un poulet en cabane, l'épicier se paye un château mais se contente d'un colibri, le prof a un chat. Il y a aussi une grotte et un chien. Où habite le toubib et quel animal possède-t-il ?

On utilise un prédicat quaternaire *dif* indiquant par des *impasses* qui forcent la remontée, que quatre objets doivent être distincts. Nous faisons une affectation de la variable *BR* avec la liste de toutes les règles traduisant l'énoncé grâce aux prédicats unaires *animal*, *maison* et au prédicat binaire *possede*.
Notons que dans la conception de ce Prolog, l'écriture peut se faire comme on veut infixe ou préfixe, l'unification se faisant sur toute une liste.

```
(set 'BR ' (
((dif X X Y Z) (impasse))
((dif X Y X Z) (impasse))
((dif X Y Z X) (impasse))
((dif X Y Y Z) (impasse))
((dif X Y Z Y) (impasse))
((dif X Y Z Z) (impasse))
((dif I J K L))
((prof possede chat))
((flic habite cabane))
((epicier habite chateau))
((flic possede poulet))
((epicier possede colibri))
((maison grotte))
((maison masure))
((maison chateau))
((maison cabane))
((animal poulet))
((animal chien))
((animal colibri))
((animal chat))
((X habite Y) (maison Y))
((X possede Y) (animal Y)) ))
```

On pose la question :
(muprolog *BR* '((toubib habite *A*) (flic habite *B*) (prof habite *C*) (epicier habite *D*) (toubib possede *P*) (prof possede *Q*) (flic possede *R*) (epicier possede *S*) (dif *P Q R S*) (dif *A B C D*) (arret (toubib *A P*))))
→ (toubib grotte chien)

3 Agence matrimoniale
Donnez des faits indiquant simplement pour des prénoms, les sexes, tailles, chevelures et âges. Puis viennent des définitions de prédicats *gouts, recherche, convenir, memes-gouts* et *assortis*.

```
(set 'BM '( ; on invente ici une petite base de données
((Alfred  homme grand brun mur))
((Victor homme moyen blond jeune))
((Hector homme petit brun mur))
((Irma femme moyenne blonde moyen))
((Rosa femme petite blonde jeune))
((Olga femme petite brune mur))
((Alfred gouts class aventurevelo))
((Victor gouts pop sf ski))
((Hector gouts jazz polar ski))
((Irma gouts class aventure velo))
((Rosa gouts pop sf s) (s dif boxe)) ; le sport est tout sauf de la boxe.
((Olga gouts m aventure velo))      ; m peut être instancié par n'importe quoi.
((Alfred recherche grande rousse jeune))
((Victor recherche t blonde jeune) (t dif grande)) ; pour Victor une petite femme
((Hector recherche petite blonde moyen))
((Irma recherche grand brun moyen))
((Rosa recherche moyen blond jeune))
((Olga recherche moyen brun mur)) ; c'est tout pour les faits bruts.

((X dif X) (impasse))       ; ces deux clauses ne peuvent pas être permutées,
((X dif Y))                 ; elles définissent la relation "être différents"

((X convient Y) (X homme T1 C1 A1) (Y femme T2 C2 A2)
                (X recherche T2 C2 A2) (Y recherche T1 C1 A1))
((X meme-gouts Y) (X gouts M L S) (Y gouts M L S))
((X assorti Y) (X convient Y) (X meme-gouts Y))
)) ; cette dernière parenthèse termine l'affectation de BM
```

On lance une recherche et on pourra vérifier à la main qu'il n'y a pas d'autre solution :

```
(muprolog BM
'((X assorti Y) (arret (X et Y se-marièrent-et-eurent-beaucoup-d-enfants
))) )
→ (Victor et Rosa se-marièrent-et-eurent-beaucoup-d-enfants)
```

4 Retour sur Peano
En redéfinissant la relation de successeur pour les premiers entiers nommés par les constantes zéro, un, deux, trois… on reprend les axiomes de Peano en se cantonant à l'addition.

Dans cette version rudimentaire dite *muprolog*, ne comportant en tout et pour tout que trois mots prédéfinis *sortie* pour une édition à l'écran, *arret* de la recherche après un succès et *impasse* pour forcer la remontée, on demande les solutions ou la première solution. Contrairement aux autres Prolog, elle ne fait pas de différence a priori entre les noms de symboles et ceux de prédicats, cela pourrait présenter de grands dangers de tentatives d'unification entre les deux faits *(X dif Y)* et *(superieur X Y)* par exemple, ce qui serait absurde.
Par contre, cela permet de choisir des noms de prédicats fractionnés tels que par exemple *(X plus Y egal Z)* ce qui est plus agréable à la vue que *(plus X Y Z).*

```
(set 'BP '(
((zero suc un))
((un suc deux))
((deux suc trois))
((trois suc quatre))
((quatre suc cinq))
((cinq suc six))

((X suc Y) (impasse))
; il n'y a pas d'autres relations de succession, on se limite à l'intervalle de 0 à 6.
((X plus zero egal X))
; expriment les axiomes de Peano x + 0 = x et x + (y + 1) = (x + y) + 1
((X plusY egal Z) (U suc Y) (X plus U egal V) (V suc Z))
))
```

On utilise maintenant cette base *BP* :

(muprolog *BP* '((un plus deux egal *X*) (arret *X*)) *BP*) → trois
(muprolog *BP* '((deux plus *X* egal cinq) (arret *X*)) *BP*) → trois

(muprolog BP '((*X* plus *Y* egal deux) (sortie (*X Y*))) BP)
→ (deux zero), (un un), (zero deux)

(muprolog *BP* '((trois plus deux egal *X*) (arret *X*)) *BP*) → cinq