

Université PARIS VII

IREM

INSTITUT DE RECHERCHE SUR L'ENSEIGNEMENT DES MATHÉMATIQUES

53

LISP
ET
PROLOG
EN CLASSE TERMINALE

PAR J.P. ROY

<u>NIVEAU</u>	Second cycle
<u>PUBLIC</u>	Enseignants/Elèves du 2ème cycle
<u>SUJET</u>	Les langages LISP et PROLOG. Un cours donné en Terminale
<u>OBJECTIF</u>	Aide à l'expérimentation

LISP et PROLOG

en

classe

terminale

(C) IREM DE PARIS-SUD
1983-84

- S O M M A I R E -

0. Introduction

I. Programmation en LISP (octobre-février)

I.1 Introduction à LISP (1ère partie)

I.2 Introduction à LISP (2ième partie)

I.3 Du Lisp et des élèves

I.3.1 PLOT : une fonction Lisp basse-résolution sur Apple II

I.3.2 Utilisation des fonctions du BDOS de CP/M sous Le_Lisp 80

I.3.3 Techniques de filtrage

I.3.4 Le robot FREDDY : prolongements

I.3.5 Un micro-BASIC sous Lisp

I.3.6 Un système déductif sans variables

I.4 Compléments : au-delà du filtrage, l'unification

II. Programmation en PROLOG (mars-juin)

II.1 Pourquoi la Logique et quelle Logique ?

II.2 Le système PROLOG II et son environnement Pascal

II.3 Programmer en PROLOG

II.3.1 Gérer des informations non numériques

II.3.2 L'arithmétique de Peano

II.3.3 Les listes

II.3.4 Les prédicats évaluables

II.3.5 Quelques thèmes d'étude

III. Perspectives. Bibliographie

0 - I N T R O D U C T I O N -

Ce travail, que l'IREM de Paris-Sud a bien voulu publier, est directement issu d'un cours donné à des élèves de classe Terminale durant l'année scolaire 1983-84 et destiné à les introduire à une programmation applicative et axée sur des traitements symboliques, plutôt que numériques ou graphiques.

Les élèves en question - dont le nombre a été limité à une douzaine pour des raisons d'encadrement et de matériel - avaient déjà une expérience préliminaire de programmation en LOGO (niveau débutant) en classe de Seconde, puis de Pascal en classe de Première (orientée plus spécialement vers les tableaux et les chaînes de caractères; la version choisie était celle du Pascal-UCSD sur Apple II, particulièrement fiable et très bien adaptée à une utilisation pédagogique).

Pourquoi avoir changé d'orientation de programme, et pourquoi ces langages "d'un autre type" que sont LISP et PROLOG ?

Il est depuis quelques années fortement question d'une option "Informatique" au Baccalauréat. C'est dans cette optique que le cours facultatif "Informatique et Programmation" fut ouvert dans notre Ecole aux élèves (volontaires) des classes du Second Cycle. Malgré des difficultés prévisibles, le cours s'est mis en place, et au moment d'aborder pour la première fois le niveau de classe Terminale, il a fallu penser le matériel à enseigner, faute d'ouvrage de référence. Les recommandations du Ministère demandaient essentiellement de manipuler des structures de données un peu plus sophistiquées que les tableaux, que sont les arborescences, et des structures de contrôle tournant autour du concept de récursivité.

La programmation récursive avait déjà été abordée, dès la Seconde avec LOGO, puis en Première - à une bien moindre échelle - avec Pascal. Le problème n'était donc pas au rendez-vous de ce côté-là. Par contre, manipuler des arborescences - une activité fondamentale en Informatique, de l'écriture des programmes à la pratique des systèmes modernes d'exploitation - n'est pas quelque chose d'intrinsèquement facile et suppose des structures de données bien adaptées et "confortables". La chaîne de caractères LSE - structure plate par excellence - ne nous apparaît pas adaptée à ce genre de voltige. S'il est vrai qu'il est parfaitement possible de générer des arbres en "tableaux" par des jeux d'indirections, ceci relève davantage d'un travail de professionnel, en tout cas de quelqu'un qui fait tout pour en devenir un. Ce qui n'est pas - jusqu'à plus ample informé - le but cherché dans cette option Informatique; former les élèves des années 80 à une nouvelle discipline scientifique, c'est surtout contribuer à leur formation tout court, et tant pis ou tant mieux s'ils abandonnent la programmation par la suite. Ce qui doit rester, c'est la satisfaction d'avoir pu expérimenter des réponses à des questions telles "Qu'est-ce que résoudre un problème ?" ou "Comment puis-je exprimer ceci ?", toutes questions qui ne sont plus de mise dans des classes de Mathématiques condamnées au Baccalauréat. Mais ne dévions pas...

Si la chaîne de caractères et le tableau sont écartés, que reste-t-il ? Les pointeurs ! Disponibles en Pascal, ils permettent effectivement de définir des structures de données récursives, et de les manipuler de façon non moins récursive. Pascal est un magnifique jeu de construction. Les structures de données s'emboîtent, se superposent, se raccordent à volonté pour le plus grand bonheur du programmeur et la plus grande perplexité de l'élève: trop d'abondance nuit.

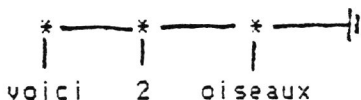
Il se trouve qu'il existe depuis l'aube de l'Informatique - contemporain de FORTRAN et antérieur à BASIC, Pascal ou LSE - un langage qui précisément ne manipule QUE des pointeurs. Enfin, presque: il dispose aussi des nombres et des identificateurs. Sa syntaxe est réduite à l'extrême, et ses structures de contrôle peuvent être constituées et étendues à l'intérieur même du langage. Bien plus qu'il n'en faut pour faire toute sorte de programmation (tout naturellement récursive) axée sur la notion de hiérarchie. Ce langage venu du fin des âges, qui a finalement si peu changé depuis sa création (n'est-ce pas un signe de qualité ?) et a suscité des travaux parmi les plus brillants que l'Informatique ait connus, a pour doux nom LISP : LIST Processing, la machine à "traiter les listes".

L'entité suivante fournit un exemple de liste:

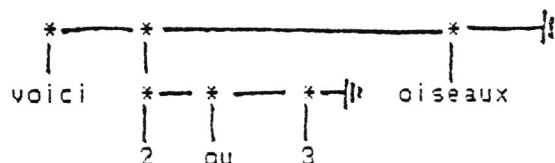
(voici 2 oiseaux)

un peu comme la chaîne de caractères "voici 2 oiseaux", mais dont les éléments peuvent eux-mêmes être des listes:

(voici (2 ou 3) oiseaux)

De l'architecture linéaire:  , on passe à l'archi-

tecture ramifiée:



La question de la représentation des données étant réglée, à la grande satisfaction des élèves (mais si, ils s'adaptent très vite), il reste à programmer, à trouver des algorithmes, des manières de représenter l'information et de la manipuler.

Décider de la manière de représenter l'information est capital. A deux niveaux: pour les données d'un programme, bien sûr, mais aussi, pourquoi pas, pour le programme lui-même. Finalement, qu'est-ce que "programmer", sinon mettre en clair des connaissances précises sur un sujet donné.

Comment peut-on effectuer un tri ?

- Prenons une variable d'index initialisée à 0, puis... Non. Evitons les "cases-mémoire", trop liées à une certaine architecture d'ordinateur (nous nous plaçons dans une optique pédagogique, ne l'oublions pas).
- Insérons le premier élément à sa place dans le reste trié auparavant. Mieux! Vous exprimez impérativement le tri en disant "faites ceci ou cela". Du bon classique, en Pascal ou LSE par exemple.
- Disons que le tri de L, c'est: si L est vide, alors le vide, sinon c'est l'insertion de son premier élément dans le tri du reste. Encore mieux. Votre "programme" devient une fonction composée, vous programmez de manière "appliquative". Vous serez parfaitement à l'aise en LISP.
- Exprimons que L' est la version triée de L s'il existe entre L et L' la relation suivante: ... Inhabituel, mais très bon. Le schéma classique:

```

-----
L --->      p(L)      ---> L'
-----

```

devient:

```

-----
L --->      r(L,L')      ---> vrai
L' --->      ---> faux
-----

```

Vous programmez en Logique, et le langage PROLOG vous attend.

Ceci n'est pas un test psycho-informatique à faire sur les plages, mais tend à donner une vue rapide sur l'évolution des idées "universitaires" en ce qui concerne le concept de "programme", ou de "calcul".

La programmation impérative classique est maintenant bien maîtrisée et enseignée comme discipline à part entière; la notion d'algorithme et de complexité commence à être bien cernée, branche des mathématiques appliquées utilisant toutes les ressources disponibles, de la combinatoire aux fonctions méromorphes. FORTRAN cède irrésistiblement la place à Pascal (créé par N. Wirth) qui tend à s'imposer comme langage d'enseignement "universel" (d'autant plus que des versions très rapides et interprétées sont maintenant disponibles), et adapté à la programmation structurée (travaux de Dijkstra, Bohm et Jacopini vers 1968).

La programmation fonctionnelle reste très étudiée depuis la conception de LISP par J. McCarthy vers 1958, en passant par les travaux formalisants de J. Backus vers 1978. Elle suscite un regain d'intérêt depuis l'apparition de nouveaux axes de recherche sur les architectures "parallèles". Mais son caractère indéracinable tient avant tout à ses avantages sur les plans formel (un programme est une "fonction") et didactique (LOGO).

La programmation en Logique (travaux de R. Kowalski) aboutit vers 1972 à la création à Marseille du langage PROLOG (Colmerauer, Roussel), qui sera choisi dix ans plus tard comme langage-noyau du projet d'ordinateurs japonais de 5ème génération (Intelligence Artificielle). Sa didactique n'a jusqu'à présent été tentée que sur de jeunes lycéens (à-partir de 11 ans, par R. Ennals et R. Kowalski).

Toujours est-il que les langages LISP (Octobre-Février) et PROLOG (Mars-Juin) furent les supports du cours 1983-84. Dois-je préciser que je préparais cette année-là un D.E.A. d'Informatique (Intelligence Artificielle) et que le choix n'était donc pas aussi innocent. En fait, ce document devait initialement servir de rapport de stage sous la direction du Professeur J. F. Perrot, mais l'idée d'un ouvrage plus complet sur LISP - édité par CEDIC/Nathan - a quelque peu bouleversé ce schéma, puisque le rapport de stage devenait l'ouvrage sur LISP, ou inversement. En accord avec F. Colmez, qui dirige notre IREM, j'ai poursuivi la rédaction des notes et des fichiers accumulés avec les élèves pendant l'année et ce qui reste est un peu "batard".

La partie sur LISP débute par une introduction que j'avais destinée à une revue d'Informatique (Micro-systèmes), très "portable" et nous servant de seul support de cours, tandis que les travaux et projets d'élèves utilisent assez la version Le_Lisp 80 de J. Chailloux et E. Saint-James, disponible sur les machines de l'Education Nationale (CNDP). Il n'y a donc pas de progression régulière en LISP, seulement du matériel brut montrant ce qu'il est envisageable

de faire avec des élèves de Terminale.

Quant à PROLOG, l'absence (désolante) d'ouvrages introductifs en Français a rendu plus impératif un texte ne supposant aucune connaissance préalable. Il reste néanmoins que l'exposé en question reprend celui d'un cours, et non du langage proprement dit dans sa généralité et sa puissance. Il ne s'agit encore une fois que d'une introduction.

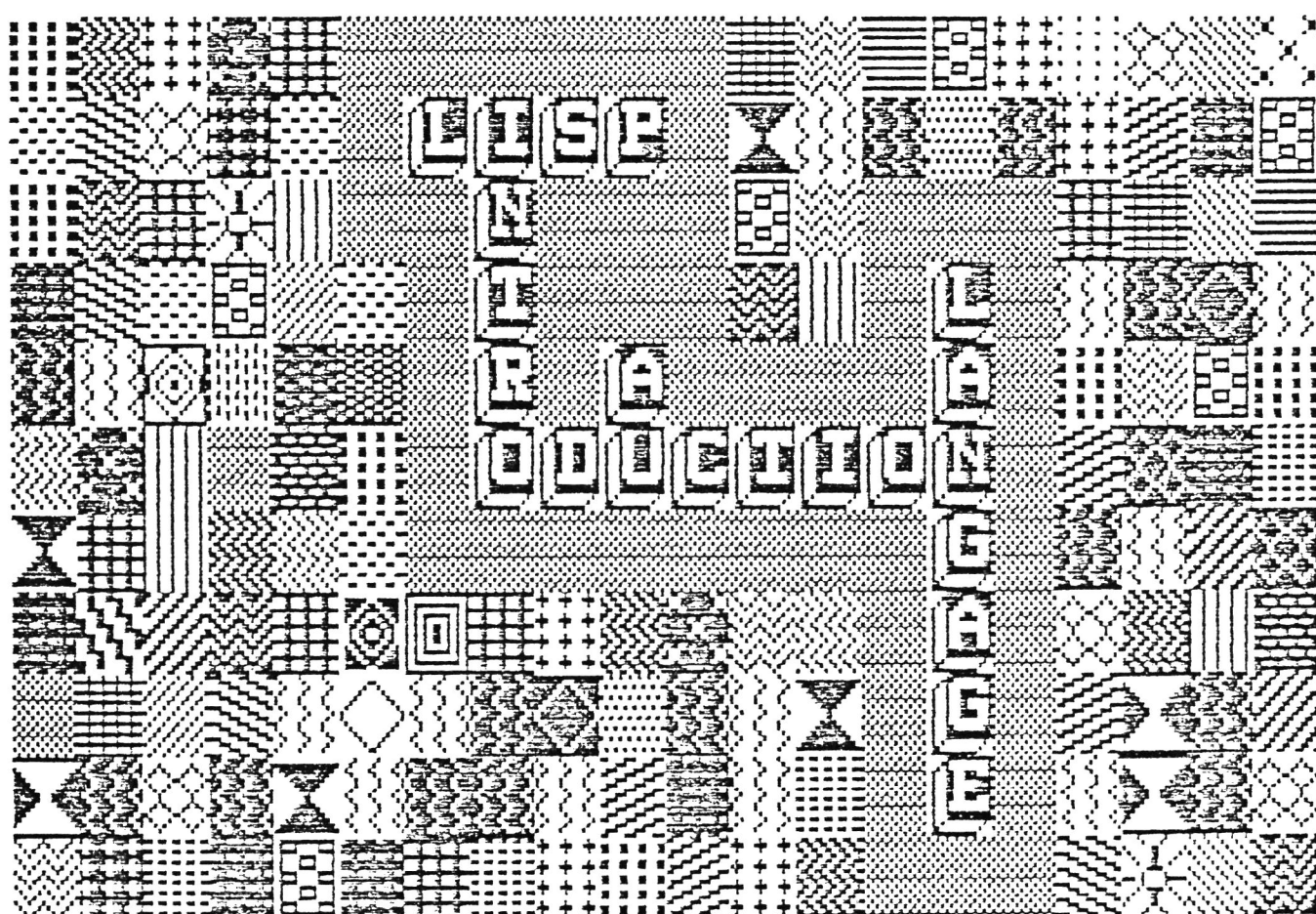
Ce polycopié, comme il est écrit sur la page de couverture, a pour but: servir à l'expérimentation. Mon souhait le plus vif est bien entendu qu'il y réussisse, et que le matériau qu'il contient serve aux collègues intéressés. Qu'ils me renvoient l'ascenseur, je le prendrai sûrement en continuant ce cours...

Un dernier mot en direction des "matheux": étudiez LISP et PROLOG. Leur esthétique devrait vous ravir.

Laboratoire d'Informatique
Ecole Alsacienne
109, rue N.D. des Champs
75006 PARIS

Paris, le 10 septembre 1984

N.B. Ce texte a été entièrement composé sur un micro-ordinateur, et une expérience antérieure (LOGO) m'a montré que le meilleur traitement de texte ne met pas à l'abri des erreurs dues aux versions successives. Je suis par avance désolé du tracis qu'elles pourraient vous causer.



I.1

Introduction à LISP
(1ère partie)LES PREMIERS PAS

L'Intelligence Artificielle (I.A.) a déjà démarré depuis quelques années, et le traitement des expressions symboliques a pris dans ce domaine le pas sur celui des quantités numériques.

La structure de "liste" (cf. infra) commence à être appréciée, surtout à travers deux réalisations (1956-58):

- i) IPL ("Information Processing Language") de Newell, Shaw et Simon, à la Rand Corporation (travaux sur la démonstration automatique en Logique: The Logic Theorist);
- ii) FLPL ("FORTAN List Processing Language") par une équipe d'IBM (en particulier Gelernter, travaux sur la démonstration automatique de théorèmes en géométrie plane).

A-partir de 1958, une équipe de chercheurs du M.I.T., dans le cadre du "M.I.T. Artificial Intelligence Project", et dirigée par John Mac-Carthy, va entreprendre les premiers travaux théoriques devant aboutir à l'implémentation de LISP 1 sur l'IBM 704. Plusieurs versions viendront ensuite, dont LISP 1.5, 1.6, puis différents dialectes: MACLISP, INTERLISP, FRANZLISP, VLISP, LE_LISP, et même des tentatives plus pédagogiques comme LOGO.

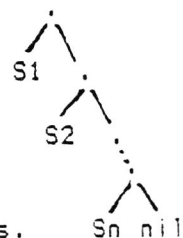
Leurs recherches débouchent sur un certain nombre de structures et de principes de base; nous allons donner dans un premier temps une présentation de "LISP pur"[3], réduit au minimum et épuré des problèmes de syntaxe. Nous verrons par la suite comment rédiger de véritables "programmes" LISP.

L I S P P U R

La structure de données essentielles en LISP est la "S-expression", ou expression symbolique, que nous nommerons "expression" tout court. La classe des expressions, objets généraux manipulés par LISP, se scinde en deux sous-classes:

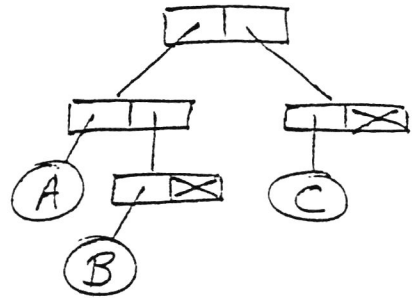
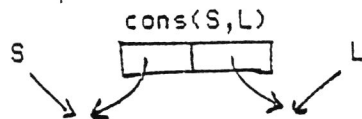
- celle des "atomes", composée des "symboles" (identificateurs comme X, X1, 1X, ATOM, +, !!!) et des "nombres" (-421, 8190). Il y a en plus deux atomes spéciaux T (le "vrai") et NIL (le "faux", ou la liste vide, cf. plus bas);
- celle des "listes", une liste étant une suite ordonnée d'expressions insérée dans une paire de parenthèses, comme par exemple (note 1):
(+ (COS X) 3) ou (A (B (C D))) ou ((1 2) nil (4))

On écrit () l'atome NIL, et on dit qu'il s'agit de la "liste vide". Cet atome sert de "terminateur" de liste lorsque l'on représente une liste (S1 S2...Sn), ou les Si sont des expressions, par l'arbre binaire ci-contre, chaque Si étant passible du même traitement (un atome étant représenté par lui-même, feuille). La figure 1 détaille les listes précédentes.



Afin d'obtenir un traitement récursif des listes (définies récursivement ci-dessus), on dispose de deux fonctions d'accès CAR et CDR, d'une fonction de construction CONS, et de deux prédicats (fonctions booléennes) ATOM et EQ, dont le fonctionnement est expliqué dans la figure 2.

Quant à la représentation en mémoire, elle se fait par l'intermédiaire de "pointeurs", une liste L étant représentée par un "doublet" d'adresses, celle de son CAR et celle de son CDR. Une liste n'est donc qu'une succession d'adressages indirects butant sur des atomes; si l'on représente NIL par une croix, la liste $\langle\langle A B \rangle C\rangle$ est chaînée en mémoire suivant le schéma ci-contre. Par exemple, la liste $\text{cons}\langle S, L \rangle$ est construite en mémoire en déposant en "zone libre" un doublet pointant là où pointent S et L:



La conditionnelle

C'est une généralisation du IF...THEN...ELSE... des langages classiques. Si $t_1, \dots, t_n$ sont des "tests" et $S_1, \dots, S_n$ des expressions, on note "COND" la fonction définie ainsi: $\text{cond}(t_1 \rightarrow S_1, t_2 \rightarrow S_2, \dots, t_n \rightarrow S_n)$ retourne la première expression S_i telle que $t_i = T$. Si tous les tests t_i valent NIL, alors COND retourne NIL. Il s'agit donc de:

```

si t1 alors S1
  sinon si t2 alors S2
    sinon ...
      ... sinon si tn alors Sn
        sinon nil
  
```

Chaque " $t_i \rightarrow S_i$ " est une "clause" de la conditionnelle, la dernière étant écrite en général " $T \rightarrow S_n$ " (clause "otherwise").

```

Ex: abs(x) = cond(x > 0 → x, T → -x)
    n! = cond(n = 0 → 1, T → n*(n-1)!)
    pgcd(n,m) = cond(n < m → pgcd(m,n), n mod m = 0 → m, T → pgcd(m, n mod m))
    p ou q = cond(p → T, T → q)
    p et q = cond(p → q)
    non p = cond(p → nil, T → T)
    p => q = cond(p → q, T → T)
  
```

Cette fonction permet de définir de nouvelles fonctions, comme celles de la figure 3.

La lambda-notation fonctionnelle

Due au logicien A. Church, elle sert à formaliser la notion de "variable muette". Nous parlons tous de la "fonction" $f(x)=x+3$ en confondant la fonction f avec sa valeur $f(x)$ au point x . Nous l'écrirons $\text{lambda}((x), x+3)$, ce qui revient d'ailleurs au même que $\text{lambda}((u), u+3)$, etc... Il peut y avoir plusieurs arguments, comme dans $\text{lambda}((x y), x+y-m)$. Une telle "lambda" comporte donc une liste de symboles ("arguments"), ici $(x y)$, et un "corps" $x+y-m$. On dit que les variables x et y figurant dans le corps $x+y-m$ sont "liées" à cette lambda, la variable m étant "libre" (dans cette lambda). En prenant des risques, on peut penser aux variables locales et globales dans le corps d'une procédure Pascal.

La valeur en (2 5) de $\text{lambda}((x y), x+y-m)$, dans un environnement de variables avec $m=-1$, sera $\text{lamoda}((x y), x+y-m) (2 5) = 8$. On peut, grâce à cette lambda-notation, décrire une fonction (la suivante calcule le N-ième élément d'une liste L):

```

nth = lambda((N L), cond(eq(N,1) → car(L),
                        T → nth(N-1, cdr(L))))
  
```

PROGRAMMES = DONNÉES : LE LISP PARLÉ

Décrivons maintenant la syntaxe de LISP. Nous nous référons, si besoin est, au dialecte Le_Lisp 80, qui est la version officielle de l'Education Nationale et qui va sans doute devenir l'une des versions les plus utilisées en France, car portable sur micro-ordinateurs à base de Z-80 (par exemple Apple II sous CP/M), mais aussi, avec la même syntaxe, sur de plus gros processeurs (68000, 8086-88), ou sur des machines comme Perkin-Elmer 32 ou VAX (sous UNIX).

La seule structure de base étant la S-expression, le programme lui-même sera une suite d'expressions! Quand on parle de LISP, on sous-entend en fait "l'interprète" du même nom (LISP est rarement compilé sur petites machines); une "session" LISP consiste à fournir une expression à l'interprète qui va l'EVALUER suivant des règles bien précises. Autrement dit, LISP n'est qu'une boucle:

... -> LIRE -> EVALUER -> ECRIRE -> ...

évaluer une expression consistant à lui appliquer une super-fonction EVAL:

```
"LISP"
S -----> eval(S)
```

la fonction EVAL étant simulée (en LISP !) dans les bons manuels de référence.

Un langage applicatif

Comment l'interprète perçoit-il une liste (F S1 S2 ... Sn) ? De manière "applicative", i.e. sous la forme F(S1,S2,...,Sn); la fonction EVAL va donc "évaluer" les arguments S1,...,Sn successivement (note 2), puis appliquer la fonction F qui est, soit une primitive du système, soit une fonction définie par l'utilisateur (une "lambda"). Nous écrirons donc (eq x y) là où nous écrivions eq(x,y) par exemple.

Considérons les exemples de l'encadré 4, et détaillons par exemple le mécanisme de la 4^{ème} évaluation: une liste à deux éléments, le car étant une "lambda", le cdr (formant la liste des arguments) étant (3). La variable N est "muette" et liée à la lambda, mais le symbole N peut avoir une certaine valeur au moment du calcul. L'interprète va alors "sauver" cette valeur dans sa pile de contrôle, puis "lier" le symbole N à l'argument 3, enfin évaluer le corps de la lambda avec N=3. Après obtention du résultat, il ne restera plus qu'à effacer cette liaison, restaurer l'ancienne valeur de N (sur le sommet de la pile), imprimer le résultat, et attendre une nouvelle expression...

Qui peut, comme dans le dernier exemple, être un nombre: sa valeur est alors ce nombre lui-même.

L'échec de l'interprète sur la liste (A B C) invite à exiger une fonction "identité" qui retourne SANS L'EVALUER toute expression qui lui est passée comme argument; on la nomme QUOTE (cf. figure 5).

L'affectation

En principe inutile dans une pure optique applicative, personne ne s'en passe. L'affectation des symboles se fait grâce à la fonction "set" qui évalue ses arguments (cf. figure 6). Plus utilisée est "setq" qui n'évalue pas son premier argument (le "q" est là pour "quote").

En parlant d'affectation, il faut aussi évoquer la possibilité de DÉFINIR en LISP ses propres fonctions. Il existe à cet effet une fonction "def" qui va "affecter" une lambda à un symbole; la factorielle s'écrit par exemple (notez au passage la vraie syntaxe du "cond"):

```

? (de fac (N) ; N est "muet" et lié à la lambda dont le corps suit:
  (cond ; (cond (t1 S1) (t2 S2) ... (tn Sn))
    ((eq N 0) 1)
    (T (* N (fac (- N 1))))))
= fac ; la valeur de retour de la fonction "de"
? (setq X 3)
= 3
? (+ 2 (fac X)) ; la fonction "fac" évalue son argument X
= 8

```

Entraînez-vous à traduire en vrai LISP les fonctions de la figure 3. Voici encore "equal" par exemple:

```

(de equal (S1 S2)
  (cond
    ((atom S1) (and (atom S2) (eq S1 S2)))
    ((atom S2) nil) ; car à ce niveau S1 n'est pas un atome
    (T (and
      (equal (car S1) (car S2))
      (equal (cdr S1) (cdr S2))))))

```

En plus de ces fonctions définies par "DE" et qui évaluent leurs arguments, il y a des fonctions définies par "DF" qui n'évaluent pas leurs arguments, ainsi que des "macros" définies par "DM" qui "fabriquent" la fonction juste avant de l'exécuter. La figure 7 en fournit quelques exemples.

Signalons enfin que LISP est un langage très riche et extrêmement puissant pour toutes applications non numériques (note 3) et symboliques (note 4), plus particulièrement orientées vers la simulation. La récursivité y joue un rôle majeur, et les interprètes LISP actuels savent l'éliminer (i.e. la transformer en boucle "while" sans saturer la pile) dans bien des cas. Notons enfin que, au moins sur grosses machines, plusieurs compilateurs LISP rivalisent avec leurs collègues FORTRAN quant à la rapidité d'exécution.

De grandes absentes dans cet exposé: les "listes de propriétés" (p-listes). Alors que pour la plupart des langages, un identificateur n'a qu'une propriété, à-savoir sa valeur, LISP attache à chaque symbole une liste de la forme:

```
(id1 val1 id2 val2 ..... idn valn)
```

dans laquelle chaque idi est un symbole de propriété et vali la valeur correspondante. Si l'atome "lisp" a pour p-liste:

```
(date 1958 créateur Mac-Carthy style applicatif)
```

il sera possible de demander la valeur de la propriété "créateur" de l'atome "lisp" en posant ainsi la question:

```

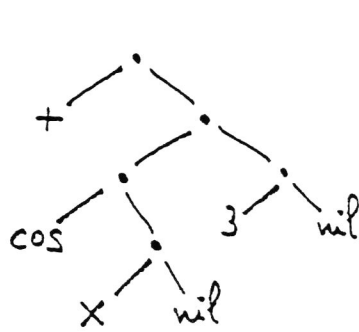
? (getprop 'lisp 'créateur)
= Mac-Carthy

```

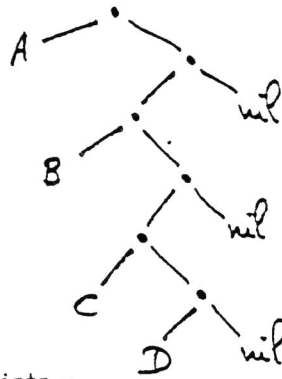
Certaines fonctions sont chargées de la gestion de ces p-listes, dont nous parlerons un peu dans l'article suivant. L'ouvrage [1] en montrera une application fondamentale en représentation des connaissances par un système de "frames" (réseau associatif d'informations).

Nous espérons que ce petit aperçu vous aura au moins donné envie de vous essayer à ce type de programmation très original, et en tout cas fort enrichissant. Nous tacherons de vous en convaincre - si ce n'est déjà fait - dans un numéro ultérieur en donnant des applications typiques à l'utilisation de LISP.

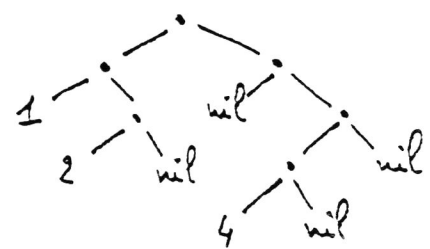
(+ (COS X) 3)



(A (B (C D)))



((1 2) () (4))



- Figure 1 : l'arbre binaire associé à une liste -

CAR et CDR: si $L = (S_1 S_2 \dots S_n)$ est une liste non vide, on note $\text{car}(L)$ son premier élément (une expression) et $\text{cdr}(L)$ le reste de la liste:

L	(+ (COS X) 3)	(A (B (C D)))	((1 2) () (4))
$\text{car}(L)$	+	A	(1 2)
$\text{cdr}(L)$	((COS X) 3)	((B (C D)))	(nil 4)

On étend la notation en posant $\text{cadr}(L) = \text{car}(\text{cdr}(L)) \dots$. Par exemple, le cadr de la première liste est l'atome X. Notez que le cdr d'une liste est toujours une liste (dans l'optique de la note 1), le car étant une expression.

CONS: n'est pas une insulte, mais une fonction permettant de construire une nouvelle liste $\text{cons}(S, L)$ à partir d'une expression S et d'une liste L, en insérant S en tête de L (les expressions S et L étant laissées intactes):

S	L	$\text{cons}(S, L)$
1	(2 3)	(1 2 3)
(A B)	(C D)	((A B) C D)
S	nil	(S) donc $\text{cons}(S, \text{nil}) = (S)$ note list(S)
nil	nil	(()) la liste réduite à la liste vide

On a donc : $\text{car}(\text{cons}(S, L)) = S$ et $\text{cdr}(\text{cons}(S, L)) = L$. De plus, en parcourant de manière préfixe (noeud, gauche, droite) tout arbre binaire de l'encadré 1, et en remplaçant le "." par "cons", on vérifie que l'on récupère bien les listes associées. Pour la liste $(+ (\text{COS } X) 3)$ par exemple, il vient par une lecture préfixe de l'arbre $.(+, .(.(\text{COS}, .(X, \text{nil})), .(3, \text{nil})))$ i.e. $\text{cons}(+, \text{cons}(\text{cons}(\text{COS}, \text{cons}(X, \text{nil})), \text{cons}(3, \text{nil})))$, que l'on réduit ensuite.

On note aussi $\text{list}(S_1, \dots, S_n)$ au lieu de $\text{cons}(S_1, \text{cons}(S_2, \text{cons}(\dots, \text{cons}(S_n, \text{nil}) \dots))$ avec par exemple $\text{list}(1, 2, 3) = (1 2 3) = \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$.

ATOM et EQ: ce sont les deux prédicats de base. $\text{atom}(S)$ vaut T ("vrai") si et seulement si l'expression S est un atome, nil sinon:

S	A	-421	+	T	()	(A)	(A B C)
$\text{atom}(S)$	T	T	T	T	T	nil	nil

La fonction EQ teste l'égalité de deux atomes, caractère à caractère:

$\text{eq}(X, X) = T$ $\text{eq}(X, Y) = \text{nil}$ $\text{eq}(T, \text{nil}) = \text{nil}$ $\text{eq}(\text{nil}, \text{nil}) = T$ $\text{eq}(-12, -12) = T$
et n'est pas définie si l'un au moins des deux arguments n'est pas un atome.

- Figure 2: les 5 fonctions de base -

l'égalité de deux expressions S1 et S2:

```
equal(S1,S2) = cond(atom(S1) → atom(S2) et eq(S1,S2),
                    atom(S2) → nil,
                    T → equal(car(S1),car(S2)) et equal(cdr(S1),cdr(S2)))
et on note alors null(S) = equal(S,nil).
```

le nombre d'éléments d'une liste L:

```
length(L) = cond(null(L) → 0, T → 1 + length(cdr(L)))
et par exemple, si L = (1 (2 3) 4), alors length(L) = 3.
```

la fonction de concaténation de deux listes:

```
append(L1,L2) = cond(null(L1) → L2,
                     T → cons(car(L1),append(cdr(L1),L2)))
e.g. si L1 = (1 2) et L2 = (3 (4)), alors append(L1,L2) = (1 2 3 (4))
```

un prédicat d'appartenance:

```
member(S,L) = cond(null(L) → nil,
                   T → equal(S,car(L)) ou member(S,cdr(L)))
e.g. si L = (3 (4) 5), alors member(5,L) = T et member(4,L) = nil.
```

une fonction d'inversion d'une liste L:

```
reverse(L) = cond(null(L) → nil,
                  T → append(reverse(cdr(L)), list(car(L))))
e.g. si L = (A (B C) D), alors reverse(L) = (D (B C) A), on dit que la
liste L vient d'être inversée "au premier niveau".
```

- Figure 3 : exemples de fonctions en LISP pur -

```
? (eq 2 3)           ; ceci est un commentaire, ignoré par LISP
= nil                ; la réponse après évaluation de (eq 2 3)
? (cdr (cons 1 (cons 2 (cons 3 nil)))) ; i.e. cdr de (1 2 3)
= (2 3)
? (null (cdr (list 2)))
= T
? ((lambda (N) (+ N 1)) 3)
= 4
? (A B C)
*** Erreur: fonction indéfinie A
? -5
= -5
```

- Figure 4 : une micro-session LISP -

```
? (quote (A B C))
= (A B C)           ; l'argument (A B C) n'a pas été évalué
? '(A B C)          ; l'abréviation usuelle
= (A B C)
? 'A
= A
? (cons 'A '(B (C D)))
= (A B (C D))
```

- Figure 5 : la fonction QUOTE -

```
? (set 'X 2)      ; donner à X la valeur 2. L'évaluation de 'X est X
= 2              ; valeur de retour de la fonction "set"
? (setq X 3)      ; la forme "quotée": X n'est pas évalué
= 3
? (setq L '(A B C))
= (A B C)
? L              ; on vérifie ?
= (A B C)        ; ok
? (set (car L) (cons -1 (cdr L))) ; donner à A la valeur (-1 B C)
= (-1 B C)
? A
= (-1 B C)       ; mais L est inchangée, bien sur
```

- Figure 6 : l'affectation des symboles -

Les EXPR : ce sont les fonctions écrites en LISP et qui évaluent leurs arguments, comme "fac" ou celles de l'encadré 3 (à la syntaxe près). La fonction "de" en permet la définition. Voici une fonction qui réalise le tri d'une liste L par insertion (tri des joueurs de cartes):

```
? (de tri (L)
  (cond
    ((null L) ()) ; la liste vide est déjà triée
    (T (ins (car L) (tri (cdr L))))) ; insérer le 1er element dans le
= tri ; reste de la liste, supposé trié
? (de ins (at L) ; insérer l'atome "at" à sa place dans la liste triée L
  (cond
    ((null L) (list at)) ; (list 2) = (2)
    ((< at (car L)) (cons at L)) ; si "at" < (car(L)), le mettre en tete
    (T (cons (car L) (ins at (cdr L))))) ; sinon avancer dans la liste
= ins
? (tri '(4 2 6 1 -1 2))
= (-1 1 2 2 4 6)
```

Certaines EXPR peuvent aussi avoir un nombre quelconque d'arguments, on les nomme alors LEXPR, mais nous n'en parlerons pas ici.

Les FEXPR : ce sont des fonctions écrites en LISP, mais qui n'évaluent pas leurs arguments. Voici par exemple une fonction INCR qui incrémente de 1 son argument:

```
? (df incr (s)
  (set s (+ 1 (eval s))))
= incr
? (setq x 2)
= 2
? (incr x)
= 3
? x
= 3
```

Notez que ceci n'est pas "normal" si vous pensez naïvement à écrire la fonction EXPR suivante:

```
? (de incr (s)
  (setq s (+ s 1)))
= incr
? (setq s 2)
= 2
? (incr s)
= 3 ; le changement...
? s
= 2 ; dans la continuité ! (seule la VALEUR de s est passée...)
```

Les MACROS : ce sont des fonctions écrites en LISP, mais plus spéciales:

- 1) c'est l'appel tout entier qui est lié à la liste des variables;
- 2) le mécanisme d'évaluation se fait "en deux temps":
 - génération du code de la fonction;
 - exécution du code engendré.

Voici par exemple une "macro" (INCR <symbole>), où <symbole> a pour valeur un nombre, qui retourne ce nombre augmenté de 1, tout en affectant cette nouvelle valeur à <symbole>:

```
? (dm incr (bid s) ; dans (incr X), bid sera liée à incr et s à X
  (list 'setq s (list '+ s 1)))
= incr
? (setq X 3)
= 3
? (incr X)
= 4
? X
= 4
```

La MACRO génère dans un premier temps (setq X (+ X 1)), puis évalue ensuite cette liste en réalisant au passage l'affectation souhaitée.

- Figure 7 : les fonctions en LISP -

NOTES

note 1: nous ne parlons pas ici des "paires pointées", un peu plus générales mais d'usage rare (cf. [1]).

note 2: il y a des exceptions suivant le type de la fonction F (cf. figure 7).

note 3: encore que le plus grand système au monde de calcul formel, que des scientifiques utilisent couramment, MACSYMA, pour de très gros calculs (intégration, tenseurs, etc...) fut écrit en...LISP au M.I.T. (cf. [4])

note 4: robotique, jeux, écriture d'interprètes de langages applicatifs (la version actuelle du futur LOGO des Lycées est interprétée par Le_Lisp), étude du langage naturel, des systèmes experts, et de tous problèmes d'I.A. (cf. [1] et [2]), dont la représentation est arborescente.

Bibliographie:

- [1] : "LISP, un langage pour l'Intelligence Artificielle", par Kiremitdjian et Roy, à paraître chez Nathan.
- [2] : "LISP", by Winston and Horn (Addison-Wesley)
- [3] : "LISP 1.5 Programmer's Manual" (MIT Press)
- [4] : "Ordinateurs et calcul symbolique", in Education et Informatique no. 14
- [5] : "LISP: mode d'emploi"
par C. Queinnec, Eyrolles éd.

I.2

Introduction à LISP
(2ième partie)

Le paragraphe I.1 vous a donné un premier aperçu sur le style de programmation en Lisp, utilisant comme structure de donnée "l'expression" (atome ou liste), et comme structure de contrôle "l'application fonctionnelle". Rappelons-en rapidement le principe : une liste est donnée à évaluer à l'interprète LISP, qui la perçoit comme application d'une fonction (le CAR - ou premier élément - de la liste) à une liste d'arguments (le CDR - ou reste - de la liste), chacun de ses arguments étant à leur tour préalablement évalués; ceci est une définition "circulaire" (i.e. récursive...) de l'interprète LISP. Par exemple:

```
? (+ (* 2 3) (car '(5 10)))
= 11
```

car il s'agit de l'application de la fonction + (le CAR de la liste formant l'appel) à la liste (6 5) formée de l'évaluation de la liste (* 2 3) et de la liste (car '(5 10)). Le symbole ' (quote) est là pour empêcher l'évaluation de la liste (5 6) et le message d'erreur "fonction indéfinie 5".

Puisque vous revoilà dans le bain, nous allons maintenant tenter de programmer en LISP et de voir l'intérêt réel de ce langage symbolique à-travers certaines réalisations. Nous tenterons de garantir au maximum la "portabilité" des fonctions LISP qui vont suivre. Plusieurs versions Lisp circulant sur micro-ordinateurs (en particulier sur Apple II et IBM-PC), nous nous référons, si besoin est, au dialecte Le_Lisp 80, qui va être la version officielle de l'Education Nationale pour ses machines à base de Z-80 (et incidemment pour nous sur l'Apple II sous CP/M). Voir la figure 6 pour de plus amples détails sur certaines primitives ici utilisées. Mais, moins de discours, et récurrons un peu...

Atout Lisp !

Commençons par détailler le tri d'un jeu de 52 cartes, sur le modèle du tri donné dans l'article précédent. Une "main" sera représentée par une liste d'un nombre quelconqué de cartes, une "carte" étant elle-même une liste à deux éléments (figure valeur). Le 8 de Trèfle et le Valet de Coeur s'écriront donc par exemple (T 8) et (C V), un exemple de main étant:

```
MAIN = ((K D)(C V)(K 7)(P 9)(T A)(C R)(T D)(K 4)(C 6)(T R)(T 2))
```

Il s'agit donc de trier cette main en ordonnant les familles par T(rèfle, K(arreau, C(oeur, P(ique, chaque famille étant elle-même ordonnée par valeurs croissantes 2, 3, ... , R(oi, A(s.

La figure 1 montre les fonctions correspondantes de ce classique "tri par insertion". La fonction +PETIT retourne T si la carte c1 doit être classée avant la carte c2; on teste la famille d'abord, et la valeur seulement en cas de cartes d'une même famille. La comparaison entre lettres se fait en regardant leurs rangs dans la liste J = (T K C P V D R A) qui est une variable globale du programme. Il ne reste qu'à essayer:

```
? (tri MAIN)
= ((T 2)(T D)(T R)(T A)(K 4)(K 7)(K D)(C 6)(C V)(C R)(P 9))
```

Il existe en fait une autre manière que le recours à la fonction RANG, c'est l'utilisation des "p-listes" ou listes de propriétés. Une telle liste est liée à chaque atome en mémoire et se compose de couples propriété-valeur. Pour conserver le fait que + est un opérateur numérique à deux arguments, on écrira:

```

*****
? (setq J '(T K C P V D R A))
= (T K C P V D R A)
?
? (de tri (L))
?   (cond
?     ((null L) ()) ; la main vide est toute triée.
?     (t (insertion (car L) ; sinon insérer la 1ère carte...
?           (tri (cdr L))))) ; ...dans le reste déjà trié.
= TRI
?
? (de insertion (c L))
?   ; retourne une copie de la liste L avec insertion de la carte c à sa place
?   (cond
?     ((null L) [c])
?     ((+petit c (car L))
?       (cons c L)) ; en tête...
?     (t (cons (car L) (insertion c (cdr L))))) ; ...ou dans le reste.
= INSERTION
?
? (de +petit (c1 c2))
?   ; la carte c1 est-elle classée avant c2 ?
?   (cond
?     ((neq (car c1) (car c2)) ; pas de la même famille.
?       (< (rang (car c1) J) (rang (car c2) J)))
?     ((symbolp (cadr c2)) ; si la 2ème carte est une figure:
?       (or (numberp (cadr c1)) ; ou bien la 1ère est un nombre,
?         (< (rang (cadr c1) J) (rang (cadr c2) J)))) ; ou une fig. +petite
?     (t ; la 2ème carte est donc un nombre:
?       (and (numberp (cadr c1)) ; la 1ère doit alors en être un aussi,
?         (< (cadr c1) (cadr c2))))) ; et même un plus petit !
= +PETIT
?
? (de rang (f J))
?   ; retourne le rang de la lettre f dans la liste ordonnée J
?   (cond
?     ((eq f (car J)) 1)
?     (t (+ 1 (rang f (cdr J)))))
= RANG

```

fig. 1 - tri d'une main d'un jeu de 52 cartes -

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

```

fig. 2 - le triangle de Pascal -

```
? (putprop '+ 'numerique 'type)
= +
? (putprop '+ 2 'arite)
= +
? (plist '+) ; pour consultation de la p-liste associée à l'atome +
= (arite 2 type numérique)
? (getprop '+ 'type) ; pour obtenir la valeur attachée à une propriété
= numérique
```

Pour le programme qui nous occupe, nous stockerons le rang d'une lettre (figure ou valeur) sur la p-liste de l'atome RANG:

```
? (mapc (lambda (x y) (putprop 'rang x y))
?      '(1 2 3 4 5 6 7 8)
?      '(T K C P U D R A))
= NIL ; valeur de retour de la fonction MAPC (cf. figure 6)
? (plist 'rang)
= (A 8 R 7 D 6 U 5 P 4 C 3 K 2 T 1)
? (getprop 'rang 'D) ; au lieu de (rang 'D J)
= 6
```

Les p-listes sont des outils d'une grande souplesse pour conserver et consulter des connaissances (cf. 1) voire des actions à envisager (en déposant des lambda-expressions dans une p-liste).

Du triangle de Pascal à la synthèse de programmes

Le triangle numérique de Pascal (figure 2), qui fait les beaux jours des classes de Maths, est engendré de la manière suivante: chaque terme x est la somme du terme y situé au-dessus et du terme z à gauche de y (0 par défaut):



La fonction principale pour le fabriquer est simple:

```
? (de triangle (n L) ; pour engendrer n lignes à-partir de L
?   (if (= n 0)      ; si...(cf. figure 6)
?     'content?      ; alors fini
?     (print L)      ; sinon écrire L...
?     (triangle (- n 1) (suivante L)))) ; ...et continuer.
= TRIANGLE
```

Pour l'engendrer, il suffit en effet de savoir calculer une ligne, connaissant la précédente; par exemple (1 3 3 1) → (1 4 6 4 1). Chaque élément de la liste est remplacé par sa somme avec le précédent, avec une exception pour le premier. La fonction MAPLIST (fig. 6) est l'une des "fonctions de balayage" ad hoc pour ce genre de situation:

```
? (de suivante (L)
?   (cons 1
?     (maplist
?       (lambda (x) (+ (car x) (cadr x)))
?       L)))
= SUIVANTE
? (suivante '(1 3 3 1))
= (1 4 6 4 1)
```

Remarquez l'utilisation d'une "fonction anonyme" notée "lambda". La fonction MAPLIST applique cette "lambda" à la liste L puis à tous ses CDR, produisant le résultat souhaité:

```
? (triangle 4 '(1))
(1)           ; Ceci...
(1 1)         ; ...est...
(1 2 1)       ; ...l'effet...
(1 3 3 1)     ; ...de...
(1 4 6 4 1)   ; ...bord.
= CONTENT?
```

La fonction "triangle" est récursive (elle s'appelle elle-même), mais d'une manière bien "itérative" : l'appel récursif se fait en dernier, et il n'est pas difficile de vérifier qu'il s'agit en fait d'une classique boucle "tant que" cachée. D'ailleurs les "bons" interprètes LISP (dont le notre, bien sur...) transformeront d'eux-mêmes un tel appel récursif, dit "terminal", en itération.

Un autre exemple bien connu de fonction récursive, la factorielle, comprend pourtant un appel récursif non terminal:

```
? (de fac (n)
?   (if (= n 0)
?     1
?     (* n (fac (- n 1)))))
= fac
```

En effet, après l'appel intérieur à "fac", il reste une multiplication à faire et la situation est à-priori un peu moins claire. Il existe néanmoins une technique classique pour obtenir à-partir de cette version une définition telle que l'appel récursif y soit terminal; sans entrer dans les détails de cette "transformation de programme", tout-à-fait automatique à la main, disons que l'idée est d'utiliser une "variable-tampon" qui va contenir le résultat intermédiaire:

```
? (de fac (n)
?   (facbis 1 n))
= FAC
? (de facbis (m n)
?   (if (= n 0)
?     m
?     (facbis (* m n) (- n 1)))) ; avec appel récursif terminal.
= FACBIS
```

Remarquez ce phénomène de "vases communicants" de n vers m. L'idée d'automatiser ce genre de transformations est l'un des grands sujets de recherche actuels en LISP: la "synthèse de programme", flirtant avec l'Intelligence Artificielle, permettrait de simplifier considérablement le travail (et la rentabilité...) du programmeur, qui n'aurait alors qu'à fournir une "spécification" de ce que le programme souhaité est censé faire. Rassurez-vous, il reste encore beaucoup à accomplir...

Le baguenaudier

La récursivité peut simplifier à l'extrême des problèmes de programmation qui seraient de haute volée sans son utilisation. L'exemple des Tours de Hanoi est bien connu, en voici donc un autre, moins populaire: le baguenaudier. Il s'agit du jeu consistant en n cases, et n pions ayant chacun une face noire et une

Le principe du baguenaudier est totalement récursif:

- Pour BLANCHIR les N pions: *****
on commence par blanchir le dernier: *****-
puis on va BLANCHIR les N-1 restants: -----
- Pour blanchir le dernier: *****
on commence par BLANCHIR les (N-2) premiers: -----**
puis on blanchis le dernier: -----*-
et il ne reste qu'à NOIRCIR les (N-2) premiers: *****-
- Pour NOIRCIR K pions à gauche d'un pion donné: -----*
on commence par NOIRCIR les (K-1) premiers: *****-
puis on noircis le K-ième: *****
- Mais, pour noircir le K-ième: *****-
il faut BLANCHIR les (K-2) premiers: -----*-
puis noircir le K-ième: -----**
et enfin NOIRCIR les (K-2) premiers: *****

```

? (de baguenaudier (n)
?   (blanchir n)
?   'let c'est fini...!) ; valeur de retour
= BAGUENAUDIER
?
? (de blanchir (n)
?   (cond
?     ((= n 0) nil)
?     ((= n 1)
?       (print 'Ije blanchis le 1er!))
?     (t (blanchir (- n 2))
?       (print 'Ije mets le pion I n 'I a blanc!))
?       (noircir (- n 2))
?       (blanchir (- n 1)))))) ; et on recommence!
= BLANCHIR
?
? (de noircir (n)
?   (cond
?     ((= n 0) nil)
?     ((= n 1)
?       (print 'I et je noircis le 1er!))
?     (t (blanchir (- n 2))
?       (print 'I et je mets le pion I n 'I a noir!))
?       (noircir (- n 2))))))
= NOIRCIR

```

fig. 3

- Le Baguenaudier -

face blanche. Initialement, chaque case contient un pion dont la face noire est visible. Il s'agit de mettre tous les pions à blanc, sans les changer de case, sachant qu'on peut changer la couleur d'un pion si :

- ou bien le pion est celui de la case numéro 1;
- ou bien il y a un pion noir immédiatement à sa gauche, et c'est le seul pion noir à sa gauche.

La figure 3 montre l'algorithme et les fonctions LISP correspondantes; les fonctions BLANCHIR et NOIRCIR sont "corécurives", en ce sens que chacune appelle l'autre !

Essayons avec 4 pions:

```

? (baguenaudier 4)          **** N=4
je mets le pion 2 à blanc   *---
je blanchis le 1er          ---*
je mets le pion 4 à blanc   ---*
et je noircis le 1er        *---
et je mets le pion 2 à noir ***- N=3
je blanchis le 1er          ---*
je mets le pion 3 à blanc   ---*
et je noircis le 1er        *--- N=2
je mets le pion 2 à blanc   *--- N=1
je blanchis le 1er          ---- N=0
= ET C'EST FINI...

```

Branchons Lisp sur une pile!

Nous nous proposons de "compiler" une expression arithmétique en notation préfixée (style LISP) en une suite d'opérations exécutables par une "machine à pile", un peu comme les calculettes Hewlett-Packard (dont la pile est constituée par exemple des 4 registres X, Y, Z et T).

Une opération comme (ADD 3 4) va additionner les contenus des registres R3 et R4 (consécutifs), puis mettre le résultat dans R3 (le plus bas), sans toucher à R4.

Une expression préfixée est, ou bien un nombre, ou bien s'écrit récursivement (<op> <exp1> <exp2>), où <op> est un opérateur binaire, comme +, -, *, et où <exp1> et <exp2> sont deux expressions préfixées.

Effectuons "à la H-P" le calcul de (* (+ 1 2) 5). On range 1 dans R1, puis 2 dans R2, on additionne R1 et R2, ce qui donne 3, que l'on range dans R1; on range ensuite 5 dans R2 (inutilisé), et l'on multiplie R1 et R2, avec le résultat dans R1. Le résultat d'un calcul débuté au registre R1 se finit toujours en R1. La figure 4 montre un programme LISP chargé du travail, ainsi que les mouvements de la pile des registres lors d'un calcul.

```

? (compilation '(* (+ 1 2) (! 3 (* 4 5))))
### opérateur inconnu !
= HP

```

```

*****
? (de compilation (exp)
?   (tag erreur                                ; TAG permet les "échappements" (fig. 6)
?     (list (compiler exp 1)))                ; compiler exp à-partir de R1
?   'HP)                                       ; la valeur retournée (un clin d'oeil)
= COMPILATION
?
? (de compiler (exp R)                        ; compiler exp à-partir du registre R
?   (if (numberp exp)
?     [[ 'STO exp R]]      ; ranger le nombre dans le reg. R (cf fig. 6 pour [])
?     (append
?       (compiler (cadr exp) R)                ; compiler les operandes...
?       (compiler (caddr exp) (+ R 1))        ; ...dans les registres R et R+1,
?       [[(code (car exp)) R (+ R 1)]])       ; puis exécuter.
= COMPILER
?
? (de code (op)
?   (selectq op
?     (* 'MUL)
?     (+ 'ADD)
?     (- 'SUB)
?     (t (exit erreur ; et rend la main à COMPILATION après avoir évalué:
?       (print 'l### opérateur inconnu l op))))
= CODE
?
? (de lister (prgm)
?   (when prgm
?     (print (car prgm))
?     (list (cdr prgm))))
= LISTER

```

fig. 4

- Une pile pour LISP -

```
? (compilation '(* (+ 1 2) (- 3 (* 4 5))))
```

```
(STO 1 1)
```

```
(STO 2 2)
```

```
(ADD 1 2)
```

```
(STO 3 2)
```

```
(STO 4 3)
```

```
(STO 5 4)
```

```
(MUL 3 4)
```

```
(SUB 2 3)
```

```
(MUL 1 2)
```

```
= HP
```

```

      ↑
4 1          5 5 5 5
3 1          4 4 20 20 20
2 1      2 2 3 3 3 3 -17 -17
1 1 1 1 3 3 3 3 3 3 -51
----->
R 1 mouvements de la pile

```

Cette fonction "compilation" ne fait, bien sur, que traduire et non calculer. En prenant alors le problème à l'envers, vous pouvez vous entraîner à écrire un "interprete" qui prend comme paramètre la liste retournée par la fonction COMPILATION (le "texte-source"), et qui calcule le résultat; ou bien un "décompilateur" qui prend le meme texte-source et restitue l'expression préfixée de départ...

Le robot FREDDY

Nous allons présenter maintenant un petit robot manipulateur de cubes empilés A, B, C, D, etc...sur une table (cf. figure 5), en suivant une idée de H. Wertz, de l'Université Paris 8.

Le "micro-monde" des cubes est initialisé en fournissant sa description. La fonction INIT-MONDE (figure 5) est chargée de cette besogne, qui pourrait tout aussi bien être interactive.

La fonction principale est POSER. L'ordre donné au clavier se présentera comme appel d'évaluation d'une liste du type (poser D sur A). Les arguments de cette fonction sont donc ici D, SUR et A qui sont des symboles non évalués, d'où l'idée d'écrire la fonction POSER sous forme d'une FEXPR (fonction n'évaluant pas ses arguments, cf. article précédent). Incorporer SUR dans la liste des paramètres est une manière de rédiger l'appel sous forme de langage pseudo-naturel; la variable SUR étant bien entendu "bidon"...

Pour POSER X sur Y, on commence par DEGAGER X et Y des cubes qu'ils peuvent soutenir, on cherche dans la description du monde sur quel cube se trouvait X, i.e. un fait du type (SUR X ?f), ceci grâce à la fonction PRESENT. Puis on raye cette "liaison", tout en rajoutant le nouveau fait (SUR X Y).

Pour DEGAGER un cube X, on cherche quel cube K est PRESENT sur X, et on DEPOSE ce cube K sur la table.

Et pour DEPOSER un cube X sur Y, on... mais ceci, on sait déjà le faire ! La démarche est là encore typique d'un certain style de programmation, récursif et descendant. Il est essentiel qu'un tel langage interactif autorise, lors de la création d'une fonction, l'utilisation d'une autre fonction, non encore définie.

Il reste à étudier la fonction de recherche PRESENT. La recherche en question se fait en avançant dans la liste des faits (il s'agit de MONDE dans la fig. 5). Regardez de près la fonction PRESENT: elle utilise une fonction RESSEMBLE pour comparer un fait donné - son second argument, comme (SUR B A) - à un squelette de fait, comme (SUR ?f A). Elle va chercher s'il existe une valeur de ?f telle que les deux listes puissent être amenées en coïncidence. Sur l'exemple des deux faits précédents, la réponse est oui, à condition que ?f prenne B comme nouvelle valeur (d'où l'usage de SET dans RESSEMBLE qui fait cette affectation).

```

*****
? (de init_monde () ; dans le desordre...
?   (setq monde '((sur B A)                                C
?   (sur D TABLE)
?   (sur C B)                                                B E
?   (sur A TABLE)
?   (sur E D)))                                              A D
= INIT_MONDE ----- TABLE
? (df poser (X sur Y)
?   (if (eq X Y)
?       'ltrop difficile...!
?       (degager X)
?       (degager Y)
?       (enlever ['sur X '?f]) ; prendre X...
?       (ajouter ['sur X Y])) ; ...et le poser sur Y.
?   monde) ; la valeur de retour.
= POSER
? (de degager (X)
?   (if (present ['sur '?f X] monde) ; si ?f est sur X...
?       (deposer ?f))) ; ...mettre ?f sur la table.
= DEGAGER
? (de depoter (X) ; sur la table !
?   (degager X)
?   (enlever ['sur X '?f] monde)
?   (ajouter ['sur X 'TABLE]))
= DEPOSER
? (de ajouter (fait)
?   (print 'l et je le pose sur l (caddr fait))
?   (setq monde (cons fait monde))) ; ajout en tete.
= AJOUTER
? (de enlever (F) ; F est un squelette de fait...
?   (print 'l je prends le cube l (cadr F))
?   (setq monde (mise_a_jour monde)))
= ENLEVER
? (de mise_a_jour (M) ; F est une variable globale pour 'mise_a_jour'
?   (cond
?       ((null M) nil) ; rien a enlever.
?       ((equal F (car M)) (cdr M)) ; on regarde le premier...
?       (t (cons (car M) (mise_a_jour (cdr M))))) ; ...sinon le reste.
= MISE_A_JOUR
? (de present (F M)
?   ; le (squelette de) fait F est-il element du monde M ? Si oui, c'est le fait
?   ; F (complet) qui est retourne.
?   (cond
?       ((null M) nil)
?       ((ressemble fait (car M)) (car M))
?       (t (present fait (cdr M)))))
= PRESENT
? (de ressemble (sqt fait) ; sqt = squelette de fait.
?   ; pour essayer de faire coïncider, avec (sur A B) par exemple, une liste de
?   ; l'un des 3 types suivants: (sur ?x y), (sur ?x ?y), (sur x ?y):
?   (cond
?       ((neq (car (explode (cadr sqt))) #/?)) ; (sur x ?y)
?           (and (eq (cadr sqt) (cadr fait))
?               (set (caddr sqt) (caddr fait))))
?       ((eq (car (explode (caddr sqt))) #/?)) ; (sur ?x ?y)
?           (set (cadr sqt) (cadr fait)
?               (caddr sqt) (caddr fait)))
?       (t ; (sur ?x y)
?           (and (eq (caddr sqt) (caddr fait))
?               (set (cadr sqt) (cadr fait)))))
= RESSEMBLE

```

***** - figure 5 - *****

Cette technique dite "de filtrage" est de plus en plus utilisée dans les gestions de bases de connaissances, avec des fonctions de ressemblance (généralement notées MATCH par allusion au "pattern-matching", ou reconnaissance des formes). On les utilise aussi pour filtrer des phrases en langage naturel et n'en retenir que les mots-clés: LISP est parfaitement adapté à ce genre de traitement, tout-à-fait symbolique.

```
? (init_monde)
= ((SUR B A) (SUR D TABLE) (SUR C B) (SUR A TABLE) (SUR E D))
? (poser A sur D)
je prends le cube C
  et je le pose sur TABLE
je prends le cube B
  et je le pose sur TABLE
je prends le cube E
  et je le pose sur TABLE
je prends le cube A
  et je le pose sur D
= ((SUR A D) (SUR E TABLE) (SUR B TABLE) (SUR C TABLE) (SUR D TABLE))
```

Ce programme pourrait être poursuivi dans de nombreuses directions. Par exemple, il serait intéressant - et assez facile - d'avoir une fonction STRUCTURE qui offrirait une autre représentation du monde sous forme de liste de piles. Ainsi, avec MONDE initialisé dans la figure 5 :

```
? (structure monde)
= ((C B A) (E D))
```

et il va alors de soi que l'étape suivante serait de visualiser en temps réel les changements du monde pour poser un cube sur un autre (même sur l'écran-texte si l'on ne dispose pas du graphique), et enfin - suprême délice - de pouvoir écrire une fonction CHANGER qui prend comme paramètres deux mondes M1 et M2, et telle que l'appel (changer M1 M2) nous montre la succession des dépilements et empilements nécessaires au transit d'un monde vers l'autre. Mais vous êtes alors en plein dans les problèmes ouverts de l'Intelligence Artificielle...

Le système Le_Lisp 80 comprenant plus de 300 fonctions prédéfinies (!), vous ne trouverez ici que celles ayant servies à l'élaboration des programmes de ces 2 articles consacrés à LISP. Les réels ne sont pas implémentés. Nous noterons -> le résultat d'une évaluation, L une liste, s un symbole, et E une expression. De plus, nous supposerons dans les exemples que le symbole X a pour valeur (3 4).

Les "pseudo-chaines de caractères" (en fait des symboles) sont encadrées par le caractère | (ascii 124): |Hello !..| -> Hello !..

Les fonctions arithmétiques + - * portent sur un nombre quelconque d'arguments: (+ 1 2 3) -> 6. La division entière est notée /.

Les fonctions de base CAR et CDR de cette version vérifient (CAR NIL) -> NIL et (CDR NIL) -> NIL, bien impur et bien pratique...

(CONS E L) construit une liste dont le CAR est E et le CDR est L.

(CONS '(1 2) '(3 4 5)) -> ((1 2) 3 4 5)

(APPEND L1 L2) construit la nouvelle liste obtenue par concaténation des listes L1 et L2. (APPEND '(1 2) '(3 4 5)) -> (1 2 3 4 5)

(LIST E1...EN) retourne la liste des évaluations de E1...EN; se note plus couramment [E1...EN]. [2 'sur X] -> (2 sur (3 4))

(READ) retourne la 1ère expression lue au clavier (ou sur disque)

(PRINT E1...EN) imprime sur la console (ou l'imprimante) les expressions E1...EN puis envoie un retour-chariot. Retourne la valeur de EN. Une variante, PRIN, ne va pas à la ligne.

(SET e1 E1...eN EN) affecte à la valeur (un symbole) de chaque ei la valeur de Ei et retourne la valeur de EN. (SET (CAR 'A B) 3) -> 3, avec A=3.

(SETQ s1 E1...sN EN) agit de meme que SET, mais n'évalue pas les symboles si. Si X=(3 4): (SETQ A X) -> (3 4), avec A=(3 4)

(NUMBERP E) retourne NIL lorsque E n'est pas un nombre, et E lui-meme sinon.

(ATOM E) retourne T lorsque E est un atome, et NIL sinon.

(ATOM NIL) -> T (ATOM 'HELLO) -> T (ATOM -10) -> T

(NULL E) retourne T lorsque E vaut NIL, et NIL sinon.

(NULL (ATOM '(1 2))) -> T

(EQ s1 s2) retourne T lorsque les symboles s1 et s2 sont égaux, et NIL sinon.

La fonction EQUAL sert à comparer l'égalité des contenus de deux listes, et (NEQ s1 s2) équivaut à (NULL (EQ s1 s2)).

(COND C1...CN) est fondamentale en Lisp. Chaque Ci est une "clause", du type (F1 E1...EM) dans laquelle F1 est perçu comme un "test"; COND va chercher le premier Fi qui ne vaut pas NIL, évalue alors les expressions E1...EM, et retourne la valeur de EM.

(COND ((EQ 1 2) 'BOF) ((EQ 2 2) 'OUI) ((EQ 2 3) 'AIE)) -> OUI

Trois variantes:

- (IF E E1 E2...EN) équivaut à (COND (E E1) (T E2...EN));

- (IFN E E1 E2...EN) équivaut à (IF (NULL E) E1 E2...EN);

- (WHEN E E1...EN) équivaut à (IFN E NIL E1...EN).

(AND E1...EN) évalue les expressions E1...EN et retourne NIL dès que l'une d'elles évalue à NIL. Sinon, AND retourne la valeur de EN.

(OR E1...EN) évalue les expressions E1...EN jusqu'à ce que l'une n'évalue pas à NIL, et retourne alors sa valeur; sinon retourne NIL.

(WHILE E E1...EN) tant que E n'évalue pas à NIL, WHILE évalue les expressions E1...EN; dès que E vaut NIL, WHILE retourne NIL.

(MAPLIST f L) applique la fonction f à la liste L, puis à tous ses CDR, et retourne la liste des valeurs de ces applications.

(MAPLIST (lambda (L) (+ (car L) (cadr L))) '(1 3 3 1)) → (4 6 4 1)
en remarquant que (c'est une facilité de cette implémentation), le dernier CDR étant (1), son CADR est NIL, qui vaut 0 pour la fonction +.

(MAPC f L) applique la fonction f successivement à tous les éléments de la liste L, et retourne NIL. MAPCAR retourne, elle, la liste des évaluations.

(MAPC 'prin '(1 2 3)) écrit 123, et → NIL
(MAPCAR 'prin '(1 2 3)) écrit 123, et → (1 2 3)

(EXPLODE s) retourne la liste des codes ASCII du symbole s. La fonction inverse est IMplode.

(EXPLODE 'CAR) → (67 65 82) et (IMplode '(67 65 82)) → CAR

#/ est un "macro-caractère" dont nous ne donnons que le modèle de fonctionnement : appliqué au caractère qui suit, il en retourne le code ASCII.

#/A → 65

(TAG s E1...EN) : s est un symbole non évalué (un nom d'erreur par ex.); les expressions E1...EN sont évaluées. Si aucun appel à la fonction EXIT n'a lieu pendant ces évaluations, TAG retourne la valeur de EN. Si LISP rencontre un appel du type (EXIT s F1...FM), les évaluations en cours sont stoppées, il y a évaluation de F1...FM, et on sort du TAG avec la valeur de FM. Très puissant pour briser proprement le contrôle usuel de l'interpréteur LISP...

(TAG BYE (PRINT 1) (IF T (EXIT BYE 2)) (PRINT 3)) imprime 1, et → 2

PLIST, GETPROP, PUTPROP sont décrites dans le texte. REMPROP sert à retirer une propriété.

D'autres fonctions, très puissantes, ne sont pas décrites ici, telles LET (pour gérer des variables locales) ou FLET (fonctions locales !). Enfin, toutes les fonctions usuelles d'entrées-sorties sont disponibles, ainsi qu'un éditeur pleine page PEPE, dérivé de EMACS (le même qu'en LOGO), et des utilitaires de Trace et Pretty-Print. Ce système est destiné à devenir, avec VLISP, l'une des versions les plus utilisées en France.

fig. 6

- Les primitives utilisées -

du Lisp
et des
élèves

I.3.1 PLOT : une fonction LISP basse-résolution sur Apple II.

Deux élèves, ayant une petite pratique de l'assembleur, n'ont pas apprécié, et ce ne sont pas les seuls, l'absence de graphisme en Le_Lisp version APPLE II. Muni d'une bonne dose de courage, ils ont décidé d'essayer d'implémenter une fonction PLOT, qui est à la base d'un tel système.

Première étape: lire en détail le manuel du constructeur de la machine utilisée, ici un Apple II sous CP/M, manuel donnant le mode de lecture et d'affichage de l'écran graphique. Sur Apple II, il y a deux modes graphiques: la "basse résolution" (le point est un pavé assez gros) et la "haute résolution".

Bien entendu, l'objectif est la résolution la plus fine. Hélas, trois fois hélas, ce mode utilise une plage-mémoire (\$2000 à \$3FFF) occupée par l'interpréteur Le_Lisp 80. Une petite remontée aux sources, et un concepteur de cette version (E. Saint-James) me laisse entendre qu'oublier ce projet est plus qu'opportun.

Les élèves décident alors de se rabattre sur la basse résolution, qui a l'avantage de n'être qu'un autre mode d'interprétation de la page-texte. La page-texte étant (et pour cause...) disponible, il ne restait plus qu'à trouver les adresses cruciales nécessaires à l'implémentation du mode graphique, avec une petite translation au passage due à la présence du système d'exploitation CP/M...

L'Apple II ayant l'énorme avantage d'être entièrement transparent et "désossé" au niveau du programme-moniteur, ces adresses se sont vite manifestées et une semaine après, la fonction PLOT était effective.

Vous trouverez ci-dessous le résultat de ce travail, très technique. Les amateurs d'octets n'auront aucun mal à le lire; les autres voudront sans doute incorporer ces nouvelles fonctions dans un fichier spécial et se borner à les utiliser. LISP permet aussi de contenter les amateurs de PEEK et autres POKE...

A-propos d'utilisation, un "casse-brique" a été réalisé: muni d'une raquette mobile pilotée au clavier, il s'agit d'abattre un mur de briques grace à une balle naviguant de la raquette au mur. Il est plus performant que son analogue BASIC présent dans la disquette de démonstration Apple II, et possède meme un "générateur de mur" ainsi que la couleur (optionnelle).

; Le_Lisp 80 et basse-résolution graphique sur Apple II
; Ecole Alsacienne, année scolaire 1983-84.

```
(setq $color$ 0)
```

```
(de couleur (n)
  (and (< n 0) (< n 16) (setq $color$ n)))
```

```
(de texte (eff)
  (if eff (tycls))
  (memoryb '(<#$E0 #51))
  t)
```

```
(de graph (eff)
  (mapc 'memoryb '(<(<#$E0 #50)
                    (<#$E0 #56)
                    (<#$E0 #54)
                    (<#$E0 #52)))
  (if eff (effgr)))
```

```
(de effgr ()
  (let ((b 0))
    (unless (= b 1024)
      (memoryb [(+ #F4 (/ b 256)) (\ b 256)]
                (* $color$ 17))
      (self (1+ b)))))
```

```
(de plot (x y) ; LA fonction...
  (let ((adr (bytegr x (/ y 2)))
        (ym2 (= (logand y 1) 0)))
    (let ((fut (logand (memoryb adr) (if ym2 #F0 #0F))))
      (memoryb adr (logor fut (if ym2 $color$ (* $color$ 16)))))))
```

```
(de bytegr (x y)
  [(logor #F4 (logand #03 (/ y 2)))
   (+ x (* (logand y #18) 5) (* (logand y #01) #80))])
```

; et une petite demo (en couleurs):

```
(de demo ()
  (couleur 0) (graph t) (couleur 15)
  (let ((n 0))
    (when (< n 40)
      (plot n n) (plot (- 39 n) n) (plot 0 n)
      (plot 39 (- 39 n)) (plot n 39) (plot (- 39 n) 0)
      (if (= (\ n 3) 0)
        (couleur (1+ (\ (1+ $color$) 15))))
      (self (1+ n)))
    (couleur 15)
    (lineplot 0 0 39 0) (lineplot 0 39 39 39) (lineplot 0 0 0 39)
    (lineplot 39 0 39 39) (lineplot 0 0 39 39)
    (let ((n 0))
      (when (< n 40)
        (plot (- 39 n) n)
        (self (1+ n)))
      (tyi)
      (texte t)
      'lowres)
```

; la fonction (LINEPLOT x1 y1 x2 y2) est laissée à votre sagacité, mise à
; l'épreuve en ce qui concerne les obliques...

1.3.2 Utilisation des fonctions du BDOS de CP/M sous Le_Lisp 80

Notre version Le_Lisp 80 sur Apple II n'offrait pas de primitives permettant de manipuler la disquette autrement que par entrées-sorties de lecture-écriture. Il est assez décevant de ne pouvoir accéder au "directory" d'une disquette, ou de ne pas être en mesure d'effacer un fichier devenu inutile, en renommer un autre, etc.

Or, un agrément de CP/M, qui en compte pourtant si peu, consiste en un certain nombre de fonctions (une trentaine) codées en langage-machine, et permettant toutes les opérations à condition de les appeler par une petite routine très simple écrite en assembleur (nul besoin de savoir compter sur 16 doigts - i.e. programmer couramment en assembleur - pour les utiliser).

Il se trouve que Le_Lisp 80 dispose d'une fonction EXECUTE qui permet, comme son nom l'indique, d'exécuter une liste d'instructions en code machine, en retournant une valeur dans le registre HL.

Nous n'allons pas détailler le système d'exploitation CP/M, mais disons simplement que les fonctions dont nous avons parlé plus haut, et qui en forment le coeur, sont regroupées dans le BDOS (Basic Disk Operating System), qui est indépendant de la machine utilisée.

Ce petit chapitre, tout comme le précédent, est utilitaire, et renseigne peu sur Lisp lui-même. Son seul but est d'aider un peu les collègues travaillant sous CP/M.

L'appel au système BDOS se fait par le point d'entrée 0005H. Un CALL à cette adresse va indiquer à CP/M que l'on veut utiliser une fonction du BDOS. Pour lui indiquer laquelle, on dépose son numéro (cf. manuel du système) dans l'accumulateur C par un MVI. S'il faut passer une information à la fonction concernée, cela se fait en déposant une adresse dans le registre DE. Si la fonction retourne une valeur, celle-ci est stockée dans le registre HL. Enfin, la fonction Lisp EXECUTE retourne la valeur trouvée dans HL.

Exemples:

1) Sélection du lecteur de disques.

Pour changer (ou consulter si d n'est pas fourni) le numéro du lecteur en de disquettes en ligne. On fait appel à aux fonctions 14 "Select Disk" ou 25 "Return Current Disk" du BDOS:

```
(de selectdrive (d)
  (ifn d
    (ascii
      (+ 65
        (execute
          [#0E 25
           #0D 05 00
           #09
          ])))
    (execute
      [#0E (- (ascii d) #/A)
       #0E 14
       #0D 05 00
      ]))
  d))
; si d n'est pas fourni
; retourner le caractère de code ascii:
; conversion chiffre -> lettre
; MVI C,25
; CALL BDOS
; RETURN (vers Lisp)
; LD E,numero du lecteur (manuel CP/M)
; MVI C,14
; RET
; valeur de retour de SELECTDRIVE
```

2) Changer de disquettes.

Opération toujours fort risquée sous CP/M, qui est très rigide, et apprécie fort les messages du type "BDOS ERROR" catastrophiques pour l'utilisateur. Avant de changer une disquette, il faut appeler la fonction 13 du BDOS, qui se nomme "Reset System Disk":

```
(de newdisk ()
  (let ((d (selectdrive)))      ; le numéro du lecteur en ligne
    (execute
      [#0E 13                  ; MVI C,13
       #0CD 05 00              ; CALL BDOS
       #0C9                    ; RET
      ])
    (selectdrive d)))           ; retour au lecteur initial
```

3) Utilitaires-mémoire

Deux ou trois fonctions pour la spéléologie dans la zone-mémoire comprise entre \$bcode\$ (beginning of user code zone) et \$ecode\$ (end of...). Cette zone-mémoire n'est pas utilisée par Lisp et donc laissée à l'utilisateur pour y faire des choses pas très pures... En l'occurrence, nous nous en servons pour y déposer des informations comme un nom de fichier, ou le FCB d'entrée (File Control Block = bloc de contrôle d'un fichier: c'est un bloc d'une trentaine d'octets donnant à CP/M toutes les informations sur l'organisation d'un fichier-disque).

```
(de getmemory (a n)
  ; retourne le symbole composé des caractères dont les codes ascii sont
  ; dans les octets a,...,a+n
  (implode
    (append
      (cons #/l                      ; l pour conserver les "blancs" sous Lisp
        (let ((p 0))
          (when (< p n)
            (cons (memoryb (+ a p))
                  (self (1+ p))))))
      [#/l])))
```

```
(de putmemory (a s)
  ; met les caractères du symbole s dans les octets a...
  (let ((a a) (L (explode s)))
    (when L (memoryb a (car L))
      (self (1+ a) (cdr L)))))
```

4) Le "directory" d'une disquette.

Ce qui nous intéressait le plus ! Ne sera décrite que la fonction DIRLIST, qui retourne la liste des noms de fichiers présents sur la disquette. Il vous faudra écrire une fonction DIR qui présente mieux (en colonnes) cette liste.

```
(de dirlist (d)
  (newdisk)
  (if d (selectdrive d))
  (let ((a (resetdir)))          ; le 1er fichier du directory
    (if (< a 255)                 ; tant qu'il y a des fichiers...
      (cons
        (getmemory (+ #081 (* a 32)) 11) ; le DMA
        (self (onedir))))))
```

Le DMA (Direct Memory Access) est un fonctionnaire de CP/M spécialisé dans les transferts d'octets entre le disque et la mémoire centrale. Sur l'Apple II, ce DMA commence à l'adresse 80H.

La fonction précédente commence par appeler RESETDIR, qui va initialiser le FCB d'entrée en le remplissant de ? pour spécifier un fichier quelconque (nous n'entrons pas plus dans les détails, il est indispensable de lire le manuel CP/M), puis fait appel à la fonction 17 "Search For First" du BDOS.

La fonction ONEDIR se contente de lire le fichier suivant dans le directory.

```
(de resetdir ()
; initialise le FCB en #$5C et place le 1er fichier dans le DMA
(let ((mem #$5D))
  (while (< mem #$68)
    (memoryb mem #/?))
    (incr mem)))
(memoryb #$5C 0) ; drive courant
(execute
  [#$0E 17      ; MVI C,17
  #$11 #$5C 00  ; LXI D,FCB avec FCB = #$5C
  #$CD 05 00    ; CALL BDOS
  #$C9          ; RET
  ]))

(de onedir ()
; charge le nom du fichier suivant dans le DMA et retourne 255 s'il n'y en
; a pas
(execute
  [#$0E 18      ; MVI C,18
  #$11 #$5C 00  ; LXI D,FCB
  #$CD 05 00    ; CALL BDOS
  #$C9          ; RET
  ]))
```

CONCLUSIONS

Tout d'abord, l'idée est (encore une fois !) venue des élèves, qui n'ont pas goûté l'absence de système d'exploitation interne à Le_Lisp 80. Mais ce qui est plus remarquable, c'est que la réalisation des fonctions précédentes est le fait d'un élève, disposé à passer un week-end à lire la documentation de CP/M.

Ce que j'ai donc fait aussi à sa suite... L'informatique en milieu scolaire sera une discipline dans laquelle l'enseignant doit accepter de se "laisser déborder" par son auditoire: la relation d'apprentissage doit aller dans les deux sens, et ce d'autant plus que bon nombre d'élèves ont un "micro" chez eux et sont tentés d'investir un temps parfois démesuré dans cette matière. Si seulement cela était aussi vrai des Mathématiques... et du reste.

Toujours est-il, pour en revenir à ce point, que le Lycée et l'Informatique (la programmation, pas le "presse-bouton"...) peuvent faire (très) bon ménage à condition de garantir une grande souplesse chez les enseignants qui en seront chargés. En attendant de définir des programmes rigides (je souhaite bien du plaisir à ceux qui en seront chargés), il faut tâcher de conserver un état d'esprit qui soit au maximum "expérimental". L'informatique scolaire aura-t-elle la chance de pouvoir figurer tout ce que n'est pas l'institution scolaire : une activité de laboratoire ?

I.3.3

Techniques de filtrage

Il a été question dans le 2ième article de cette technique dite "de filtrage" utilisée pour cribler la base de données du robot FREDDY.

L'idée est de comparer deux expressions Lisp "sans entrer trop dans les détails", i.e. en tâchant de comparer deux "formes" (en anglais "pattern", d'où le vocable "pattern-matching", ou reconnaissance des formes).

Etant donnés:

- une expression Lisp, en général une liste, comme:

```
LISTE = (X 1 Y LISP OK (5 6 (OK)))
```

- un squelette d'expression Lisp (un "pattern"), comme:

```
PAT = ((* X) LISP (? Y) (** Z) (< (? Y)))
```

Le but est de trouver un jeu de substitutions pour les variables X, Y, Z, U qui rendent les deux listes identiques (EQUAL). Si l'on y parvient, le filtrage a réussi, sinon il échoue.

La signification des caractères de filtrage * et ? est la suivante:

- a) (* X) filtre n'importe quel "contenu de liste":

```
? (filtre '(1 (* X) 5 6) '(1 2 (3) 4 5 6))
= T
? X ; filtrage réussi, on vérifie:
= (2 (3) 4) ; OK, mais:
? (filtre '(1 (* X) 5 6) '(1 2 (3) 4 5)) ; impossible.
= NIL
```

- b) (? X) filtre n'importe quelle expression Lisp (mais une seule, atome ou liste):

```
? (filtre '(encore raté (? X) sabords) '(encore raté mille sabords))
= T
? X
= MILLE
? (filtre '(le (? Y) bleu) '(le lotus vert))
= NIL
```

- c) (** X) est remplacé par le contenu de liste de la variable X (attention, pas par la valeur de X ; on ne met pas les parenthèses les plus extérieures):

```
? (setq L '(base lunaire))
= (BASE LUNAIRE)
? (filtre '(ici la (** L) parlez) '(ici la base lunaire parlez))
= T
```

- d) (< ? X) est remplacé par la valeur du symbole X:

```
? (filtre '(non (? X) de (< ? X)) '(non rien de rien))
= T
? X
= RIEN
```

Après le (? X), le symbole X a reçu une valeur, que l'on peut tester !

Deux remarques donc. Tout d'abord, le filtrage est l'oeuvre d'une seule fonction `FILTRE`, qui renvoie `T` ou `NIL` suivant qu'il y a eu succès ou échec. Mais le plus important est qu'elle ne se contente pas d'envoyer ce message, mais surtout c'est à-travers son effet de bord qu'elle est le plus souvent utilisée: elle modifie en effet des variables de votre programme, comme `X$` ou `C$`, d'où un soin tout particulier à contrôler le partage de ces variables avec le reste de vos fonctions.

C'est pourquoi, dans les exemples qui suivent, ces "variables de filtrage" seront toujours notées avec le suffixe `$`, pour les reconnaître et ne pas s'en servir inconsidérément.

Dans un premier temps, nous allons utiliser cette technique pour écrire des fonctions Lisp normalement récursives. Il se trouve que pour certaines d'entre elles, la récursivité est prise en charge par la fonction `FILTRE` seule. Ceci nous fait plaider pour l'incorporation en série d'un noyau de filtrage dans les interprètes Lisp futurs, pour des raisons d'efficacité.

Le filtrage est en effet très récursif, donc un peu lent, mais le gain en facilité de programmation est tel - notamment pour les débutants et les élèves - que sa présence en langage-machine serait plus que la bienvenue...

D'autre part, son utilisation dans l'apprentissage de la programmation, voire au niveau plus général des processus cognitifs a été souligné, par exemple par H. Wertz dans:

"Education et Informatique", numéro 16, Mai-Juin 1983
(Nathan-Cédic éd.)

article ayant directement inspiré notre robot `FREDDY`.

H. Wertz fait partie de ces chercheurs en Intelligence Artificielle pour lesquels l'aisance dans un bon environnement de programmation est essentielle, et qui prouvent le développement de véritables outils destinés à faciliter la tâche du programmeur. Il est de fait que la France ne se réveille que bien tardivement sur ce plan: combien de programmeurs ne disposent même pas d'un éditeur de texte "pleine page" ?...

La situation est encore pire s'il s'agit de Lycéens que l'on fait travailler avec des éditeurs de lignes !

Mais ceci nous écarte de notre sujet. Avant d'aborder les "définitions filtrées" de fonctions, reprenons l'exemple du début, avec la liste `LISTE` et le squelette `PAT`:

```
? (setq LISTE '(X 1 Y LISP OK (5 6 (OK)))
      PAT '((* X) LISP (? Y) (** Z) ((? Y)))
      Z '(5 6))
= (5 6)
? (if (filtre PAT LISTE)
      (print 'IEffets de bord : I 'IX=I X 'I Y=I Y 'I Z=I Z)
      'IPas fameux...I)
Effets de bord : X=(X 1 Y) Y=OK Z=(5 6)
= (5 6)
```

Attention cependant: `(** L)` ne signifie pas `((* L) ...)`. L'analogue de `((? X)` avec UNE expression ne fonctionne pas avec les contenus de liste, à cause de la stratégie récursive de la fonction `FILTRE` (implémentation à compléter !)

```
? (filtre '((* X) 4 (** X)) '(1 2 3 4 1 2 3))
** EVAL : Variable indéfinie X
```

Les fonctions "filtrées" qui suivent ont fait l'objet de deux séances de TP avec les élèves, qui à aucun moment n'ont eu à regarder en détail comment fonctionnait la fonction `FILTRE` (ce que vous êtes par contre invité à faire...).

1) Inversion d'une liste au premier niveau.

Exemple: `(inverser '(1 (2 3) 4 5)) -> (5 4 (2 3) 1)`

```
-----
(de inverser (L x$ y$ z$)
  (if (or (atom L) (null (cdr L)))
      L
      (filtre '(<(? x$) (* y$) (? z$)) L)
      (append [z$] (self y$) [x$]))))
-----
```

N.B. Supposez que `x$`, `y$` et `z$` n'aient pas été locales à la fonction `INVERSER`. Détailler sur un exemple simple, comme:
`(inverser '(1 2 3 4))`
 le comportement (inattendu ?) de la fonction... Expliquez.

2) Palindromes.

Une liste `L` est-elle palindromique, i.e. `(equal L (inverser L))` ?

Exemple: `(palindrome '(1 (2 3) 4 (2 3) 1)) -> T`

```
-----
(de palindrome (L x$ y$ z$)
  (and (filtre '(<(? x$) (* y$) (<(? x$)) L)
        (self y$)))
-----
```

3) Le langage $X^n Y^n Z^n$.

Reconnaitre si une liste est du type:

$$\underbrace{(x \ x \ \dots \ x)}_{n \text{ fois}} \underbrace{(y \ y \ \dots \ y)}_{n \text{ fois}} \underbrace{(z \ z \ \dots \ z)}_{n \text{ fois}} \quad (\text{avec un certain } n)$$

Exemple: `(XnYnZn '(x x x y y y z z z)) -> T`

```
-----
(de XnYnZn (L x$ y$ z$)
  (if (filtre '(x (* x$) y (* y$) z (* z$)) L)
      (self (append x$ y$ z$))
      (null L)))
-----
```

Le langage formé de tous les mots $X^n Y^n Z^n$ est-il algébrique ?...
 Ecrire le programme PROLOG correspondant.

4) La position d'un élément dans une liste.

Retourner le rang de la 1ère occurrence d'une expression `E` dans une liste `L` et 0 si ce n'est pas un élément de `L`.

Exemple: `(pos 'x '(a b (c d) x y)) -> 4`
`(pos 'x '(a b c (d x) y)) -> 0`

```

-----
(de pos (E L x$ y$)
  (if (filtre '(<(* x$) (<? E) (* y$)) L)
    (1+ (length x$))
    0))
-----

```

5) La fonction d'appartenance (au 1er niveau)

Retourne le N-ième CDR d'une liste L commençant à la 1ère occurrence d'une expression E.

Exemple: (membre 'x '(a b x (c d) e)) -> (x (c d) e)
 (membre 'x '(a b (x y))) -> NIL

```

-----
(de membre (E L x$ y$)
  (if (filtre '(<(* x$) (<? E) (* y$)) L)
    (cons E y$)))
-----

```

6) Sous-liste.

Tester si L1 est ou non une sous-liste de L2 (i.e. si L1 est l'enveloppe d'un contenu de liste de L2).

Exemple: (sous-liste '(3 4) '(1 2 3 4 5)) -> T

```

-----
(de sous-liste (L1 L2 x$ y$)
  (if (null L2)
    (null L1)
    (filtre '(<(* x$) (** L1) (* y$)) L2)))
-----

```

7) Supprimer tous les 0 d'une liste numérique.

Exemple: (saufzero '(1 2 0 3 0 4 5 6 0 7)) -> (1 2 3 4 5 6 7)

```

-----
(de saufzero (L x$ y$)
  (if (filtre '(<(* x$) 0 (* y$)) L)
    (append x$ (self y$))
    L))
-----

```

La généralisation s'impose: et si l'on enlevait tous les nombres pairs ?
 Hélas, le problème est plus compliqué !

8) Recherche des entiers pairs dans une liste de nombres.

De manière filtrée. Ce n'est pas aussi évident... Les caractères de filtrage vont être augmentés de 2, autorisant les prédicats unaires:

```

? (filtre '(<(* X) (& symbolp N) (* Y)) '(400 20 et un))
= T
? N
= ET

```

Là aussi, une affectation est réalisée pour que le filtrage ait lieu, mais avec en plus un test de type correspondant au prédicat SYMBOLP (est-ce un symbole ?)

Nous voulons retourner la liste constituée de tous les entiers pairs d'une liste numérique L.

Exemple: (que-les-pairs '(1 2 3 4 5 6 7 8)) -> (2 4 6 8)

```
-----
(de que-les-pairs (L x$ y$)
  (if (filtre '((* x$) (Q pair? y$) (* z$)) L)
    (cons y$ (self z$))))
-----
```

Pour etre complet, notons que:

```
-----
pair? = (lambda (N) (= 0 (\ N 2)))
-----
```

expression dans laquelle \ est l'opérateur de reste en "division entière" (le MOD de Pascal).

Le code de la fonction FILTRE

Nous renvoyons à l'ouvrage "LISP" publié chez Nathan pour le (long) code de la fonction FILTRE, ainsi que de nombreux compléments.

1.3.4Le robot FREDDY : prolongements

Juste quelques mots. Le "monde des blocs" est un domaine classique d'étude en Intelligence Artificielle, aussi bien pour les problèmes de génération de plans que pour ceux du langage naturel; ce "micro-monde" au sens de Papert est en effet à la fois simple et riche (cf. par exemple "L'Intelligence Artificielle", par D. Waltz, in "L'intelligence de l'Informatique", diffusion Belin, 1984).

Une extension réalisée indépendamment par deux élèves a consisté à visualiser ce micro-monde sur l'écran (texte) afin de voir le "bras" du robot empiler et dépiler les cubes à volonté. Ceci ne pose en effet pas de gros problèmes; il suffit d'utiliser à foison les codes ascii des quatre directions cardinales.

Notez que les "cubes" ne sont cubiques par faiblesse de notre imagination: on peut très bien interpréter les faits simultanés:

(sur B A) (sur C A)

comme indiquant que A a une dimension double de B et C...

I.3.5

Un micro-BASIC sous LISP

LISP est un langage parfait pour la plupart des simulations, ainsi que pour l'obtention rapide d'une maquette à échelle réduite d'un programme pouvant être assez sophistiqué. Il s'agit dans ce paragraphe d'exposer un travail écrit en collaboration avec un élève et consistant à simuler une petite version d'interpréteur d'un langage proche de BASIC: étiquettes à chaque ligne, sauts inconditionnels, sous-programmes avec retour, boucles et affectation. Bref, il y a plus qu'il n'en faut pour faire de la programmation, même si l'on décide de rester bouche cousue sur son esthétique...

Dans la mesure où les élèves sont amenés à programmer de manière impérative, même si l'on souhaite en diminuer la dose, il est bon de leur exposer ce qu'est la gestion d'un GOSUB, comment on empile une adresse de retour, comment on gère une itération, etc. Nous verrons que LISP offre toutes ses facilités pour exprimer ce genre de situations, malgré l'absence de tableaux.

L'élève en question, dont c'était le projet de programmation, avait choisi un style itératif, en gérant la ligne courante et la suivante, ainsi bien entendu qu'une pile pour les GOSUB et les boucles. J'ai préféré montrer un programme récursif, plus lisible bien qu'équivalent (très légèrement plus lent). Cet élève avait choisi de faire une première passe de repérage des boucles avec stockage des informations en p-listes. Contrairement à ses boucles "nommées" (loop 1, loop 2, ...), celles qui sont présentées sont "anonymes" et implémentent la structure LOOP-ENDLOOP-EXIT très agréable, quitte à faire de l'itération. L'ordre EXIT ne permet de franchir qu'un seul niveau de boucle LOOP (la dernière entrée), ce qui force un peu plus à structurer ses programmes (le rajout tardif du déconseillé GOTO n'a demandé qu'une seule ligne!).

Vous connaissez sans doute les principaux mots réservés de BASIC:

LET	: affectation (let x = 2)
PRINT	: écriture (print 'x= x) avec retour à la ligne (print 'x= x ,) sans retour à la ligne
INPUT	: lecture (input x)
GOTO	: saut inconditionnel (goto 150)
IF	: le IF-THEN sans le ELSE, sur le mode du Basic Applesoft
GOSUB	: saut à sous-programme (gosub 150)
RETURN	: retour à l'instruction suivant le dernier GOSUB
END	: termine le programme

Moins connue est la boucle LOOP, non standard en BASIC:

```

10 let x = 1
15 let k = 1
20 loop
30   let k = (* k x)
40   if (<= x 4) then exit
50   let x = (1+ x)
60 endloop
70 print k

```

L'ordre EXIT permet de sortir immédiatement de la (dernière) boucle entrée, i.e. à l'instruction suivant le ENLOOP correspondant.

La boucle LOOP permet de simuler une boucle FOR, l'inverse n'étant pas vrai. Le FOR est une structure privilégiée d'accès aux tableaux et devrait être réservé à cet effet.

Pour éviter la tâche laborieuse d'écrire la grammaire du langage, voici trois exemples de programmes, que nous exécuterons plus tard en mode "trace", et qui devrait éclaircir les idées. Un "programme" est représenté par une liste d'instructions, chaque instruction étant une liste débutant par une étiquette numérique suivie d'un mot réservé, etc.

- les GOSUB emboîtés

```
(setq prog1 '(
  (10 print 'nombre: ,)
  (20 input n)
  (30 if (= n 0) then goto 200)
  (40 gosub 100)
  (50 print 'carre= (* n n))
  (60 end)
  (100 print 'G100)
  (105 gosub 500)
  (110 print 'G110)
  (120 return)
  (200 print 'G200)
  (300 goto 40)
  (500 print 'G500)
  (510 return)))
```

; les spaghettis...

- les LOOP emboîtées

```
(setq prog2 '(
  (10 print 'basic)
  (20 let j = 1)
  (30 loop)
  (40 print 'j= j)
  (41 let k = 2)
  (42 loop)
  (43 print (* j k))
  (44 if (= k 5) then exit) ; EXIT ligne 50
  (45 let k = (1+ k))
  (46 endloop) ; recommencer ligne 43
  (50 if (= j 4) then exit) ; EXIT ligne 80
  (60 let j = (1+ j))
  (70 endloop) ; recommencer ligne 40
  (80 print 'fini))) ; END optionnel
```

Vous noterez d'emblée l'utilisation de LISP comme calculateur. A l'intérieur d'un test, ou dans son résultat, toute la puissance de l'évaluateur LISP peut être sournoisement invoquée. Ecrire un évaluateur d'expressions arithmétiques infixes avec priorités (comme dans un vrai BASIC) relève aussi d'une gestion de pile, mais c'est un travail moins amusant...

Un dernier programme pour faire hurler les amoureux de LISP: l'éternelle fonction REVERSE.

- la fonction REVERSE

```

(setq prog3 '(
  (10  print 'reverse)
  (20  print 'L= ,)
  (30  input L)                ; on peut lire une liste
  (40  gosub 100)              ; au travail!
  (50  print 'lrev(L)=l LR)
  (60  end)
  (100 let LR = ( ))
  (110 loop)
  (120  if (null L) then exit)
  (130  let LR = (cons (next1 L) LR))
  (140 endloop)
  (150 return)))

```

La fonction de lancement du programme est RUN, e.g. (run prog1) ou, si l'on veut un suivi ligne à ligne (trace), (run prog1 t). Le "repérage" du programme consiste à chercher les couples LOOP-ENDLOOP, et à stocker l'adresse du ENDLOOP comme propriété de l'adresse du LOOP. Par exemple, dans PROG1, la p-liste du symbole LOOP serait, après repérage, NIL (pas de boucle), tandis qu'avec PROG2, on obtiendrait (30 70 42 46), et (110 140) avec PROG3. L'adresse 70 du ENDLOOP de la boucle commençant en ~~42~~ dans PROG2 s'obtient donc par:

```
(getprop 'loop 30)
```

Une fonction très utile dans ce programme est ASSCDR, qui retourne (un pointeur sur) la partie du programme débutant à une adresse donnée. Toujours dans PROG2:

```
(asscdr 30 prog2) -> ((30 LOOP) (40 PRINT (QUOTE J=) J) (41 LET ...))
```

La main est passée de RUN à la fonction INTERPRETER, qui va traiter la première ligne du programme, en la passant à TRAITEMENT. Cette dernière va disséquer la ligne (regardez la forme de la liste des paramètres) et aiguiller suivant le mot réservé. Sauf indirection spéciale (goto, gosub, ...), la main est rendue à INTERPRETER avec le reste (cdr) du programme. L'intérêt d'interpréter non une ligne, mais une section finale du programme réside dans le fait qu'une telle section contient elle-même la continuation du calcul.

Si la ligne est une ligne de GOSUB, il faut empiler l'adresse de retour (là joue la continuation du calcul), puis une marque G indiquant la nature de l'empilement, pour récupération lors de la rencontre d'un RETURN. Le même phénomène a lieu, en un peu plus compliqué, pour les boucles; le code n'est pas difficile à lire. En sortant d'un GOSUB ou d'une LOOP, il faut dépiler les informations. Ce n'est pas exactement ce qui a été fait; plus précisément, elles ont été "gommées" et remplacées par le symbole FFFF, pour des raisons didactiques, afin de suivre les empilements et dépilements en mode trace. Vous voudrez sans doute les éliminer plus brutalement: une pile entrée vide devrait toujours ressortir vide, même si elle est en variable locale (initialisation à NIL).

Le IF provoque une "régénération" de la ligne en cours (voyez les traces) et agit donc un peu comme une macro LISP.

Voici maintenant les fonctions proprement dites.


```

(de run (-prog -trace -pile)
  (bind ((prompt 'l l))
    (plist 'loop nil)
    (reperer -prog)
    (if -trace (print 'loops= (plist 'loop)))
    (tag end (interpreter -prog))
    'ISure, but Basic isn't much fun...l))

(de reperer (-prog)
  (and -prog
    (let ((n res . x) (car -prog)))
      (if (eq res 'loop)
        (putprop 'loop (endloop n) n)
        (reperer (cdr -prog)))))

(de endloop (n)
  (tag fini
    (let ((nloop 0) (-prog (cdr (asscdr n -prog)))))
    (ifn -prog
      (erreur 'endloop n)
      (selectq (cadar -prog)
        (loop (self (1+ nloop) (cdr -prog)))
        (endloop (if (= nloop 0)
          (exit fini (caar -prog))
          (self (1- nloop) (cdr -prog)))))
      (t (self nloop (cdr -prog)))))))

(de interpreter (-prgm)
  (ifn -prgm
    (exit end)
    (traitement (car -prgm))))

(de traitement ((-label -res . -L))
  (if -trace (print '-- -label '--))
  (selectq -res
    (print
      (let ((-x (nextl -L)))
        (selectq -x
          (, (flush))
          (nil (terpri))
          (t (prin (eval -x)) (self (nextl -L))))))
      (interpreter (cdr -prgm)))
    (let
      (let (((-symb = -val) -L))
        (if (and (symbolp -symb)
          (eq = '='))
          (set -symb (eval -val))
          (erreur 'let -label)))
        (interpreter (cdr -prgm)))
      (input
        (let (((-symb . x) -L))
          (if (and (symbolp -symb)
            (null x))
            (set -symb (read))
            (erreur 'input -label)))
          (goto (interpreter (asscdr (car -L) -prog)))
          (if
            (let (((-test then . -altv) -L))
              (cond
                ((neq then 'then) (erreur 'if -label))
                ((eval -test)
                  (interpreter (cons (cons -label -altv)
                    (cdr -prgm)))))
                (t (interpreter (cdr -prgm)))))))

```

```

(end      (exit end))
(gosub    (empiler (caadr -prgm))
          (empiler 'G)
          (interpreter (asscdr (car -L) -prog)))
(return   (let ((retour (retour -pile)))
            (ifn retour
              (erreur 'return -label)
              (interpreter (asscdr retour -prog))))))
(loop     (empiler (getprop 'loop -label))
          (empiler (caadr -prgm))
          (empiler 'L)
          (if -trace (print 'loop:pile= -pile))
          (interpreter (cdr -prgm)))
(endloop  (let ((begloop (begloop -pile)))
            (ifn begloop
              (erreur 'endloop -label)
              (if -trace (print 'endloop:pile= -pile))
              (interpreter (asscdr begloop -prog))))))
(exit     (let ((exitloop (exitloop -pile)))
            (ifn exitloop
              (erreur 'exit -label)
              (interpreter (cdr (asscdr exitloop -prog)))))))

```

```

(de exitloop (pile)
  (cond
    ((null pile) nil)
    ((eq (car pile) 'L)
     (rplaca pile 'ffff)
     (rplaca (cdr pile) 'ffff)
     (progl (caddr pile) (rplaca (cddr pile) 'ffff)))
    (t (exitloop (cdr pile)))))

```

```

(de begloop (pile)
  (cond
    ((null pile) nil)
    ((eq (car pile) 'L) (cadr pile))
    (t (begloop (cdr pile)))))

```

```

(de empiler (x)
  (newl -pile x))

```

```

(de retour (pile)
  (cond
    ((null pile) nil)
    ((eq (car pile) 'G) (retour (cdr pile)))
    (t (progl (cadr pile)
              (rplaca pile 'ffff)
              (rplaca (cdr pile) 'ffff)))))

```

```

(de asscdr (n prog)
  (cond
    ((null prog) nil)
    ((= (caar prog) n) prog)
    (t (asscdr n (cdr prog)))))

```

```

(de erreur (f -label)
  (print 'lErreur : l f 'l en ligne l -label))

(df lister (-prog)
  (flet ((println (L)
    (while L (prin (next1 L) 'l l))
    (terpri)))
    (mapc 'println (eval -prog))
    -prog))

```

Les traces

```

? (run prog1)
NOMBRE: 0
G200
G100
G500
G110
CARRE=0
= Sure, but Basic isn't much fun...
?
? (run prog1)
NOMBRE: 3
G100
G500
G110
CARRE=9
= Sure, but Basic isn't much fun...
?
? (run prog1 t) ; trace
LOOPS=NIL
--10--
NOMBRE:--20--
  3
--30--
--40--
--100--
G100
--105--
--500--
G500
--510--
--110--
G110
--120--
--50--
CARRE=9
--60--
= Sure, but Basic isn't much fun...

```

```

? (run prog2)
BASIC
J=1
2
3
4
5
J=2
4
6
8
10
J=3
6
9
12
15
J=4
8
12
16
20
FINI
= Sure, but Basic isn't much fun...

```

```

? (run prog3)
REVERSE
L = (1 2 3)
rev(L) = (3 2 1)
= Sure, but Basic isn't much fun...
?
? (run prog3 t) ; trace
LOOPS=(110 140)
--10--
REVERSE
--20--
L =--30--
(1 2 3)
--40--
--100--
--110--
LOOP:PILE=(L 120 140 G 50)
--120--
--130--
--140--
ENDLOOP:PILE=(L 120 140 G 50)
--120--
--130--
--140--
ENDLOOP:PILE(L 120 140 G 50)
--120--
--130--
--140--
ENDLOOP:PILE=(L 120 140 G 50)
--120--
--120--
--150--
--50--
rev(L) = (3 2 1)
--60--
= Sure, but Basic isn't much fun...

```

Extensions envisageables

Les élèves auront sans doute à écrire l'an prochain un éditeur de lignes permettant d'écrire, de sauver, de charger un programme, ainsi que divers utilitaires agrémentant l'environnement de programmation. Par exemple, cacher complètement LISP afin de "durcir" la simulation.

La boucle WHILE doit permettre de tester si les idées de base ont été assimilées. Ce projet d'interprete BASIC était à l'origine un projet individuel d'élève, mais son incorporation dans le cours proprement dit en classe Terminale me paraît être une idée valable.

Autre extension plus ambitieuse: incorporer des sous-programmes avec passage de paramètres par valeur. Ou encore instructions multiples sur une même ligne, etc...

1.3.6

Un système déductif sans variables

La recherche dans une base de connaissances (de données) est un secteur au coeur même de toute manipulation d'informations.

Le petit programme qui va être présenté dans ce chapitre n'a bien sûr d'autre intérêt que pédagogique. Sa structure de contrôle est réduite au minimum, et pleinement récursive; l'interface avec l'utilisateur est réduite, mais somme toute assez souple.

Il s'agit de gérer (créer, interroger, mettre à jour) une petite banque de données constituée d'informations du type:

```
-----
(????? etre ?????)
-----
```

Il s'agit donc d'une liste globale \$base\$ constituée de listes (les "informations"); voici l'exemple que nous manipulerons dans les tests:

```
(setq $base$ '(<(oiseau doux)
                <(chien dangereux)
                <(chien animal)
                <(gentil doux)
                <(animal gentil)
                <(epagneul enrage)
                <(epagneul chien)))
```

Deux remarques:

- 1) La liste est donnée dans le désordre le plus complet; on pourrait rajouter des connaissances au début, au milieu ou à la fin. Nous verrons par la suite que les résultats seraient les mêmes. De telles connaissances sont dites "déclaratives": données en vrac, elles ne contiennent pas leur mode d'emploi!
- 2) Le verbe ETRE a été éliminé car inutile dans la représentation interne. Si vous désirez mélanger ETRE, AVOIR, VOULOIR, etc. attendez-vous à des difficultés d'ordre sémantique: l'avantage du verbe ETRE est sa TRANSITIVITE, que l'algorithme donné plus bas exploitera à fond.

Le programme commenté

L'un de ses intérêts est aussi son utilisation de la fonction FILTER, qui est donc supposée chargée en mémoire.

```
-----
; recherche transitive dans une base de connaissances désordonnée.
```

```
(def database (base)
  ; le lanceur du programme
  (prompt 'l)>> l) ; pour ne pas confondre avec la boucle Lisp
  (untilexit fin (executer (read)))
  ; la "boucle-système" du programme, dont on sortira à la 1ère évaluation du
  ; type (fin), où le symbole FIN est le nom d'un échappement.
  (prompt 'l? l) ; on revient sous Lisp
  'lau revoir...l)
```

```
(de executer (E q$ x$ y$)
```

```

; q$ x$ et y$ sont des variables de filtrage.
; E est un ordre fourni au clavier, du style:
; (+ chien etre animal)          ajout d'une règle dans la base
; (- chien etre animal)          retrait d'une règle dans la base
; (- chien etre *)                retrait d'une partie des règles
; (? chien etre animal et fou)   interrogation
; (! * etre animal)              lister les animaux
; (! * etre *)                   lister tout.

```

```

(ifn (filtre '(<(? q$) (? x$) etre (* y$)) E)
  (if (member E '(bonsoir fini stop basta ciao assez exit))
      (exit fin)
      (print 'IReformulez votre demande...!))
  (setq E (cons x$ (delete 'et y$))) ; ETRE et ET sont inutiles
  (selectq q$
    ('+ (ajouter E))
    ('- (eliminer E))
    ('? (rechercher E base))
    ('! (lister E base))
    (t (print 'IReformulez votre demande...!))))

```

```

(de ajouter (E)
  ; pour ajouter la connaissance E dans la variable globale BASE
  (newl base E)) ; i.e. (setq base (cons E base))

```

```

(de eliminer (E)
  ; pour retirer la règle E de la base
  (setq base (mise-a-jour base)))

```

```

(de mise-a-jour (B)
  (cond
    ((null B) nil)
    ((ressemble E (car B)) (mise-a-jour (cdr B)))
    (t (cons (car B) (mise-a-jour (cdr B))))))

```

```

(de rechercher (E B)
  (print (if (verifiable E B) 'oui 'non)))

```

```

(de verifiable (E B a$ b$ c$)
  ; E est du type (A P1 ... Pn) et sera vérifiable si chaque (A Pi) est déduc-
  ; tible
  (filtre '(<(? a$) (? b$) (* c$)) E)
  (if (null c$) ; une seule recherche !
      (deductible [a$ b$] B)
      (and (deductible [a$ b$] B) ; ou la 1ere,
            (verifiable (cons a$ c$) B))) ; et le reste.

```

```

(de deductible (E B)
  ; par exemple essayons de déduire (chien fou)
  (let ((x$ (car E)) (y (cadr E)) (a$) (b$) (c$))
    ; on va voir si B est du type (.....(chien ????). ....) par filtrage:
    (if (filtre '(<(* a$) (<(? x$) (? z$)) (* c$)) B)
        (or (eq y z$) ; gagné
              (deductible [z$ y] base) ; prouver (???? fou)
              (deductible E (cdr B)))) ; ou chercher plus loin.

```

```

(de lister (E B)
  ; pour lister tout ou partie des règles contenues dans la base
  (if B
    (if (ressemble E (car B))
      (print [(caar B) 'l etre l (cadr B)]))
    (self E (cdr B))))

(de ressemble (E1 EB x y)
  ; une autre petite fonction de filtrage pour l'égalité ou la ressemblance de
  ; listes à deux éléments au sens suivant:
  ; (* animal) ressemble à (chien animal)
  ; (chien animal) ne ressemble pas à (chien affamé)
  ; (* *) ressemble à n'importe quoi...
  (filtre '((? x) (? y)) E1)
  (if (eq x '* )
    (or (eq y '* ) (eq y (cadr EB)))
    (and (eq x (car EB))
      (or (eq y '* )
        (eq y (cadr EB))))))

```

Un exemple:

```

? (database $base$)
>> (! * etre doux)
(OISEAU ETRE DOUX)
(GENTIL ETRE DOUX)
>> (? epagneul etre gentil et doux)
OUI
>> (? chien etre doux)
OUI
>> (? chien etre dangeureux) ; la deduction est purement formelle
OUI
>> (? chien etre enrage)
NON
>> pourquoi?
Reformulez votre demande...
>> ciao
= a bientot...
?

```


I.4 Compléments : au-delà du filtrage, l'unification

Ce chapitre n'a pas vraiment été abordé avec les élèves, seules quelques allusions ont été faites au moment de l'étude de PROLOG. Un élève a néanmoins creusé la question, et le résultat sera un article à paraître sous sa signature dans un numéro de la revue "Education et Informatique" consacré à l'Intelligence Artificielle, article dans lequel le code LOGO d'un micro-PROLOG simplifié sera décrit.

Dans le cadre d'un polycopié de l'IREM, nous nous devons d'entrer un peu plus dans les détails, au demeurant fort intéressants, dans l'optique du filtrage dont il a été question précédemment.

Le filtrage est en effet un cas particulier d'unification. De quoi s'agit-il au juste ? D'un "pattern" P contenant des variables de filtrage $X, Y, \dots$ et d'une liste L parfaitement connue, donc sans variable. Bornons-nous au filtrage par expressions Lisp, du genre $(? X)$:

$$P = ((? X) 2 3 (? Y)) \quad \text{et} \quad L = (1 2 3 4)$$

Filtrer L à-travers P consiste à trouver des valeurs pour X et Y telles qu'en remplaçant X par sa valeur et Y par son contenu dans P , on trouve exactement L , i.e. consiste à trouver une liste de substitutions:

$$U = ((1 X) ((4 5 6) Y)) \quad \text{ou} \quad U = (1/X, (4 5 6)/Y)$$

dont la signification est: pour appliquer à P la substitution u représentée par la liste U , substituer 1 à $(? X)$, et $(4 5 6)$ à $(? Y)$ dans P . On veut donc que $u(P) = L$.

Exemple: si $U = ((1 X) (R Y) ((\text{COS } A) Z))$ et $P = (\text{ADD } (? X) (\text{MUL } (? Y) (? Z)))$
----- alors $u(P) = L$ avec:
 $L = (\text{ADD } 1 (\text{MUL } R (\text{COS } A)))$

De manière plus générale, une substitution sera donc une liste (pensée comme une composition de substitutions élémentaires de gauche à droite):

$$U = ((T_1 V_1) \dots (T_n V_n))$$

dans laquelle les V_i sont des variables et les T_i des expressions Lisp.

Deux substitutions U et V peuvent être composées et donner lieu à la substitution $(\text{append } U V)$.

Et l'unification ?

Elle consiste, quant à elle, à autoriser la comparaison directe de 2 patterns P et Q , donc avec présence de variables des deux côtés !

Il s'agira là encore de trouver une substitution U telle que les deux patterns s'unifient (soient EQUAL). Mais l'existence de variables des deux côtés entraîne une modification de la définition; on exigera que:

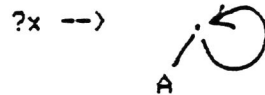
$$u(P) = u(Q)$$

Si on fait du filtrage, i.e. si Q ne contient pas de variable, on a $u(Q) = Q$, et on retombe sur la définition $u(P) = Q$.

On peut montrer (c'est la partie "unicité") que la substitution u ainsi obtenue, si elle existe, est "la plus générale" au sens suivant: toute autre substitution u' unifiant e_1 et e_2 se factorise de manière unique à-travers u . On dit que u est l'unificateur le plus général (upg) de e_1 et e_2 ; il est fourni par l'algorithme précédent.

Dernières remarques avant de passer aux exemples:

- nous avons noté $?x, ?y \dots$ les variables d'unification, conformément au souhait terminant le chapitre sur le filtrage;
- la fonction UNIFIER décrite ci-dessus fait un test d'occurrence à un certain endroit pour vérifier que l'on n'essaie pas (malheur !) d'unifier une variable $?x$ avec une liste la contenant. En effet, essayez donc d'unifier en un temps fini $?x$ avec $(A ?x)$ par exemple: c'est impossible. Notez que ceci EST possible si vous autorisez des "listes infinies" (mieux: des arbres infinis "rationnels", i.e. n'admettant qu'un nombre fini de sous-arbres distincts):



mais ceci est une autre histoire...

Des exemples !

1) ? (unifier '(P A ?x (F (G ?y))) '(P ?z (F ?z) (F ?u)))
= (A ?Z (F A) ?X (G ?Y) ?U)

En clair, l'upg de $P(A, x, F(G(y)))$ et $P(z, F(z), F(u))$ consiste à effectuer simultanément les substitutions:

$A / z \qquad (F A) / x \qquad (G y) / u$

et ces deux expressions s'unifient alors en: $P(A, F(A), F(G(y)))$ i.e. en:

$(P A (F A) (F (G ?y)))$

Nous allons "tracer" la fonction UNIFIER pour suivre le déroulement des appels récursifs:

```
? (trace unifier)
= (UNIFIER)
? (unifier '(P A ?x (F (G ?y))) '(P ?z (F ?z) (F ?u)))
UNIFIER ---> E1=(P A ?X (F (G ?Y))) E2=(P ?Z (F ?Z) (F ?U)) Z1=NIL Z2=NIL
UNIFIER ---> E1=P E2=P Z1=NIL Z2=NIL
UNIFIER <--- NIL
UNIFIER ---> E1=(A ?X (F (G ?Y))) E2=(?Z (F ?Z) (F ?U)) Z1=NIL Z2=NIL
UNIFIER ---> E1=A E2=?Z Z1=NIL Z2=NIL
UNIFIER <--- (A ?Z)
UNIFIER ---> E1=(?X (F (G ?Y))) E2=((F A) (F ?U)) Z1=NIL Z2=NIL
UNIFIER ---> E1=?X E2=(F A) Z1=NIL Z2=NIL
UNIFIER <--- ((F A) ?X)
UNIFIER ---> E1=((F (G ?Y))) E2=((F ?U)) Z1=NIL Z2=NIL
UNIFIER ---> E1=(F (G ?Y)) E2=(F ?U) Z1=NIL Z2=NIL
UNIFIER ---> E1=F E2=F Z1=NIL Z2=NIL
UNIFIER <--- NIL
UNIFIER ---> E1=((G ?Y)) E2=(?U) Z1=NIL Z2=NIL
UNIFIER ---> E1=(G ?Y) E2=?U Z1=NIL Z2=NIL
UNIFIER ---> E1=?U E2=(G ?Y) Z1=NIL Z2=NIL
UNIFIER <--- ((G ?Y) ?U)
UNIFIER <--- ((G ?Y) ?U)
UNIFIER ---> E1= NIL E2=NIL Z1=NIL Z2=NIL
```

```

UNIFIER <--- NIL
UNIFIER <--- ((G ?Y) ?U)
UNIFIER <--- ((G ?Y) ?U)
UNIFIER <---> E1=NIL E2=NIL Z1=NIL Z2=NIL
UNIFIER <--- NIL
UNIFIER <--- ((G ?Y) ?U)
UNIFIER <--- ((F A) ?X (G ?Y) ?U)
UNIFIER <--- (A ?Z (F A) ?X (G ?Y) ?U)
UNIFIER <--- (A ?Z (F A) ?X (G ?Y) ?U)
= (A ?Z (F A) ?X (G ?Y) ?U)

```

Notez bien les répercussions sur les CDR d'une substitution unifiant les CAR.

2) ? (unifier '(Q (F A) (G ?x)) '(Q ?y ?y)) ; Q(F(A),G(x)) et Q(y,y) ?

```

UNIFIER <---> E1=(Q (F A) (G ?X)) E2=(Q ?Y ?Y) Z1=NIL Z2=NIL
UNIFIER <---> E1=Q E2=Q Z1=NIL Z2=NIL
UNIFIER <--- NIL
UNIFIER <---> E1=((F A) (G ?X)) E2=(?Y ?Y) Z1=NIL Z2=NIL
UNIFIER <---> E1=(F A) E2=?Y Z1=NIL Z2=NIL
UNIFIER <---> E1=?Y E2=(F A) Z1=NIL Z2=NIL
UNIFIER <--- ((F A) ?Y)
UNIFIER <--- ((F A) ?Y)
UNIFIER <---> E1=((G ?X)) E2=((F A)) Z1=NIL Z2=NIL
UNIFIER <---> E1=(G ?X) E2=(F A) Z1=NIL Z2=NIL
UNIFIER <---> E1=G E2=F Z1= NIL Z2=NIL
UNIFIER <--- ECHEC
UNIFIER <--- ECHEC
UNIFIER <--- ECHEC
UNIFIER <--- ECHEC
UNIFIER <--- ECHEC
= ECHEC

```

puisque F et G, étant des "constantes" distinctes, ne s'unifient pas !

* Enfin, et pour conclure, une utilisation fondamentale de l'unification est de
 * pouvoir "appliquer une règle de ré-écriture", comme :

```

*
*      (+ ?x 0) --> ?x
*

```

* Une telle règle comporte un membre gauche, par exemple (+ ?x 0), et un membre
 * droit, comme ?x. Etant donné un "énoncé" comme (+ (* a b) 0), on regarde si
 * cet énoncé s'unifie avec le membre gauche de la règle. Si c'est le cas, on
 * applique alors au membre droit les substitutions découlant de l'unification.
 * Dans le cas présent, (+ ?x 0) et (+ (* a b) 0) s'unifient via la substitution
 * ((* a b) ?x) qui appliquée au membre droit ?x, donne (* a b). Ces règles sont
 * à la base des systèmes de "calcul formel", voire de PROLOG lui-même...

PROGRAMMER ->
EN(PROLOG) ;

II.1 Pourquoi la Logique et quelle Logique ?

II.1.1 Introduction.

Jusqu'à présent, nous avons représenté les "connaissances" au moyen de listes (cf. par exemple I.3.5 et I.3.8). Ceci est parfaitement adapté pour mettre en machine des "faits" comme "Socrate est mortel": nous pourrions mettre un indicateur MORTEL sur la p-liste du symbole SOCRATE:

```
? (putprop 'socrate T 'mortel)
= SOCRATE
? (if (getprop 'socrate 'mortel) 'ok)
= OK
```

Mais si nous voulons passer de la "Logique des Propositions" à celle du 1er ordre (i.e. avec variables, mais prédicats constants), nous aurons du mal (sauf à implémenter un tel système de représentation) à stocker et utiliser efficacement une "règle" comme:

"Tout homme est mortel"

que l'on représente en Logique du 1er ordre par:

$$\forall x \text{ homme}(x) \Rightarrow \text{mortel}(x)$$

ou encore, en supposant les variables $x, y, z, \dots$ universellement quantifiées:

$\text{homme}(x) \Rightarrow \text{mortel}(x)$ "entraîne"

ou enfin: (1) $\text{mortel}(x) \Leftarrow \text{homme}(x)$ "si"

L'implication n'est pas vraiment nécessaire:

(2) $\sim \text{homme}(x) \vee \text{mortel}(x)$

si l'on utilise les connecteurs usuels de la Logique:

$\sim$ (NON) & (ET) $\vee$ (OU)

Une formule comme (2) est une "clause": c'est une disjonction (des OU) de "littéraux". Un littéral est une "formule atomique" ou sa négation. Enfin, une formule atomique est une formule du type:

```
homme(x)
plus(x,moins(y,4))
père-de(x,Paul)
père-de(Paul,Jacques)
possible
```

i.e. une fonction avec ou sans arguments.

Exercices: Représenter les phrases suivantes par une formule quantifiée de la logique du 1er ordre:

1) "Une personne est un ascendant d'une autre si elle en est un parent ou un ascendant d'un parent".

$$\forall x \forall y \left\{ \left(\text{parent}(x,y) \vee \left[\exists z \text{ parent}(z,y) \wedge \text{ascendant}(x,z) \right] \right) \Rightarrow \text{ascendant}(x,y) \right\}$$

2) "Un ordinateur est intelligent s'il peut accomplir une tâche qui, réalisée par un homme, demande de l'intelligence".

$$\forall x \{ [\exists y \text{ calcule}(x,y) \ \& \ \text{tache}(y) \ \& \ [\forall h \text{ humain}(h) \ \& \ \text{calcule}(h,y) \Rightarrow \text{intelligent}(h)]] \Rightarrow \text{intelligent}(x) \}$$

3) "Tout le monde fait des erreurs".

$$\forall x \{ \text{humain}(x) \Rightarrow [\exists y \text{ erreur}(y) \ \& \ \text{fait}(x,y)] \}$$

II.1.2 L'algorithme de résolution de Robinson.

Nous avons déjà parlé en I.4 de l'unification comme généralisation du filtrage. En fait, l'algorithme d'unification a été introduit par J.A. Robinson (1966) dans l'article décrivant aussi son "principe de résolution" qui sert à démontrer des théorèmes dans le cadre de la Logique du 1er ordre.

L'idée de "résolution" est en fait dérivée du "raisonnement par l'absurde":

- * Si S est un "état du monde", dont on veut déduire un fait W, on ne
- * prouve pas que $S \Rightarrow W$, mais plutôt que $S \ \& \ \sim W$ conduit à une clause
- * fausse (une contradiction).

Nous noterons $\square$ la clause fausse (par exemple $P \ \& \ \sim P$).

1ère étape: pour prouver une formule logique F quantifiée du 1er ordre, on commence à la mettre sous "forme clausale":

$$F = C1 \ \& \ C2 \ \& \ \dots \ \& \ Cn$$

où les Ci sont des clauses $L1 \vee L2 \vee \dots \vee LK$ disjonctions de littéraux.

Exemple: "Une personne est un ascendant d'une autre si elle en est un parent ou un ascendant d'un parent".

$$F = \text{ascendant}(x,y) \Leftarrow \{ \text{parent}(x,y) \vee [\exists z \text{ parent}(z,y) \ \& \ \text{ascendant}(x,z)] \}$$

Traduisons $Q \Leftarrow P$ par $\sim P \vee Q$:

$$F1 = \text{ascendant}(x,y) \vee \sim \{ \text{parent}(x,y) \vee [\exists z \text{ parent}(z,y) \ \& \ \text{ascendant}(x,z)] \}$$

$$F2 = \text{ascendant}(x,y) \vee \{ \sim \text{parent}(x,y) \ \& \ \sim [\exists z \text{ parent}(z,y) \ \& \ \text{ascendant}(x,z)] \}$$

$$F3 = \text{ascendant}(x,y) \vee \{ \sim \text{parent}(x,y) \ \& \ [\forall z \sim \text{parent}(z,y) \vee \sim \text{ascendant}(x,z)] \}$$

$$F4 = \text{ascendant}(x,y) \vee \sim \text{parent}(x,y) \quad (\text{une 1ère clause})$$

$$\&$$

$$\text{ascendant}(x,y) \vee [\forall z \sim \text{parent}(z,y) \vee \sim \text{ascendant}(x,z)]$$

$$F5 = \text{ascendant}(x,y) \vee \sim \text{parent}(x,y) \quad = C1$$

$$\&$$

$$\text{ascendant}(x,y) \vee \sim \text{parent}(z,y) \vee \sim \text{ascendant}(x,z) \quad = C2$$

Nous avons travaillé de manière purement "mécanique":

- éliminer les $\Rightarrow$
- faire rentrer les NON à l'intérieur des sous-formules
- distribuer les ET sur les OU

Ceci peut donc être facilement automatisé, en LISP par exemple. Si dans la formule "clausale" F5, nous ré-introduisons les $\leq$, il vient quelque chose qui ressemble de près à ce que nous appellerons un "programme PROLOG":

C1 : ascendant(x,y) $\leq$ parent(x,y)

C2 : ascendant(x,y) $\leq$ parent(z,y) & ascendant(x,z)

2ième étape: Etant donné un "monde" S formé d'un nombre fini de clauses:

C1 & ... & Cn

(donc supposées "vraies"), démontrer une formule W.

1) On commence par mettre la négation $\sim W$ sous forme clausale:

$$\sim W = C_{n+1} \quad \& \quad \dots \quad \& \quad C_p$$

2) On considère alors le nouveau monde $S' = S \& \sim W$ formé des clauses:

$$\left\{ \begin{array}{l} C1 \\ C2 \\ \vdots \\ Cn \\ Cn+1 \\ \vdots \\ Cp \end{array} \right.$$

et l'on essaie d'en déduire la clause vide, ce qui démontrera que W est déductible de S.

Exemples: 1) Un exemple "abstrait" en logique des Propositions (sans variables):

le "monde" S $\left\{ \begin{array}{l} C1 : p(a) \vee q(b) \\ C2 : p(a) \vee \sim r(a) \\ C3 : \sim p(a) \vee r(a) \\ C4 : \sim p(a) \vee \sim q(b) \\ C5 : r(a) \vee \sim q(b) \end{array} \right.$

à prouver W : $r(a) \& \sim q(b)$

Considérons $\sim W = \sim r(a) \vee q(b)$, qui est déjà sous forme clausale, et l'on rajoute cette clause à S:

le "nouveau monde" S' $\left\{ \begin{array}{l} C1 : p(a) \vee q(b) \\ C2 : p(a) \vee \sim r(a) \\ C3 : \sim p(a) \vee r(a) \\ C4 : \sim p(a) \vee \sim q(b) \\ C5 : r(a) \vee \sim q(b) \\ C6 : \sim r(a) \vee q(b) \end{array} \right.$

Il faut maintenant prouver l'inconsistance de S'. Pour cela, on va grouper deux clauses qui peuvent "se résoudre", i.e. qui contiennent un fait affirmé dans l'une et nié dans l'autre comme C1 et C3:

C1 : $p(a) \vee q(b)$
C3 : $\sim p(a) \vee r(a)$

$p(a)$ est affirmé
 $p(a)$ est nié

Puisqu'on ne peut pas avoir à la fois $p(a)$ & $\sim p(a)$, on déduit de C1 & C3 la nouvelle clause ("résolvante de C1 et C3"):

$$C7 : q(b) \vee r(a)$$

On rajoute C7 au monde S' et on continue ainsi jusqu'à obtention de la contradiction $\square$:

$$C6 \& C7 \Rightarrow C8 : q(b)$$

$$C2 \& C4 \Rightarrow C9 : \sim r(a) \vee \sim q(b)$$

$$C5 \& C9 \Rightarrow C10 : \sim q(b)$$

$$C8 \& C10 \Rightarrow C11 : \square \quad \text{CQFD}$$

- 2) Un exemple "concret" avec variables quantifiées. Le monde initial consiste en:

M1: "Les gens sérieux sont attentifs"

M2: "Les cancre ne sont pas attentifs"

M3: "Certains cancre sont intelligents"

et nous voulons "prouver":

W : "Il y a des gens intelligents qui ne sont pas attentifs"

On met M1, M2 et M3 sous forme quantifiée:

$$M1: \forall x \text{ sérieux}(x) \Rightarrow \text{attentif}(x)$$

$$M2: \forall x \text{ cancre}(x) \Rightarrow \sim \text{attentif}(x)$$

$$M3: \exists x \text{ cancre}(x) \& \text{intelligent}(x)$$

puis sous forme clausale:

$$S \left\{ \begin{array}{l} C1: \sim \text{sérieux}(x) \vee \text{attentif}(x) \\ C2: \sim \text{cancre}(x) \vee \sim \text{attentif}(x) \\ C3: \text{cancre}(\text{TOTO}) \\ C4: \text{intelligent}(\text{TOTO}) \end{array} \right.$$

Ici, TOTO est un identificateur représentant une "construction" d'un objet x tel que $\text{cancre}(x) \& \text{intelligent}(x)$. L'ordinateur n'appréciant pas les "théorèmes d'existence", on note TOTO une "réalisation" de x dans l'univers du discours, même en suspens. C'est ce que la littérature logique nomme les "constantes de Skolem".

On met ensuite $\sim W$ (la négation du théorème à prouver) sous la forme clausale:

$$C5: \sim \text{intelligent}(x) \vee \text{attentif}(x)$$

Il s'agit donc maintenant de prouver l'inconsistance du monde S' formé des clauses C1...C5, clauses dans lesquelles, ne l'oublions pas, les variables x sont universellement quantifiées.

D'ailleurs, puisque nous allons travailler avec plusieurs clauses à la fois contenant chacune des variables universelles de même nom, il y a tout profit à utiliser des identificateurs distincts pour les variables de chaque clause (rassurez-vous tout de suite: PROLOG le fera tout seul...):

S' { C1: $\sim$ sérieux(x) $\vee$ attentif(x)
 C2: $\sim$ cancre(y) $\vee$ attentif(y)
 C3: cancre(TOTO)
 C4: intelligent(TOTO)
 C5: $\sim$ intelligent(z) $\vee$ attentif(z)

Attaquons alors la "résolution": nous groupons des paires de clauses qui se "résolvent" et rajoutons leur résolvante en nouvelle clause, jusqu'à obtention de la clause vide $\square$. Une nouveauté par-rapport à l'exemple précédent. Deux clauses se résolvent au sens de la Logique du 1er ordre s'il existe un jeu de substitutions des variables de ces clauses tel qu'après substitution, un littéral se trouve affirmé dans l'une et nié dans l'autre (donc deux propositions atomiques s'unifient). Par exemple:

C4 & C5 avec $z = \text{TOTO}$ se résolvent et produisent:

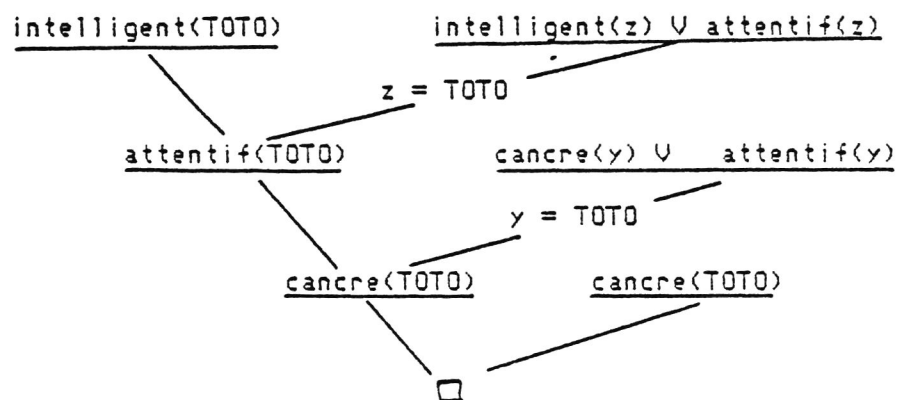
C6: attentif(TOTO)

C6 & C2 avec $y = \text{TOTO}$ se résolvent et produisent:

C7: cancre(TOTO)

Enfin, C7 & C3 se résolvent en $\square$.

On peut visualiser ce processus de réfutation suivant un arbre lu de haut en bas, et dans lequel chaque noeud est la résolvante de ses deux pères (modulo une substitution de tout ou partie des variables):



Il existe plusieurs stratégies de sélection de clauses susceptibles de se résoudre, le conflit majeur résidant entre l'efficacité souhaitable de la méthode d'obtention de $\square$ en économisant au maximum le nombre des résolvantes intermédiaires et la garantie d'aboutir au résultat. Là comme ailleurs, qui dit algorithme dit compromis. Nous verrons plus tard que PROLOG

adopte une démarche très simple (donc parfois couteuse) qui est d'ordonner les clauses et d'effectuer un balayage systématique. De plus, PROLOG n'est pas la Logique du 1er ordre, mais bel et bien un LANGAGE DE PROGRAMMATION, et comme tel va imposer des contraintes sur les objets de son calcul. C'est ce que nous allons voir.

Bibliographie sur la Résolution:

- "Principles of Artificial Intelligence"
N.J. NILSSON, Springer 1982, pp. 145-191
- "Logic: Form and Function"
J.A. ROBINSON, Edinburgh University Press, 1979
(ré-édité par North-Holland)

II.1.3 Les clauses de HORN.

Une clause est une "disjonctions de littéraux" (!) :

$$L_1 \vee \dots \vee L_n$$

chaque L_i étant une proposition atomique ou sa négation:

$$L_1 \vee \dots \vee L_p \vee \sim L_{p+1} \vee \dots \vee \sim L_n$$

en groupant les affirmations et les négations. Mais C peut aussi s'écrire:

$$L_1 \vee \dots \vee L_p \Leftarrow L_{p+1} \& \dots \& L_n$$

dont l'interprétation en termes d'appels de procédures est:

$$\text{POUR prouver } (L_1 \vee \dots \vee L_p), \text{ prouver en séquence } L_{p+1}, \dots, L_n$$

Le "flou" naissant de la forme disjonctive de la conclusion (quels sont les L_i avec $i = 1..p$ qui ont été prouvés ?) incite à ne considérer que certaines clauses dites "de Horn" qui comportent un seul littéral affirmé, comme:

$$\text{ascendant}(x,y) \Leftarrow \text{parent}(x,y)$$

ou encore:

$$\text{ascendant}(x,y) \Leftarrow \text{parent}(z,y) \& \text{ascendant}(x,z)$$

Cette restriction n'est - contrairement à ce que l'on pourrait penser - pas gênante en ce sens précis que pour tout ensemble S de clauses, il existe un ensemble S' de clauses de Horn, tel que S est inconsistent si et seulement si S' l'est (un point de vue plus théorique est exprimé par S.A. Tarnlund dans le rapport cité plus bas: les clauses de Horn permettent de calculer toute fonction calculable par une machine de Turing).

Bien entendu, ces résultats (ne) sont (que) théoriques, mais le fait est que la plupart des problèmes courants se formalisent bien en termes de clauses de Horn, et encore mieux si l'on pense le membre gauche d'une telle clause comme étant en fait un appel de procédure (éventuellement récursif).

La forme générale des clauses (de Horn) que nous utiliserons dorénavant en PROLOG est donc:

$L \leftarrow P_1 \ \& \ \dots \ \& \ P_n$	"règle"
$L \leftarrow$	"fait"

Reprenons par exemple l'exercice sur la relation d'ascendance. Voici le véritable programme PROLOG correspondant:

```

-----
"les faits"  F1  parent(paul,jacques)  <=
              F2  parent(marie,jacques) <=
              F3  parent(pierre,marie) <=

"les règles" R1  ascendant(x,y) <= parent(x,y)
              R2  ascendant(x,y) <= parent(z,y) & ascendant(x,z)
-----

```

II.1.4 L'interprétation procédurale des clauses de Horn.

Les "faits" ont une sémantique claire: "parent(paul,jacques)" signifie que Paul est un parent direct (père ou mère) de Jacques.

Quant aux règles, nous avons utilisé plus haut le POUR connus des habitués de LOGO:

POUR <nom-de-procédure> <arguments>
<suite des opérations de calcul>

L'interprétation d'une règle comme:

$L \leftarrow P_1 \ \& \ \dots \ \& \ P_n$
--

sera en optique-PROLOG:

POUR PROUVER L
PROUVER $P_1 \ \& \ \dots \ \& \ P_n$

Mais que signifie "prouver" L ? N'oublions pas que L peut contenir (et contient en général) des "variables" x, y, ... PROUVER L signifie donc TROUVER UN ENSEMBLE DE VALEURS POUR LES VARIABLES FIGURANT DANS L ET RENDANT L "VRAI", i.e. DEDUCTIBLE DES "FAITS" ET DES "REGLES". Quatre remarques:

- 1) En fait, PROLOG ne va pas se contenter de fournir une "preuve" (i.e. les valeurs ad-hoc des variables), mais va offrir toutes les "preuves" possibles. C'est ainsi qu'on parle de "NON-DETERMINISME" pour PROLOG en ce sens qu'un calcul, pour un certain jeu de données, n'a pas une solution unique. Avec le programme-PROLOG précédent, on peut par exemple vouloir prouver:

ascendant(x,jacques)

et PROLOG va alors nous décrire toutes les valeurs de x qui sont telles que ascendant(x,jacques) soit déductible de notre ensemble de clauses:

x = paul
x = marie
x = pierre

- 2) Va pour la "vérité" et la "preuve". Qu'en est-il donc pour le "faux" ? Très pragmatique, PROLOG dira que L est "FAUX" s'il n'est PAS PROUVABLE, i.e. déductible de l'ensemble des clauses. Il faudrait donc parler d'"inconnu" plutôt que de "faux". Nous aurons plus tard l'occasion de préciser plus à fond ce point de vue en définissant (en PROLOG) le mot réservé NON...
- 3) Dans l'optique du principe de résolution, mieux vaudrait dire "contre-exemple" que "preuve". Au lieu de prouver que W est déductible de $P_1 \& \dots \& P_n$ on tente plutôt de TROUVER TOUS LES CONTRE-EXEMPLES montrant l'absurdité de l'assertion $P_1 \& \dots \& P_n \& \sim W$. Si l'on n'en trouve pas, c'est que W n'est pas prouvable (si W ne contient pas de variable, il faudra bien un moyen pour distinguer un énoncé déductible d'un énoncé dont la négation n'admet aucun contre-exemple; nous y reviendrons dans II.2).
- 4) Les manuels de PROLOG II, que nous utilisons, ont un autre mot que "prouver", celui d'"effacer", dans l'optique d'un système de ré-écriture. Il est en effet significatif de la démarche de résolution: pour aboutir à la clause vide à-partir de l'assertion

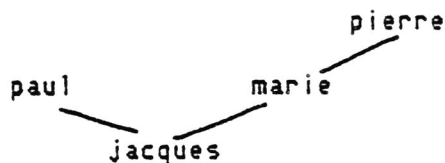
(A) ascendant(x,jacques)

on va chercher dans notre base de clauses, et de haut en bas, la première règle dont le membre gauche s'unifie avec (A), et qui est (R1) avec $y = \text{jacques}$:

(R1) ascendant(x,jacques) $\Leftarrow$ parent(x,jacques)

Il faut alors effacer parent(x,jacques), ce qui est possible à-travers les substitutions:

x = paul OU x = marie



Si l'on commence par $x = \text{paul}$, on obtient une 1ère solution de notre problème

1ère solution : x = paul

Ce choix est remis en cause, puisque l'on peut aussi choisir $x = \text{marie}$:

2ième solution: x = marie

Il n'y a alors plus de solution en x, donc plus de possibilité d'effacement de parent(x,jacques). Mais alors on remet en cause le choix de la règle (R1) dont le membre gauche s'unifiait avec ascendant(x,jacques). Le prochain choix possible est la règle (R2) avec $y = \text{jacques}$:

(R2) ascendant(x,jacques) $\Leftarrow$ parent(z,jacques) & ascendant(x,z)

Cette règle demande alors l'effacement de parent(z,jacques), et on retrouve les deux choix intermédiaires $z = \text{paul}$ et $z = \text{marie}$.

Pour $z = \text{paul}$, il faut alors effacer ascendant(x,paul), ce qui n'est pas possible (nous ne déroulons pas toute la résolution, cf. l'arbre ci-dessous). On essaie alors $z = \text{marie}$, en tâchant d'effacer:

ascendant(x,marie)

et pour cela, on remonte au sommet des clauses et on cherche la 1ère règle dont le membre gauche s'unifie avec ascendant(x,marie). C'est (R1) avec

$y = \text{marie}$, qui demande l'effacement de $\text{parent}(x, \text{marie})$, possible avec:

3ième solution: $x = \text{pierre}$

Le choix de (R1) est remis en cause par celui de (R2) avec $y = \text{marie}$:

(R2) $\text{ascendant}(x, \text{marie}) \leftarrow \text{parent}(z, \text{marie}) \ \& \ \text{ascendant}(x, z)$

qui demande d'effacer $\text{parent}(z, \text{marie})$, ce qui n'est faisable qu'avec $z = \text{pierre}$, mais alors $\text{ascendant}(x, \text{pierre})$ ne s'efface plus. Nous sommes alors en situation de blocage, et disposons de toutes les solutions de notre problème. Ouf.

L'arbre de la page suivante montre le cheminement de la recherche, avec ses retours-arrière ("back-track").

Le prochain paragraphe II.2 va quitter la théorie pour examiner le système PROLOG II, utilisé sur Apple II avec les élèves, en environnement UCSD, qu'ils ont déjà pratiqué avec le langage Pascal.

Bibliographie sur les clauses de Horn.

- "Logic for problem solving"
R. Kowalski, North-Holland, 1979.
- "Predicate Logic as programming language"
R. Kowalski, in Actes Congrès I.F.I.P. 1974.
- "Logic Information Processing"
S.A. Tarnlund, Univ. of Stockholm, 1975.

II.2 Le système PROLOG II et son environnement Pascal

II.2.1 Historique.

Deux interprètes PROLOG sont principalement disponibles sur micro-ordinateurs:

- MICRO-PROLOG, réalisé par F.G. McCabe à Londres, pour les micro-ordinateurs des établissements scolaires de sa gracieuse Majesté. Il fonctionne essentiellement sous le système d'exploitation CP/M-80, et se trouve en écrivant à:

Logic Programming Associates
10 Burntwood Close
London SW183JU

Des versions pour Apple II avec la carte Z-80, et pour IBM-PC, sont en particulier disponibles. Il fut (et reste) utilisé dans une école londonienne avec des élèves dont les plus jeunes ont 11 ans (mais oui...). Les résultats sont apparemment encourageants, et de plus amples détails (avec exemples de programmes réalisés par les enfants sur des sujets allant des astres ... à la 1ère guerre mondiale) sont donnés dans:

"Children program in PROLOG"
R. Ennals
DOC 81/8, new printing Oct. 1983 (Imperial College, London)

(voir aussi l'article du même auteur dans le numéro spécial de la revue "Education & Informatique" consacré à l'Intelligence Artificielle, Automne 1984).

La représentation est parenthésée, assez proche de celle de LISP:

((Ascendant x y) (Parent z y) (Ascendant x z))
((Parent Marie Paul))

mais un langage de surface (SIMPLE) en permet un considérable assouplissement, vital pour les jeunes âmes:

x ascendant-of y if z parent-of y and x ascendant-of z
Marie parent-of Paul

- PROLOG II sur Apple II, sous le système d'exploitation UCSD (donc avec tous les avantages afférents, à-savoir éditeur pleine page, etc...). Il provient des concepteurs mêmes du langage, M. van Caneghem, H. Kanoui et A. Colmerauer du Groupe d'Intelligence Artificielle de l'Université d'Aix-Marseille, où PROLOG est né des travaux d'A. Colmerauer sur le traitement des langues naturelles. C'est celui dont il sera question ci-dessous.

Le premier interprète fut écrit en FORTRAN, baptisé PROLOG I, et les étudiants de nombreuses universités n'ont que cette version sur MULTICS. Les lourdeurs de ce premier interprète ont été rectifiées dans la version PROLOG II. Espérons que cette dernière tournera un jour sur matériel Education Nationale !

Pour cette seconde réalisation, les concepteurs ont été amenés à dessiner une machine virtuelle qu'ils baptisèrent "Micromégas" (qui est l'analogue de la p-machine virtuelle du Pascal UCSD) et qui est la seule partie à être dépendante de la machine (du micro-processeur). L'interprète PROLOG II est donc écrit en Micromégas, mais il a fallu écrire la machine Micromégas ! Pour cela, les auteurs ont développé - en Pascal - un langage spécialement adapté, portant le doux nom de Candide, et la machine Micromégas fut écrite en Candide...

Enfin, PROLOG est particulièrement gourmand en mémoire, et ceci est renforcé par la présence en mémoire centrale de l'interpréteur PROLOG... et de l'interpréteur Pascal ! Il fut alors décidé d'utiliser un disque souple 5" comme mémoire virtuelle (dans le second lecteur de disques), en ne mettant en mémoire centrale que les informations nécessaires à court terme.

Le système PROLOG II est donc relativement lent, mais à son actif, il a deux bons points:

- i) il est d'une grande fiabilité (nous n'avons pas encore trouvé de "bug", mais patience...);
- ii) il autorise de larges programmes (une quarantaine de pages, paraît-il).

Notre expérience avec les élèves a montré qu'ils acceptent volontiers de sacrifier le temps (dont ils ont en général l'obsession) et le graphique au profit de la nouveauté conceptuelle du langage, à condition de leur montrer qu'il ne s'agit pas d'un gadget, mais d'un style d'écriture et de pensée qui entre pour une bonne part dans le futur de l'Informatique. A cette occasion, un article leur a été distribué, dans lequel il est question de la 5^{ème} génération d'ordinateurs et de PROLOG:

"Introduction to the Fifth Generation"
Pamela Mc Corduck
Comm. ACM, Sept. 1983

Pourquoi enfin a-t-on choisi l'interprète de Marseille au profit de celui de Londres ?

Travaillant avec des élèves de Terminale ayant déjà utilisé l'UCSD, et pratiqué la programmation, le problème de la syntaxe n'était pas crucial au point de nécessiter le langage d'interface dont il était question avec MICRO-PROLOG. Bien plus, la clarté du style de règles PROLOG II:

```
ascendant(x,y) -> parent(z,y) ascendant(x,z);
parent(marie,paul) ->;
```

avec l'interprétation "si" du signe ->, et "et" du caractère (espace), en font un outil d'usage somme toute assez confortable. Il n'est pas si évident que le choix n'aurait pas été différent avec de très jeunes enfants (ne serait-ce que du fait des accès-disques constants risquant de briser l'attention). Une version de PROLOG II devrait voir le jour sur Micral 16 bits, et surtout sur le nouveau "Macintosh" d'Apple (version 512 K) annoncée par M. Van Caneghem pour début 85. Ces deux versions très attendues devraient éviter ces problèmes.

II.2.2 Comment entrer un programme PROLOG II ?

De deux manières: à-partir de l'éditeur du Pascal-UCSD, ou directement sous PROLOG.

II.2.2.1 A-partir de l'éditeur UCSD

Un programme PROLOG n'est autre qu'une succession de clauses séparées par un ';', la dernière se terminant par un ';;'. Des commentaires peuvent être insérés, encadrés par des " qui délimitent aussi les chaînes de caractères de PROLOG II. Voici un exemple de fichier PARENT.TEXT:

"la parente"

```
pere(edmond,jean) ->;
```

```
parent(x,y) -> pere(x,y);
```

```
parent(x,y) -> mere(x,y);
```

"les relations d'ascendance"

```
ascendant(x,y) -> parent(x,y);
```

```
ascendant(x,y) -> parent(x,z) ascendant(z,y);
```

"on peut mettre les littéraux sur plusieurs lignes:"

```
gd-parent(x,y) -> parent(x,z)
                  parent(z,y);;
```

N'oubliez pas le double point-virgule à la fin du fichier. Toutes les clauses ayant même tête (par ex. toutes les clauses 'parent(...) -> ...') doivent être regroupées dans un même "paquet", sous peine d'en voir certaines non reconnues, silencieusement ou avec le message d'erreur de PROLOG:

"ON NE PEUT PAS INSERER DANS CE PAQUET"

ou encore:

"CE PAQUET EXISTE AILLEURS"

Cette contrainte est uniquement due à un souci d'efficacité, étant quelque peu étrangère à l'esprit de PROLOG.

Une fois le programme rédigé, il n'y a qu'à le sauver sous forme usuelle d'un fichier-texte UCSD. Au moment du rappel en mémoire PROLOG, il faudra qu'il soit présent dans la disquette 'PROLOG:' en drive 4; nous verrons plus loin comment s'effectuera le chargement.

II.2.2.2 Sous l'éditeur PROLOG

Il faut tout d'abord "entrer" sous PROLOG. Pour cela, on tape au niveau "Command" de l'UCSD la commande X (eXecute), puis PROLOG, en ayant pris soin, auparavant, d'entrer Ctrl-E pour activer les minuscules (ceci est bien sûr inutile si, comme il est conseillé, vous disposez d'une carte 80 colonnes):

```
m.van caneghem a.colmerauer h.kanoui
gia luminy
```

```
bienvenue a la version Apple de
PROLOG II version 1.1 mars 82
aujourd'hui dimanche 15 juillet 1984
```

```
>
```

Après la bienvenue, le prompt > apparaît, en attente d'une suite de littéraux à effacer. Tout ordre donné à l'interprète PROLOG sera perçu par ce dernier comme un ordre d'effacement (comme LISP attendait une demande d'évaluation). Certaines manœuvres, comme le changement de date, s'effacent ipso-facto et ne sont là que pour leurs effets de bord:

```
>date(16);
lundi 16 juillet 1984
```

```
>date(11,aout);
samedi 11 aout 1984
```

```
>
```

PROLOG II dispose donc d'un calendrier perpétuel, écrit en PROLOG, et listé à la fin du manuel de référence. Entièrement listé se trouve en fait tout le "superviseur", qui contient les règles d'usage courant ou proche du système. PROLOG, notons-le immédiatement, indique une réponse affirmative par un saut de ligne avant le renvoi du prompt.

Pour insérer en mémoire (virtuelle) le fichier-texte PARENT.TEXT en drive 4, il faut désigner l'unité d'entrée et demander l'insertion du fichier, puis redonner à la console le rôle principal:

```
>entree("parent") inserer entree("console");
```

```
>
```

Si nous voulons lister la mémoire (i.e. les "énoncés" qu'elle contient), il faut savoir que cette mémoire est gérée de haut en bas par un "pointeur d'énoncés". Pour positionner ce pointeur "en haut du monde" (resp. en bas), on demande l'effacement du littéral HAUT (resp. BAS); pour lister N énoncés à-partir du pointeur, on utilise le prédicat LISTER:

```
>haut lister(4);
"la parente"

pere(edmond,jean) ->;

parent(x,y) -> pere(x,y);
parent(x,y) -> mere(x,y);
```

```
>
```

Pour lister la totalité des énoncés, il suffit de prendre N assez grand (le plus grand entier admissible est 2897151). Pour un écho sur imprimante de ce qui apparaît à l'écran (entrée + sortie), on demande l'effacement du prédicat PAPIER (et SANS-PAPIER pour l'annuler). Une fois rassuré, vous pouvez interroger PROLOG:

```
> parent(edmond,jean);

> parent(sophie,jean);
> parent(x,jean);
x=edmond
>
```

Nous avons dit plus haut que PROLOG II disposait d'un éditeur d'énoncés. Pour y accéder, il n'y a qu'à le demander:

```
>editer;
```

L'écran s'efface et la ligne d'invite apparaît en haut, suivie du prompt + de l'éditeur:

```
m<ont d<esc h<aut b<as e<n l<ster x<cut
s<pprimer c<hnger r<nommer i<nsrer f<in
```

```
+
```

Nous n'allons pas détailler toutes les commandes (cf. le manuel de référence), seulement celles qui sont les plus utiles, comme:

- h<aut et b<as (mettre le pointeur en haut ou en bas du monde des énoncés)
- l<ster (suivi du nombre d'énoncés à lister, 1 par défaut)
- m<ont et d<esc (pour monter ou descendre de N énoncés, 1 par défaut)
- i<nsrer (pour insérer un ou plusieurs énoncés, clauses ou commentaires, avant le pointeur courant. L'insertion se termine par un ;;)
- s<pprimer (pour supprimer N énoncés à-partir du pointeur courant, 1 par défaut)
- f<in (pour sortir de l'éditeur et retourner sous le prompt >>)

Importante pour nous est la possibilité d'insérer un fichier à-partir de l'éditeur via la commande i<nsrer suivie du nom de fichier entre doubles quotes comme toutes les "chaines de caractères" en PROLOG.

Supposons que la mémoire soit vide. Voici un exemple de session avec l'éditeur:

```
-----
m<ont d<esc h<aut b<as e<n l<ster x<cut
s<pprimer c<hnger r<nommer i<nsrer f<in

+h

;FIN DU MONDE: ordinaire
+i "parent"
"la parente"

pere(edmond,jean) ->;

parent(x,y) -> pere(x,y);
parent(x,y) -> mere(x,y);

"les relations d'ascendance"

ascendant(x,y) -> parent(x,y);
ascendant(x,y) -> parent(x,z) ascendant(z,y);

"on peut mettre les litteraux sur plusieurs lignes:"

gd-parent(x,y) -> parent(x,z)
                  parent(z,y);

+l

gd-parent(x,y) ->
  parent(x,z)
  parent(z,y);

+tm 7

pere(edmond,jean) ->;
```

```

+s
parent(x,y) -> pere(x,y);

+i
pere(louis,jean) ->;
pere(gerard,louis) ->;
mere(sophie,louis) ->;

+h
"la parente"

+f

>gd-parent(x,y);
x=gerard y=jean
x=sophie y=jean
>
-----

```

Notre programme a donc été modifié sous éditeur après son insertion. Si nous voulons sauver sur disque cette nouvelle version, nous pouvons là encore le faire sous le prompt de PROLOG:

```
>sortie("parent") haut lister(100) sortie("console");
```

qui va lister 100 énoncés sur l'unité de sortie précisée; mais nous pouvons aussi le faire à-partir de l'éditeur:

```

+h
"la parente"

+l 100 "parent"

```

Vous avez sans doute noté le message:

```
;FIN DU MONDE: ordinaire
```

indiquant la fin du monde (sic) usuel, dénommé "ordinaire". Le fait est que l'espace des énoncés est découpé en "mondes" ayant chacun son propre domaine de visibilité, et organisés hiérarchiquement. Nous n'en dirons pas plus, ne les utilisant pas dans les pages qui suivent (la "vraie" version de Le_Lisp utilise de même un découpage arborescent de la zone des symboles).

II.2.3 Un petit pas vers la syntaxe des objets PROLOG II

Il ne s'agit pas de décrire la véritable "grammaire" des objets manipulés par ce système, mais seulement, dans un premier temps (le second sera celui des listes), de se mettre d'accord sur la représentation d'une "variable" et d'une "constante".

Pour nous, les prédicats seront toujours des constantes, bien que PROLOG, très laxiste, ait parfois des reflets de Logique du Second Ordre (quantification des prédicats).

- les CONSTANTES seront les entiers (421), les chaînes de caractères ("trois") et les identificateurs, ces derniers débutant par au moins deux lettres suivies de lettres ou chiffres (xy, parent, arcl), mais pouvant contenir un trait d'union (grand-parent) ou se terminer par un ou plusieurs ' (in-car').
- les VARIABLES ne diffèrent des constantes qu'en ne débutant par une seule lettre (x, x1, x1', x-nouveau, x-nouveau', x'-nouveau).

Bibliographie

- les manuels de référence PROLOG II, à-savoir:
 - Manuel de la version Apple II
 - Manuel de description du modèle théorique
 - Manuel d'exemples (disquette disponible)
- manuels décrivant Micromégas et Candide, pour les personnes intéressées par l'architecture interne.
- un ouvrage du GIA de Marseille-Luminy, à paraître (chez Inter-Editions ?).

Pour tous renseignements, contacter: PROLOGIA
278, rue Saint-Pierre
13005 Marseille

II.3.1

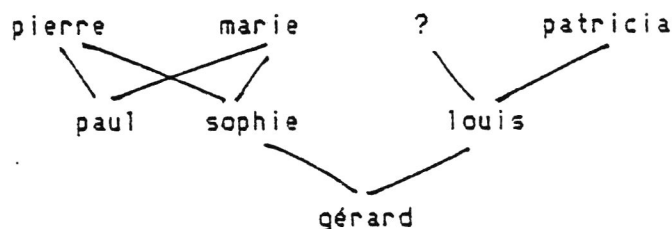
Gérer des informations non numériques

Comme Lisp (encore que pour ce dernier, ceci peut se discuter), le langage PROLOG n'est pas vraiment adapté au calcul numérique; leur domaine commun de prédilection est le traitement symbolique, même s'ils ne l'abordent pas de manière identique, loin s'en faut.

Lisp est un langage de programmation dont on peut dire que la seule structure de contrôle est l'application fonctionnelle. De même, PROLOG est gouverné par le principe de résolution (même sous une forme assez contrainte), qui à son tour utilise deux mécanismes fondamentaux, l'unification et le back-track.

Nous allons utiliser les capacités de PROLOG à "représenter un monde" en montrant comment il permet de gérer une petite base de données, et surtout la grande variété des questions qu'il permet de poser sur cette base.

Un premier aperçu en a été donné en II.1 sur la relation d'ascendance familiale. Le premier programme que les élèves ont eu à écrire consistait précisément à décrire le microcosme familial suivant:



dans lequel les flèches sont interprétées, de haut en bas, par la relation "être parent de". Le prédicat de base imposé était:

individu(x,s,p,m)

signifiant que l'individu nommé x a pour sexe s (masc ou fem), pour père p et pour mère m. Il était demandé de décrire l'effacement des prédicats:

- père
- mère
- parent
- gd-parent
- frère
- soeur
- oncle
- tante
- neveu

Le prédicat PERE s'entend ainsi:

père(x,y) s'efface ssi x est le père de y

et il en est de même pour les autres.

La constante INCONNU représentera un parent inconnu. L'oubli du test "connu ou non" a été quasiment le seul défaut trouvé dans certains programmes; il était crucial dans le prédicat FRERE par exemple. Les élèves ont ressenti le besoin d'un prédicat permettant de tester les différences. Il existe en PROLOG II et se

nomme DIF. Son effacement doit être vu en fait comme celui d'une contrainte supplémentaire ajoutée à l'effacement des autres prédicats, l'effacement étant subordonné à la réalisation de cette contrainte.

"les données familiales"

```
individu(pierre,masc,inconnu,inconnu) ->;
individu(marie,fem,inconnu,inconnu) ->;
individu(paul,masc,pierre,marie) ->;
individu(sophie,fem,pierre,marie) ->;
individu(louis,masc,inconnu,patricia) ->;
individu(patricia,fem,inconnu,inconnu) ->;
individu(gerard,masc,louis,sophie) ->;
```

A noter, Louis, né de père inconnu (ou non répertorié), mais de mère connue.

"les relations parentales"

```
parent(x,y) -> pere(x,y);
parent(x,y) -> mere(x,y);

gd-parent(x,y) -> parent(x,z) parent(z,y);

pere(x,y) -> individu(y,s,x,z);
mere(x,y) -> individu(y,s,z,x);

frere(x,y) -> individu(x,masc,p,m)
              dif(p,inconnu)
              dif(m,inconnu)
              individu(y,s,p,m)
              dif(x,y);

soeur(x,y) -> individu(x,fem,p,m)
              dif(p,inconnu)
              dif(m,inconnu)
              individu(y,s,p,m)
              dif(x,y);

oncle(x,y) -> frere(x,z) parent(z,y);

tante(x,y) -> soeur(x,z) parent(z,y);

neveu(x,y) -> individu(x,masc,p,m) oncle(y,x);
neveu(x,y) -> individu(x,masc,p,m) tante(y,x);
```

N.B. Dans les deux dernières clauses, nous aurions pu "factoriser" par le littéral "individu(x,masc,p,m)" en introduisant un prédicat TANTONCLE se subdivisant en TANTE et ONCLE. En effet, en cas de gain sur le littéral "individu(x,masc,p,m)" et d'échec sur "oncle(y,x)", l'interprète va de nouveau tenter d'effacer "individu(x,masc,p,m)" au cours de son second essai d'effacement, ce qui est une perte de temps. Nous vous laisserons souvent optimiser ainsi les programmes présentés.

Nous pouvons maintenant interroger de diverses façons cette petite base de connaissances familiales:

- Donner tous les individus de sexe féminin, puis seulement ceux de parents connus:

```

>individu(x,fem,p,m);
x=marie p=inconnu m=inconnu
x=sophie p=pierre m=marie
x=patricia p=inconnu m=inconnu
>individu(x,fem,p,m) dif(p,inconnu) dif(m,inconnu);
x=sophie p=pierre m=marie

```

- Qui est grand-parent de Gérard ?

```

>gd-parent(x,gerard) dif(x,inconnu);
x=pierre
x=marie
x=patricia

```

- Qui sont les petit-enfants de Pierre ?

```

>gd-parent(pierre,e);
e=gerard

```

- Donner les couples connus Grand-parent---petit-enfant

```

>gd-parent(x,y) dif(x,inconnu);
x=pierre y=gerard
x=marie y=gerard
x=patricia y=gerard

```

- Qui est l'oncle de Gérard ?

```

>oncle(x,gerard);
x=paul

```

On voit que l'une des richesses pédagogiques de PROLOG ne consiste pas seulement à construire le "programme", mais aussi à le questionner. Ce qui n'est pas toujours aussi immédiat que les exemples ci-dessus peuvent le laisser croire. En effet, en décrivant l'effacement du prédicat ONCLE par exemple, comment être sûr que l'interprète acceptera dans la question ONCLE(--,--) que la variable porte sur l'argument de gauche ou de droite, ou des deux ? Le concept de "paramètre d'entrée" semble se dissoudre, mais nous verrons plus loin que cette propriété "d'inversibilité" des programmes PROLOG n'est pas générale et pose parfois bien des problèmes.

II.3.2

L'arithmétique de Peano

Très jolie application de PROLOG, et vif succès auprès des élèves, une fois le choc passé qui les astreindra à écrire `suc(suc(suc(0)))` en lieu et place de ce bon vieux 3...

L'arithmétique de Peano considère que les nombres entiers (naturels) sont "engendrés" à-partir de 0 par la fonction SUCESSEUR. Un objet `n` de l'univers sera donc considéré comme "entier" s'il s'agit de 0 ou bien du successeur d'un entier. Cette définition (récursive) se traduit en clauses PROLOG par:

```
-----
entier(0)      ->;
entier(suc(n)) -> entier(n);
-----
```

spécifiant bien que pour prouver que `suc(n)` est un entier, il suffit de prouver qu'il en est de même pour `n`.

N.B. Conformément à la syntaxe PROLOG II, nous avons noté SUC et non S la fonction "successeur", le symbole S désignant une variable.

Nous sommes d'ores et déjà capables de "reconnaitre" les entiers naturels:

```
>entier(0) entier(suc(suc(0))) exm("oui");
oui
>entier(2) exm("oui");
>
```

et le symbole 2 n'est pas reconnu comme "entier" puisque n'obéissant pas à la loi de formation adoptée. N'oubliez pas que l'effacement du dernier littéral d'une requête suppose effacés tous ceux qui le précèdent.

Nous pouvons même demander la liste des entiers naturels, ce qui risque d'être long. Dès que la fatigue gagne, taper sur la touche ESC pour provoquer une interruption fatale des effacements en cours:

```
>entier(n);
n=0
n=suc(0)
n=suc(suc(0))
n=suc(suc(suc(0)))
n=suc(suc(suc(suc(0))))
n=
VOUS VENEZ D'APPUYER SUR LA TOUCHE ESC
>
```

il s'agit maintenant de définir les opérations algébriques sur ces entiers, et tout d'abord l'addition: `somme(n,m,s)` s'effacera si et seulement si `n+m=s`. Nous pouvons choisir de "vider" un entier - par exemple `n` - dans la somme `s`, jusqu'à obtention d'une addition avec 0, qui est évidente. Autrement dit, observons récursivement que:

$$0 ++ n = n \qquad \text{et} \qquad (n + 1) ++ m = (n ++ m) + 1 = s + 1$$

Pourquoi avons-nous noté `+` et `++` ? Tout simplement parce que ces symboles ne représentent pas, au fond, la même opération. Le premier n'est autre que la fonction de succession, tandis que l'autre est la fonction d'addition générale, à construire:

```

-----
somme(0,n,n)          -> naturel(n);
somme(suc(n),m,suc(s)) -> somme(n,m,s);
-----

```

Dans la première clause, le membre droit "si naturel(n)" n'est pas vraiment nécessaire pour les calculs. Il garantit seulement qu'une tentative d'effacement d'une requête telle que :

```
somme(0,bateau,bateau)
```

ne s'efface pas, BATEAU n'étant pas un entier à la Peano. En contrepartie, il rajoute un temps proportionnel à n, et peut induire des effets imprévus (cf. plus bas).

```

> somme(suc(0),suc(suc(0)),s);
s=suc(suc(suc(0)))
> somme(n,suc(suc(0)),suc(suc(suc(0))));
n=suc(0)
> somme(suc(suc(0)),m,suc(suc(suc(suc(0)))));
m=suc(suc(0))
> somme(n,m,suc(suc(0)));
n=0 m=suc(suc(0))
n=suc(0) m=suc(0)
n=suc(suc(0)) m=0
>

```

Le "programme" SOMME est ainsi totalement inversible: dès que l'un au moins de ses arguments est instancié (a reçu une valeur), tous les autres sont calculables, éventuellement de manière non déterministe (toutes les solutions sont fournies). Le fait que notre domaine de calcul (celui des entiers) ne soit pas borné nous empêche de demander l'effacement de somme(n,m,s) qui resterait bloqué sur le choix n=0.

La multiplication de deux entiers est définie récursivement en remarquant que :

$$(n + 1) * m = (n * m) ++ m = p$$

soit :

```

-----
produit(0,n,0)        -> naturel(n);
produit(suc(n),m,p) -> produit(n,m,p1) somme(p1,m,p);
-----

```

Le calcul du produit de deux entiers s'en suit (ici $2 * 3$):

```

> produit(suc(suc(0)),suc(suc(suc(0))),p);
p=suc(suc(suc(suc(suc(suc(0))))))
>

```

et l'on pourrait être tenté de croire à une inversibilité analogue à celle de SOMME :

```

> produit(suc(0),m,suc(0));
m=suc(0)

```

VOUS VENEZ D'APPUYER SUR LA TOUCHE ESC

```
>
```

qui n'a donc pas lieu. La solution m=1 est obtenue, mais le programme ne termine pas; que s'est-il passé ? Pour effacer la requête :

```
produit(suc(0),m,suc(0))
```

PROLOG l'unifie avec le membre gauche (la tête) de la seconde clause "produit"

```
produit(suc(x1),x2,x3) -> produit(x1,x2,x4) somme(x4,x2,x3)
```

dont les variables ont été renommées, unifiant ainsi x1 avec , x2 avec m et x3 avec suc(0). Il reste donc à effacer:

```
produit(0,m,x4) somme(x4,m,suc(0))
```

Le premier but s'efface en utilisant la règle 1:

```
produit(0,x5,0) -> naturel(x5);
```

et la nouvelle liste de buts devient:

```
naturel(m) somme(0,m,suc(0))
```

et le premier "naturel(m)" va provoquer un back-track sur toutes les valeurs entières possibles de m:

m=0 :	somme(0,0,suc(0))	=> échec
m=suc(0) :	somme(0,suc(0),suc(0))	=> GAIN : m=suc(0)
m=suc(suc(0))	somme(0,suc(suc(0)),suc(0))	=> échec
m=suc(suc(suc(0)))	somme(0,suc(suc(suc(0))),suc(0))	=> échec
....		

et notre programme, bon enfant, part à l'aventure. Supprimons donc les conditions "si naturel(n)" évoquées plus haut:

```
somme(0,n,n) ->;
produit(0,n,0) ->;
```

et essayons à nouveau:

```
>produit(suc(0),n,suc(0));
n=suc(0)
>produit(suc(suc(0)),n,suc(suc(suc(suc(suc(suc(0)))))));
n=suc(suc(suc(0)))
>
```

avec terminaison du programme. PROLOG - souvent qualifié de langage "déclaratif" - ne dispense pas toujours d'une réflexion (ici mineure) sur les variables à instancier en entrée.

Notez que ceci ne résoud pas le problème symétrique:

```
>produit(n,suc(0),suc(0));
n=suc(0)
```

VOUS VENEZ D'APPUYER SUR LA TOUCHE ESC

```
>
```

que nous vous laissons méditer. Morale: en écrivant un programme PROLOG - i.e. en décrivant l'effacement d'un prédicat - soyons lucide sur la manière dont nous l'utiliserons (quelles seront les variables "libres" et celles "prises" ?).

N.B. Il est aussi possible de prendre le problème à bras-le-corps; il existe deux prédicats LIBRE et PRIS permettant de tester si une variable a déjà été instanciée ou non:

```

>somme(suc(0),suc(0),s) pris(s);
s=suc(suc(0))
>somme(suc(0),suc(0),s) libre(s);
>

```

En effet, après l'effacement de "somme(suc(0),suc(0),s)", la variable s a été instanciée à suc(suc(0)) et donc "pris(s)" s'efface, tandis que "libre(s)" ne s'efface pas. Nous pouvons ainsi aiguiller sur divers traitements suivant les jeux de variables libres, et n'en dirons pas plus. Une dernière façon de faire consistait à ne travailler qu'avec de "petits" entiers. Notez en effet que $\text{produit}(n,m,p) \Rightarrow \text{infegal}(n,p) \ \& \ \text{infegal}(m,p)$; nous pouvons ainsi ajouter des contraintes ayant pour effet de "borner" le calcul.

Continuons l'arithmétique à la Peano. La relation de divisibilité est fondamentale; ne pouvant utiliser l'inversibilité du prédicat PRODUIT, nous allons la décrire pour elle-même:

```

divise(n,0) ->;
divise(suc(n),m) -> somme(suc(n),r,m) divise(suc(n),r);

```

Vous avez reconnu la chanson "en suc(n) combien de fois m?" consistant à "purger" peu à peu tous les m de l'entier suc(n), en utilisant pour cela une part d'inversibilité du prédicat SOMME, calculant r une fois n et m connus, i.e. effectuant une soustraction:

```

>divise(suc(suc(0)),suc(suc(suc(suc(suc(suc(0))))))) exm("oui");
oui
>divise(suc(suc(0)),suc(suc(suc(0)))) exm("oui");
>

```

Il est alors possible d'obtenir tous les diviseurs d'un entier:

```

>divise(d,suc(suc(suc(suc(0)))));
d=suc(0)
d=suc(suc(0))
d=suc(suc(suc(suc(0))))
>

```

et donc les diviseurs communs 1, 2, 3 et 6 des deux entiers 6 et 12 par ex:

```

>divise(d,suc(suc(suc(suc(suc(suc(0)))))))
divise(d,suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(0))))))))))));
d=suc(0)
d=suc(suc(0))
d=suc(suc(suc(0)))
d=suc(suc(suc(suc(suc(suc(0))))))
>

```

L'infinité des nombres pairs s'obtient par:

```

>divise(suc(suc(0)),n);
n=suc(suc(0))
n=suc(suc(suc(suc(0))))
n=suc(suc(suc(suc(suc(suc(0))))))
n=
VOUS VENEZ D'APPUYER SUR LA TOUCHE ESC
>

```

Nous vous proposons d'écrire vous-mêmes (plusieurs versions possibles) les prédicats PAIR(n) et IMPAIR(n), puis ceux de comparaison INFEGAL(n,m) et INFSTRICT(n,m), etc.

II.3.3

Les listes

Où nous allons retrouver la structure de donnée propre à Lisp, mais avec le mécanisme d'évaluation en moins, remplacé par celui de résolution.

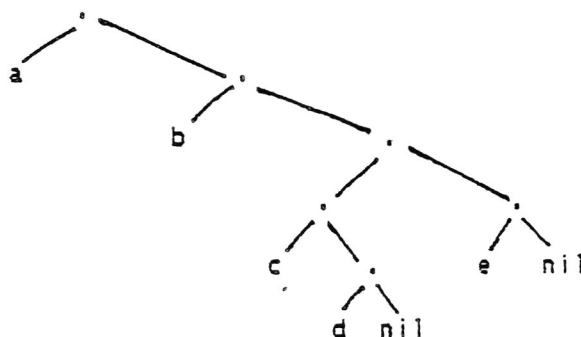
La fonction Lisp de base pour CONStruire une liste - ou plutôt une "paire pointée" - n'était autre que CONS, qui prenait deux arguments x et y, et retournait la paire pointée (x . y), dont le CAR est x et le CDR y (non numérique). Dans le cas usuel où y est une liste, comme (a b c), on note alors:

$$(\text{cons } x \ y) = (x \ . \ y) = (x \ a \ b \ c)$$

en oubliant les effets dus à l'évaluation des arguments. Rappelons-nous enfin (cf. I.1) qu'une liste telle que:

(a b (c d) e)

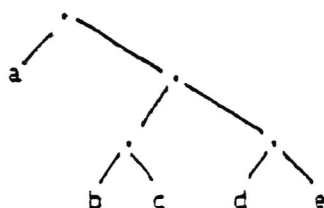
dénote en fait un arbre binaire dont les feuilles sont des atomes:



tandis que la paire pointée:

(a (b . c) d . e)

dénote elle l'arbre:



PROLOG, quant à lui, utilise aussi - et même uniquement - la notation en paire pointée pour représenter une liste:

LISP

(x . y)
(a 1 b 2)
(+ (* 2 3) 4)
(a (b . c) . d)

PROLOG

x.y
a.1.b.2.nil
+.(*.2.3.nil).4.nil
a.(b.c).d

N.B Une différence importante avec Lisp concerne la syntaxe des variables (cf. II.2.3). Si nous écrivons x.y sous PROLOG, ce dernier lira des variables et tentera des unifications non souhaitées. Les constantes (identifica-

teurs) débutent en effet par deux lettres. Ecrivons donc `xx.yy`, ou bien `"x".y`, ou encore limitons-nous aux listes numériques pour traiter des exemples. C'est cette dernière option que nous avons prise, pour de seules raisons de lisibilité. Afin de garder en vue les listes usuelles, nous n'utiliserons pas non plus de "vraie" paire pointée (un CDR ne sera jamais un atome <> NIL).

Cette représentation rend évidents les effacements de CONS, CAR et CDR:

```
cons(x,y,x.y) ->;
car(x.y,x)     ->;      "le CAR de x.y est x"
cdr(x.y,y)     ->;      "le CDR de x.y est y"
```

Notez que CAR et CDR, indispensables en Lisp pur (les liaisons de `Le_Lisp` permettent souvent de s'en passer), ne seront JAMAIS utilisées dans les exemples qui vont suivre, l'unification se chargeant de "séparer" une liste en CAR et CDR, précisément en la notant `x.y` au lieu de `L`. D'ailleurs, voici une autre manière de les effacer sans liaison:

```
car(x,y) -> cons(y,z,x);    "le CAR de x est y si x = y.z"
cdr(x,y) -> cons(z,y,x);
```

Nos seuls "atomes" seront NIL et les entiers:

```
atom(nil) ->;
atom(e)   -> entier(e);
```

en utilisant le prédicat prédéfini ENTIER (ciao Giuseppe).

```
>cons(1,2.1.3.4.nil,x);
x=1.2.1.3.4.nil
>cons(x,y,1.2.3.nil);
x=1 y=2.3.nil
>
```

Le dernier test montre bien l'extraction du CAR et du CDR dont il était question ci-dessus.

Nous sommes dès lors en mesure de rétablir l'impression usuelle d'une liste LISP (encore une fois sans paire pointée):

```
print(e)  -> atom(e) ex(e);
print(x.y) -> exm("(") contenu(x.y) exm(")");

contenu(nil) ->;
contenu(x.nil) -> print(x);
contenu(x.y)  -> dif(y,nil) print(x) exm(" ") contenu(y);
```

Nous ne disons rien d'autre que ceci: un atome s'écrit sans autre forme de procès (le prédicat EX s'efface en écrivant la valeur d'une variable); une liste est constituée d'une parenthèse ouvrante, d'un contenu de liste, d'une parenthèse fermante. Un contenu de liste peut être vide, réduit à une expression (et alors pas de blanc avant la fermante) ou composé d'une expression et d'un contenu de liste (et alors édition d'un blanc après l'expression). Quant au prédicat LIGNE, il s'efface en éditant un saut vers le début de la ligne suivante:

```
>print(1.2.3.nil) ligne;
(1 2 3)
>print(1.(2.3.nil).4.5.nil);
(1 (2 3) 4 5)
```


Voyons maintenant comment l'on peut simuler en PROLOG les fonctions Lisp suivantes:

APPEND MEMBER FLAT REMOVE SUBST

APPE D

Quelle relation doit-il exister entre les listes x,y et z pour que:

append(x,y,z)

soit vérifié, i.e. (equal z (append x y)) ? Une chose est sûre: x et z doivent avoir meme CAR, donc autant écrire tout de suite x sous la forme u.x et z sous la forme u.z. Quand aura-t-on alors:

append(u.x,y,u.z) ?

Facile: u.z est la concaténation de u.x et y à condition que z soit celle de x et y. Mais peut-on toujours écrire u.x ? Oui, à condition que ce soit une liste non vide. L'effacement de APPEND s'en déduit:

```
append(nil,y,y)    ->;                                -1-
append(u.x,y,u.z) -> append(x,y,z);                    -2-
```

Ce prédicat est totalement inversible:

```
>append(1.2.nil,3.4.5.nil,x);
x=1.2.3.4.5.nil
>append(x,3.4.nil,1.2.3.4.nil);
x=1.2.nil
>append(x,y,1.2.3.nil);
x=nil y=1.2.3.nil
x=1.nil y=2.3.nil
x=1.2.nil y=3.nil
x=1.2.3.nil y=nil
>
```

MEMBER

member(x,l) s'efface si et seulement si x est élément (au premier niveau) de la liste l. Si l=nil l'effacement n'a pas lieu (on n'en parle donc pas, grace aux vertus de la négation par l'échec en PROLOG); sinon l s'écrit plutôt y.z, et deux cas se présenteront suivant ou non que y=x:

```
member(x,x.z) ->;                                -1-
member(x,y.z) -> member(x,z);                    -2-
```

En clair: ou bien x est le CAR de l, ou bien x doit figurer dans le CDR de l. Tout se passe à peu près bien, sauf que:

```
>member(1,2.1.3.1.nil) exm("oui");
oui
oui
>
```

Fidèle à ses principes non-déterministes, PROLOG a prouvé deux fois que 1 figurait bien dans la liste 2.1.3.1.nil ! Là encore, une fois le gain obtenu sur la première clause, il fallait empêcher l'appel récursif de la seconde, et couper court la recherche:

- 83 -

```
member(x,x.z) ->; -1'-
member(x,y.z) -> dif(x,y) member(x,z); -2'-
```

et alors:

```
>member(1,2.1.3.1.nil) exm("oui");
oui
>member(x,1.(2.3.nil).4.nil);
x=1
x=2.3.nil
x=4
>
```

Nous vous laissons écrire le "vrai" MEMBER à la Le_Lisp:

```
member(x,l,m)
```

s'effaçant si m est le "premier CDR" de la liste l commençant par x (donc nil si x ne figure pas dans l).

FLAT

Il s'agit "d'aplatir" une liste l à tous les niveaux, de retourner la liste des atomes figurant dans le feuillage de l:

```
>flat(1.(2.3.(1.nil).5.nil).6.7.nil,x);
x=1.2.3.1.5.6.7.nil
>
```

Un atome est tout aplati:

```
flat(1,1) -> atom(1); -1-
```

sinon l est une liste, notée x.y, mais x est ou n'est pas déjà aplati:

```
flat(x.y,x.z) -> atom(x) flat(y,z); -2-
```

Pour exprimer enfin que x n'est pas aplati, écrivons-le u.v (c'est la meilleure façon en PROLOG d'exprimer qu'il ne s'agit pas d'un atome):

```
flat((u.v).y,z) -> flat(u,v,x1) flat(y,y1) append(x1,y1,z); -3-
```

REMOVE

Comme son nom l'indique, remove(e,u,v) s'efface si la liste v s'obtient à partir de la liste u par suppression de toutes les occurrences de l'élément e (au premier niveau). Comme d'habitude, nous coupons la liste u en deux (si ce n'est pas nil), et l'écrivons sous la forme x.y, ce qui pose la question: est-ce que x=e ?

```
remove(e,nil,nil) ->; -1-
remove(e,e.y,z) -> remove(e,y,z); -2-
remove(e,x.y,x.z) -> dif(e,x) remove(e,y,z); -3-
```

On peut alors tester:

```
>remove(1,2.1.3.4.1.nil,x);
x=2.3.4.nil
>remove(x,2.1.3.4.1.nil,2.3.4.nil);
x=1
>
```

N.B. Il est possible d'oter un élément x figurant dans une liste l en une seule clause, en exprimant - comme lors d'un filtrage en Lisp - qu'il est possible de scinder la liste l en deux sous-listes, celle précédant x et celle qui le suit (une scission de liste s'exprime avec APPEND). Nous vous laissons l'exprimer en PROLOG. Quelle est la différence entre ce nouveau prédicat ENLEVER et l'ancien REMOVE ? Comparez par exemple les appels:

enlever(x,1.2.3.nil,z) et remove(x,1.2.3.nil,z)

SUBST

L'obtention de la liste y à-partir de la liste x par substitution de e à toutes les occurrences de l'élément $e1$ (premier niveau), est la condition d'effacement de:

subst(e,e1,x,y)

Encore et toujours la meme technique de séparation en CAR et CDR. Tout d'abord:

subst(e,e1,nil,nil) ->; -1-

Si $x=e1.u$, alors il est nécessaire que $y=e.v$ et dans ce cas le littéral:

subst(e,e1,e1.u,e.v)

s'effacera dès que l'on pourra garantir que la substitution s'est bien propagée dans le reste:

subst(e,e1,e1.u,e.v) -> subst(e,e1,u,v); -2-

Si enfin la liste x ne commence pas par $e1$, on préserve son CAR:

subst(e,e1,e2.u,e2.v) -> dif(e1,e2) subst(e,e1,u,v); -3-

Essayons:

```
>subst(10,1,2.1.3.(4.1.nil).5.1.nil,x);
x=2.10.3.(4.1.nil).5.10.nil
>subst(x,1,2.1.3.(4.1.nil).5.1.nil,2.10.3.(4.1.nil).5.10.nil);
x=10
>
```

Permutations

Dans une remarque concernant l'écriture de REMOVE, nous avons évoqué la possibilité de l'écrire en une seule clause, à condition que l'élément à effacer figure effectivement dans la liste. Vous avez sans doute écrit ceci:

enlever(x,l,r) -> append(p,x.q,l) append(p,q,r) -1-

Intéressons-nous maintenant aux manières de permuter les éléments d'une liste l , par exemple 1.2.3.nil. Ecrivons:

perm(u,v)

pour indiquer que la liste u est une permutation de la liste v . A quelle condition ceci est-il vrai ? Evidemment si $u=v=nil$:

perm(nil,nil) ->; -2-

Si d'autre part u s'écrit $x.y$, quand a-t-on perm(x.y,z) ? Il faut que x figu-

re dans z , mais ce n'est pas suffisant; il faut aussi qu'une fois que x ait été enlevé de z , alors y soit une permutation du reste:

$\text{perm}(x.y,z) \rightarrow \text{enlever}(x,z,z1) \text{ perm}(y,z1);$ -3-

Bien. Voici un programme de permutation qui "tourne". Mais qui tourne dans quel sens ? Comment faire l'appel:

$\text{perm}(1.2.\text{nil},z)$ ou $\text{perm}(z,1.2.\text{nil})$?

Il se trouve que l'un d'eux est correct, l'autre pas ! Ce type de problème, qui nous a été indiqué par J.F. Perrot, demande de bien savoir quelles sont les variables devant être instanciées.

Le maximum d'instanciation semble avoir lieu en instanciant à gauche, i.e. $x.y$. Fournissons donc $x.y$ (e.g. $1.2.\text{nil}$) dans la clause -3-; que va-t-il se passer lors de l'effacement de:

$\text{enlever}(1,z,z1)$ (*)

i.e. de:

$\text{append}(p,1.q,z) \text{ append}(p,q,z1)$?

L'effacement de cette liste de buts aura bien lieu, sous certaines "contraintes" (pour la notion de contrainte, cf. l'article de TSI cité plus bas), comme par exemple:

$\{z=1.z1\}$

ou: $\{z=x14.1.x12, z1=x14.x12\}$

ou encore: $\{z=x14.x18.1.x12, z1=x14.x18.x12\}$

etc., ces contraintes donnant en quelque sorte les solutions paramétrées de l'identité (*). Elles vont d'ores et déjà occasionner un back-track infini sur le premier littéral à effacer dans -3- ! Allons plus loin. Au cours de ce back-track, allons-nous trouver toutes les solutions ? Dressons à cet effet l'arbre d'effacement du but $\text{perm}(1.2.\text{nil},z)$, cf. figure ci-contre. Nous nous apercevons qu'après avoir trouvé la solution $z=1.2.\text{nil}$, PROLOG (alias le programmeur) se perd dans un back-track infini au niveau de branches trop basses pour pouvoir récupérer les autres solutions.

En fait, le bon appel est:

```
>perm(p,1.2.nil);
p=1.2.nil
p=2.1.nil
>
```

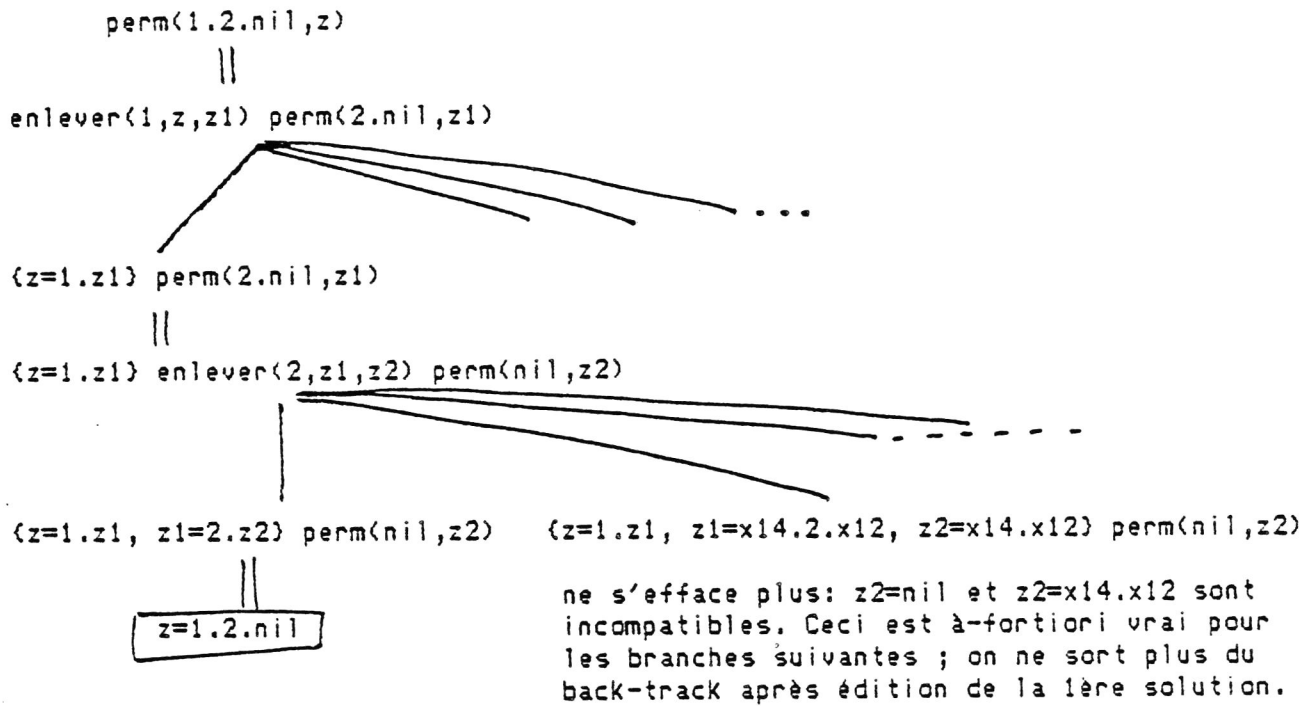
et nous laissons dresser l'arbre correspondant. Notez que l'appel devient:

$\{p=x.y\} \text{ enlever}(x,1.2.\text{nil},z1) \text{ perm}(y,z1)$

et cette fois-ci, l'effacement de $\text{enlever}(x,1.2.\text{nil},z1)$ est fini (il se fait en deux temps, la liste $1.2.\text{nil}$ étant de longueur 2):

```
>enlever(x,1.2.nil,z1);
x=1 z1=2.nil
x=2 z1=1.nil
>
```

et une récurrence sur la correction du programme PERM peut avoir lieu. Ne vous



----- l'arbre d'effacement de $\text{PERM}(1.2.\text{NIL}, X)$ -----

reposez pas, une dernière question: que se passe-t-il si l'on utilise REMOVE au lieu d'ENLEVER ?...

Voici une autre façon de décrire les permutations des entiers de 1 à 4, donnée dans le manuel de référence PROLOG II (non, ne le prenez pas...) : pour fabriquer une permutation de 1..N, on exprime que ce doit être une liste de longueur 4 - i.e. on l'écrit sous la forme x.y.z.t.nil - constituée de nombres de 1 à 4 tous différents. PROLOG se chargera du travail combinatoire. A vous...

N.B. Avant de terminer avec les listes, il existe une commande de l'éditeur qui est x (exécuter) permettant de demander l'effacement d'une liste de buts en restant sous éditeur (la commande f est lente). On entre un but ou une paire pointée de buts, sans oublier d'exiger les impressions souhaitées:

```
+x append(x,y,1.2.3.nil).exm("x = ").ex(x).ligne
x = nil
x = 1.nil
x = 1.2.nil
x = 1.2.3.nil
```

+

Un exemple intéressant d'application des listes, et du prédicat APPEND, est le programme MUTANTS, qui figure dans le manuel d'exemples PROLOG II (et sur la disquette PRO-EX:). Tout programme de tri exigeant des comparaisons, nous les aborderons dans la section suivante sur les prédicats "évaluables".

II.3.4

Les prédicats évaluables

Il s'agit de prédicats débordant du cadre strict de la Logique, mais qui font de PROLOG un langage de programmation utilisable. Ceci dans deux directions:

- l'arithmétique entière et booléenne, par le biais du prédicat VAL et des "fonctions évaluables":

ADD SUB MUL DIV MOD INF EQ SI

- le contrôle, avec le fameux prédicat / (le "slash" ou "cut"). Dans le cadre de ce cours, nous ne l'avons jamais utilisé, utilisant partout DIF pour écrire des clauses disjointes. Malgré tout, il s'agit d'un outil efficace, par exemple lorsqu'on entre dynamiquement des clauses en cours d'exécution d'un programme (ce dont nous n'avons pas non plus parlé), ou tout simplement lorsqu'on préfère un comportement plus déterministe. Nous vous laissons donc l'étudier si vous désirez aller plus loin avec PROLOG.

Rappelons que PROLOG II (version 1.1) n'utilise que les entiers positifs ou nuls. Il est possible de faire des affectations comme dans un langage classique, mais nous n'en parlerons pas; ceci reste trop "professionnel" et étranger à la programmation en Logique, qui consiste plutôt à définir des relations entre des objets.

Nous n'utiliserons le prédicat VAL que dans le sens suivant:

```
-----
VAL(t1,t2) s'efface si:    1) t1 est un "terme évaluable"
                           2) t2 = valeur(t1) si t2 est un nombre
                              ou
                           t2 := valeur(t1) si t2 est une variable
-----
```

Un terme évaluable consiste en l'application d'une fonction composée des fonctions élémentaires suivantes, toutes binaires sauf la dernière:

add(x,y) sub(x,y) mul(x,y) div(x,y) mod(x,y)
 inf(x,y) eq(x,y) si(x,y,z)

Voici des exemples de termes évaluables arithmétiques:

mod(add(3,4),2) dont la valeur est 1 (7 modulo 2)
 div(add(9,5),sub(9,6)) dont la valeur est 4 (14 div 3)

Il est impératif que tous les arguments d'un terme évaluable soient instanciés au moment de l'évaluation de ce terme:

```
>val(add(2,3),x);
x=5
>val(add(2,3),x) val(sub(x,1),y);
x=5 y=4
>val(mul(2,3),5);
>
```

Un terme évaluable ne peut figurer qu'en premier argument de VAL, et pas ailleurs; ce ne sont donc pas des fonctions au sens usuel (et ceci, au passage, peut être regretté; pourquoi pas des "macros" aussi en PROLOG...):

```

>ex(3);
3
>ex(add(2,3));
add(2,3)
)

```

Les fonctions INF (inférieur-strict, fonctionnant aussi avec les chaînes de caractères) et EQ (égalité numérique) sont "booléennes" et retournent 0 (faux) ou 1 (vrai):

```

>val<inf(2,3),x>;
x=1
>val<add<inf(2,3),inf(3,4)>,2> exm("ok");
ok
>val<eq<add(2,3),5>,x>;
x=1
)

```

On voit sur le dernier test la possibilité de jouer sur les valeurs 0 et 1 pour effectuer des ET-OU (opérations booléennes). Voici un moyen d'établir les tables de vérité de ET et de OU qui sont duales (en identifiant les opérateurs ET et OU sur l'ensemble F,V aux opérations MIN et MAX sur l'ensemble 0,1):

"table de verite de ET et de OU"

```

et(x,y,z) -> acceptables(x,y,z) min(x,y,z);          -1-
ou(x,y,z) -> acceptables(x,y,z) max(x,y,z);          -2-
acceptables(x,y,z) -> valeur(x) valeur(y) valeur(z); -3-
max(x,y,y) -> val<inf(y,x),0>;                        -4-
max(x,y,x) -> val<inf(y,x),1>;                        -5-
valeur(0) ->;                                           -6-
valeur(1) ->;                                           -7-

```

Nous avons oublié MIN. Que dites-vous de ceci pour s'entraîner sur ces prédicats (et tant pis pour l'efficacité...):

```

min(x,y,z) -> val<add(x,y),s> max(x,y,m) val<sub(s,m),z>; -8-

```

Une petite présentation pour le test:

```

montrer(x,y,z,t) -> ex(x) exm(" ") exm(t) exm(" ")    -9-
                  ex(y) exm(" : ") ex(z);
essai-et -> et(x,y,z) montrer(x,y,z,"et");            -10-
essai-ou -> ou(x,y,z) montrer(x,y,z,"ou");             -11-

```

```

>essai-et;
0 et 0 : 0
0 et 1 : 0
1 et 0 : 0
1 et 1 : 1
>essai-ou;
0 ou 0 : 0
0 ou 1 : 1
1 ou 0 : 1
1 ou 1 : 1
)

```


Dans $\text{si}(t, t_1, t_2)$, qui est le IF-THEN-ELSE des nostalgiques, le résultat rendu est la valeur de t_1 si le "test" t (un booléen) évalue à 1, et celle de t_2 s'il évalue à 0 :

```
>val(si<inf<3,2>,4,5),x);
x=5
>
```

Quant aux opérations arithmétiques, il nous est loisible d'utiliser maintenant à fond les fonctions évaluables. Voici la classique factorielle, dans laquelle $\text{fac}(x,y)$ s'efface si $y = x!$:

```
fac(0,1) ->; -1-
fac(x,y) -> val<inf<0,x>,1> val<sub(x,1),x1> -2-
           fac(x1,y1) val<mul(y1,x),y>;
```

en clair: $x! = \text{si } x=0 \text{ alors } 1$
 sinon si $x>0$ alors $(x - 1)! * x$
 sinon rien.

La présence des prédicats évaluables empêche en général l'inversibilité des programmes PROLOG. La requête $\text{fac}(x,24)$ n'aboutit pas, puisque dans -2- la variable x n'est pas instanciée, ce qui est incorrect pour $\text{val}<\text{inf}<0,x>,1>$.

Nous aurons plus loin l'occasion d'utiliser ces prédicats évaluables.

N.B. La prochaine version devrait accepter les nombres "réels".

II.3.5 - Quelques thèmes d'étude en PROLOG -

II.3.5.1 Tri par insertion

Reprenons en PROLOG le tri dont il a été question en Lisp (cf I.1). Il s'agissait de trier récursivement une liste en insérant son premier élément dans le reste de la liste, supposé trié:

```

-----
"tri par insertion"

tri(nil,nil) ->;                                -1-
tri(x.y,z)  -> tri(y,y1) insertion(x,y1,z);      -2-

insertion(x,nil,x.nil) ->;                      -3-
insertion(x,y.z,y.z1) -> val<inf(y,x),0> insertion(x,z,z1); -4-
insertion(x,y.z,x.z1) -> val<inf(x,y),1> insertion(y,z,z1); -5-
-----

```

et alors:

```

>tri(3.2.5.3.4.2.7.1.nil,x);
x=1.2.2.3.3.4.5.7.nil
>tri("livre"."libres"."lisp".nil,x);
x="libres"."lisp"."livre".nil
>

```

Nous vous laissons développer un "tri-fusion" dont le principe est le suivant: scinder la liste en deux sous-listes que l'on trie séparément (par le même procédé), puis fusionner les deux listes triées:

```

-----
tri-fusion(l,l1) -> scission(l,x,y)
                  tri-fusion(x,x1)
                  tri-fusion(y,y1)
                  fusion(x1,y1,l1);
-----

```

Vous noterez au passage, dans les programmes qui ont été développés, le souci de programmer de manière descendante, en "retardant" l'écriture des sous-programmes. Cet encouragement à un style de programmation éminemment modulaire est aussi un bon atout pédagogique de PROLOG.

II.3.5.2 Le crible d'Eratosthène

Nous avons écrit avec les élèves des prédicats concernant les nombres premiers, en particulier le crible d'Eratosthène. Une fois l'idée du crible comprise (désolé, l'Arithmétique ne figure plus au programme, ce qui, tout-à-fait entre nous, est désolant...), la plupart des groupes n'ont pas eu de difficulté à l'écrire en PROLOG. Le résultat de leurs cogitations a donné à peu près ceci:

```

-----
"le crible d'Eratosthene: liste des nombres premiers de 2 à n"

premiers(n,p) -> entiers(2,n,1) crible(1,p)                                -1-

entiers(n,m,nil) -> val<inf(m,n),1>;                                     -2-
entiers(n,n,n,nil) ->;                                                 -3-
entiers(n,m,n,r) -> val<inf(n,m),1>                                     -4-
                    val<add(n,1),n1>
                    entiers(n1,m,r);

crible(nil,nil) ->;                                                    -5-
crible(p,r,p,1) -> eliminer(p,r,r1) crible(r1,1);                     -6-

eliminer(p,nil,nil) ->;                                               -7-
eliminer(p,m,r,r1) -> val<mod(m,p),0> eliminer(p,r,r1);              -8-
eliminer(p,m,r,m,r1) -> val<eq(mod(m,p),0),0>                          -9-
                    eliminer(p,r,r1);
-----

```

Donnant n , le prédicat PREMIERS (clause -1-) calcule la liste p des nombres premiers inférieurs ou égaux à n :

```

>premiers(10,p);
p=2.3.5.7.nil
>

```

Pour cela, il commence (clauses -2- à -4-) par dresser la liste des entiers de 2 à n :

```

>entiers(2,10,1);
l=2.3.4.5.6.7.8.9.10.nil
>

```

puis (clauses -5- et -6-) opère le crible sur cette liste:

```

>crible(2.3.4.5.6.7.8.9.10.nil,p);
p=2.3.5.7.nil
>

```

Rappelons la méthode de ce crible: il s'agit d'éliminer tous les nombres non premiers de la manière suivante: la tête de la liste courante est un nombre premier, on le garde et l'on élimine tous ses multiples, et on refait pareil avec le premier entier non éliminé, jusqu'à se trouver en queue de liste. Pour éliminer tous les multiples de p dans une liste r pour obtenir une liste $r1$, on efface "eliminer(p,r,r1)" (clauses -7- à -9-):

```

>eliminer(3,2.3.4.5.6.7.8.9.10.nil,r1);
r1=2.4.5.7.8.10.nil
>

```

Enfin, les littéraux " $\text{val}(\text{mod}(m,p),0)$ " et " $\text{val}(\text{eq}(\text{mod}(m,p),0),0)$ " signifient respectivement " m est multiple de p " et " m n'est pas multiple de p ".

A cause des accès-disque constants, le temps est vite prohibitif: plus de 2 minutes pour la liste des premiers inférieurs à 40. Mais dans de telles conditions, ce facteur n'est pas très significatif; attendons les versions sur de plus gros processeurs (Motorola 68000) avec (une bonne partie de) la base de règles résidente en mémoire centrale.

II.3.5.3 Un (tout petit) voyageur de commerce

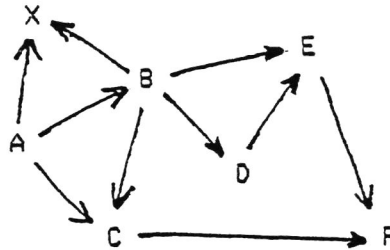
Des villes A, B, C, ... sont reliées par des routes à sens unique. Pour exprimer que la ville A est reliée à la ville E, nous déclarerons:

```
arc(va,ve) ->;
```

La notation VA au lieu de A est due à la syntaxe des variables. Il était aussi possible de les noter par des chaînes de caractères "A", "B", ...

Voici la base des faits:

arc(va,vb) ->;	-1-
arc(va,vc) ->;	-2-
arc(va,vx) ->;	-3-
arc(vb,vc) ->;	-4-
arc(vb,vd) ->;	-5-
arc(vb,ve) ->;	-6-
arc(vb,vx) ->;	-7-
arc(vc,vf) ->;	-8-
arc(vd,ve) ->;	-9-
arc(ve,vf) ->;	-10-



Nous nous proposons de savoir s'il est possible de relier une ville à une autre par une suite de villes intermédiaires. On peut par exemple aller de A vers F en passant par C, ou par B,D,et E etc. L'effacement de:

```
connect(x,y,l)
```

aura lieu si x et y sont des villes, et s'il existe une liste l formée de villes constituant les étapes intermédiaires pour se rendre de x à y:

```

>connect(va,vf,x);
x=vb.vc.nil
x=vb.vd.ve.nil
x=vb.ve.nil
x=vc.nil
>

```

Il est analogue aux relations d'ascendance familiale (la relation est directe ou nécessite un transit):

connect(x,y,nil) -> arc(x,y);	-11-
connect(x,y,z,r) -> arc(x,z) connect(z,y,r);	-12-

Peut-on aller de X à D ?

```

>connect(vx,vd,x);
>

```

Nenni. Comment se rendre de B à E ?

```

>connect(vb,ve,x);
x=nil
x=vd.nil
>

```

La réponse `x=nil` n'est pas négative: elle indique simplement l'absence de ville intermédiaire, et donc un voyage direct. Notez enfin qu'une ville n'est pas reliée à elle-même, dans l'optique prise:

```
>connect(va,va,x);
>
```

Quels sont les chemins aboutissant à F ?

```
>connect(v,vf,x);
v=vc x=nil
v=ve x=nil
v=va x=vb.vc.nil
v=va x=vb.vd.ve.nil
v=va x=vb.ve.nil
v=va x=vc.nil
v=vb x=vc.nil
v=vb x=vd.ve.nil
v=vb x=ve.nil
v=vd x=ve.nil
>
```

Nous pouvons utiliser les prédicats arithmétiques pour introduire les distances (ou "couts") entre villes reliées:

```
arc-d(va,vb,7) -> -1'-
arc-d(va,vc,5) -> -2'-
....
arc-d(ve,vf,5) -> -10'-
```

et maintenant "`connect-d(x,y,l,n)`" s'effacera si de plus `n` est le cout total du trajet reliant `x` à `y`:

```
-----
connect-d(x,y,nil,n) -> arc-d(x,y,n); -11'-
connect-d(x,y,z.r,n) -> arc-d(x,z,n1) -12'-
                        connect-d(z,y,r,n2)
                        val(add(n1,n2),n);
-----
```

Quels sont les chemins de A à F passant par D, et leurs couts ?

```
>connect-d(va,vf,x,n) member(vd,x);
x=vb.vd.ve.nil n=22
>
```

(MEMBER a été vu en II.3.3).

Quels sont les chemins issus de A de cout au plus 17 ?

```
>connect-d(va,v,x,n) val(inf(17,n),0);
v=vb x=nil n=7
v=vc x=nil n=5
v=vx x=nil n=4
v=vc x=vb.nil n=12
v=vd x=vb.nil n=15
v=vx x=vb.nil n=12
v=ve x=vb.vd.nil n=17
>
```

Quels sont les chemins allant de A vers F et ne passant pas par D ? Une première solution consiste à écrire un utilitaire HORS-DE tel que:

hors-de(x,l)

s'efface si et seulement si x n'est pas élément de la liste l. C'est facile à faire (...), et produirait la question:

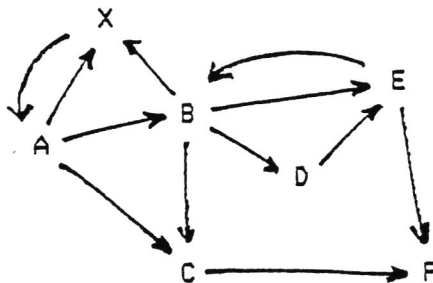
```
>connect-d(va,vf,x,n) hors-de(vd,x);
x=vb.vc.nil n=24
>
```

mais aurait en contre partie l'inconvénient de calculer tout un chemin, même s'il passe par D, au lieu d'arrêter dès que D est rencontré. Trop aveugle... Autant ré-écrire le prédicat CONNECT-D:

```
-----
chemin-sans-d(x,y,nil,n) -> arc-d(x,y,n);
chemin-sans-d(x,y,p.r,n) -> arc-d(x,p,n1)
                           dif(p,vd)
                           chemin-sans-d(p,y,r,n2)
                           val(add(n1,n2),n);
-----
```

qui, lui, teste avant de faire le plus petit pas que ce n'est pas vers D ! Bien entendu, D pourrait être passé en paramètre.

Certaines configurations peuvent présenter des "boucles" comme:



dans laquelle on reconnaît le graphe engendré par les clauses -1- ... -10-, augmenté des faits:

```
arc(vb,vb) ->;
arc(vx,va) ->;
```

qui induisent l'apparition de deux "boucles" ABXA et BEDB. Pour éviter de tomber dans ces pièges (fâcheux avec un programme qui back-track...), on va "interdire", à chaque étape, les villes déjà visitées:

```
-----
connect-sans-boucle(x,y,nil,l) -> arc(x,y) hors-de(y,l);      -11''-
connect-sans-boucle(x,y,z.l,l1) -> arc(x,z)                  -12''-
                                hors-de(z,l1)
                                connect-sans-boucle(z,y,l,z.l1);
-----
```

L'effacement de "connect-sans-boucle(x,y,l,l1)" a lieu si l'on peut relier x à y avec une liste l d'arcs intermédiaires, en s'interdisant de passer par une ville élément de l1:

```

>connect-sans-boucle(va,vf,l,va.nil);
l=vb.vc.nil
l=vb.vd.ve.nil
l=vb.ve.nil
l=vc.nil
>

```

D'autres stratégies sont possibles pour éviter les boucles dans un parcours dans un "graphe" (cf. par exemple l'ouvrage de Clark & McCabe cité plus bas, pp. 341-364).

Un exemple de "cryptarithmétique"

Bien que présent dans le manuel d'exemples de PROLOG II, nous présenterons ce très bel exemple. Il s'agit de résoudre le puzzle arithmétique suivant, dans lequel chaque lettre représente un chiffre distinct de 0 à 9. Y-a-t-il une solution ? Plusieurs ?

```

      S E N D
    + M O R E
    -----
    M O N E Y

```

La recherche à la main est possible mais fastidieuse, et requiert de la méthode. En PROLOG, nous allons plutôt "spécifier" les données du problème, au lieu d'en décrire une solution détaillée. En particulier, la façon dont nous exprimerons en PROLOG II le fait que deux lettres distinctes représentent deux chiffres distincts se fera d'une manière tout-à-fait originale par-rapport à un langage classique, par le biais d'une contrainte qui se propagera pendant un certain temps, pour "éclater" ensuite à retardement. Ce mécanisme d'évaluation retardée, possible ici grâce à DIF, ne se trouve pas dans toutes les versions de PROLOG (lire par exemple le rapport de L. Naish cité plus bas).

Spécifier ce problème, c'est essentiellement exprimer qu'il faut effectuer une addition avec retenue. Appellons r_1 , r_2 , r_3 et r_4 les retenues (négligeons même le fait que $r_1 = M$ par exemple):

```

    r1 r2 r3 r4
      S E N D
    + M O R E
    -----
    M O N E Y

```

Nous déclarerons simplement que:

```

      D + E = Y et je retiens r4
    r4 + N + R = E et je retiens r3 etc...

```

et exprimons que " $r + c_1 + c_2 = c$ et je retiens r_1 " si et seulement s'il y a effacement de:

```

somme-avec-retenue(r,c1,c2,c,r1)

```

Notre programme s'écrira alors:

- 97 -

```

resolution(s.e.n.d,m.o.r.e,m.o.n.e.y) ->          -1-
  tous-distincts(s.e.n.d.m.o.r.y.nil)
  dif(s,0) dif(m,0)
  somme-avec-retenue(r1,0,0,m,0)
  somme-avec-retenue(r2,s,m,o,r1)
  somme-avec-retenue(r3,e,o,n,r2)
  somme-avec-retenue(r4,n,r,e,r3)
  somme-avec-retenue( 0,d,e,y,r4)

```

Le littéral "tous-distincts(l)" s'effacera si tous les éléments de la liste l sont distincts:

```

tous-distincts(nil) ->;          -2-
tous-distincts(x.r) -> hors-de(x,r) tous-distincts(r); -3-

```

avec, pour mémoire:

```

hors-de(x,nil) ->;          -4-
hors-de(x,y.r) -> dif(x,y) hors-de(x,r); -5-

```

L'effacement de "tous-distincts(s.e.n.d.m.o.r.y.nil)" aura donc lieu parallèlement à la mise en place de tous les "dif(s,e)", "dif(d,r)", ... qui seront retardés tant que les deux arguments du DIF ne sont pas "pris". Qu'est-ce qu'une addition avec retenue ?

```

somme-avec-retenue(r,c1,c2,c,r1) ->          -6-
  retenue(r)
  chiffre(c1)
  chiffre(c2)
  val(add(r,add(c1,c2)),s)
  val(div(s,10),r1)
  val(mod(s,10),c);

```

En effet, il faut que r soit une retenue(0 ou 1), c1 et c2 des chiffres (il y aura donc back-track sur r, c1 et c2), et l'on exprime que dans l'expression "c et je retiens r1", c est le chiffre des unités et r1 celui des dizaines, qui sont calculés par MOD et DIV respectivement. Enfin:

```

retenue(0) ->;          -7-
retenue(1) ->;          -8-

```

et:

```

chiffre(0) ->;          -9-
chiffre(1) ->;          -10-
.....
chiffre(9) ->;          -18-

```

La solution est obtenue en un peu plus de 4 minutes, avec deux minutes en plus pour en vérifier l'unicité:

```

>resolution(x,y,z);
x=9.5.6.7 y=1.0.8.5 z=1.0.6.5.2
>

```

Très joli, mais ne vous emballez pas, et essayez plutôt les suivantes:

LYNDON * B = JOHNSON ou FORTY + TEN + TEN = FIFTY ...

Il est écrit quelque part dans la documentation que le temps est extensible, pas l'espace. Nous avons pu le constater !

Approche grammaticale du langage naturel

Il ne s'agit pas, bien sur, de décrire une grammaire très compliquée, mais de faire travailler les élèves sur le tout petit sous-ensemble du français suivant, qui reste néanmoins extensible:

```

<phrase> := <groupe-nominal> <groupe-verbal>
<groupe-nominal> := <article> <nom>
<groupe-verbal> := <verbe> / <verbe> <groupe-nominal>
<article> := le / la / un / ...
<nom> := chien / chat / viande / ...
<verbe> := mange / avale / ...

```

Cette grammaire n'est pas récursive, mais on peut la rendre telle, en introduisant les subordonnées ou les conjonctions par exemple.

Voici quelques exemples de phrases reconnues par cette grammaire:

```

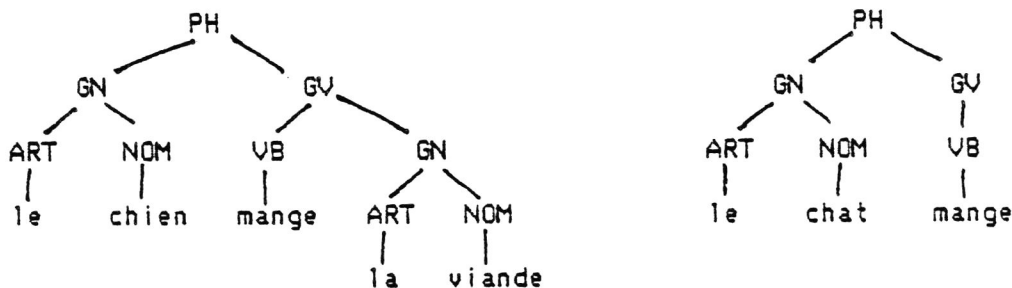
"le chat mange"
"le chien avale la viande"
"le souris grignotte la chien"   etc...

```

Il n'y a donc pas de règle d'accord, ceci fait partie des extensions. Notre but est le suivant: à-partir de phrases comme:

"le chien avale la viande" ou "le chat mange"

nous voulons obtenir l'arbre syntaxique associé, représentant en quelque sorte la structure de la phrase selon la grammaire précédente:



Une phrase sera tout naturellement représentée par une liste:

le.chien.avale.la.viande.nil ou le.chat.mange.nil

Nous allons exprimer qu'une phrase est grammaticalement correcte s'il est possible de la scinder en groupe nominal g1 et un groupe verbal g2. Son arbre associé aura alors la forme (PH d' d'') où d' et d'' sont les sous-arbres de ses groupes composants:

```

phrase(p,PH.d'.d''.nil) -> append(g1,g2,p)
                           gr-nominal(g1,d')
                           gr-verbal(g2,d'');

```

-1-

Le meme principe s'utilise pour décortiquer les deux groupes:

```

gr-nominal(g,GN.d'.d''.nil) -> append(a,n,g)
                              article(a,d')
                              nom(n,d'');

```

-2-

en pensant que le groupe verbal peut avoir deux formes:

```

gr-verbal(v.nil,GV.d.nil)    -> verbe(v.nil,d);           -3-
gr-verbal(v.r,GV.d'.d''.nil) -> dif(r,nil)                -4-
                                append(g1,g2,v.r)
                                verbe(g1,d')
                                gr-nominal(g2,d'');

```

et enfin la classification des mots:

```

article(a.nil,ART.a.nil) -> mot(a,ART);                    -5-
nom(n.nil,NOM.n.nil)     -> mot(n,NOM);                    -6-
verbe(v.nil,VB.v.nil)    -> mot(v,VB);                    -7-

```

et le dictionnaire:

```

mot(le,ART)      ->;                                       -8-
mot(la,ART)      ->;                                       .
mot(un,ART)      ->;                                       .
mot(une,ART)     ->;                                       .
mot(chien,NOM)   ->;
mot(chat,NOM)    ->;
mot(chatte,NOM)  ->;
mot(souris,NOM)  ->;
mot(fromage,NOM) ->;
mot(viande,NOM)  ->;
mot(patee,NOM)   ->;
mot(mange,VB)    ->;
mot(grignotte,VB) ->;                                       .
mot(digere,VB)   ->;                                       .
mot(avale,VB)    ->;                                       .
mot(engloutit,VB) ->;                                       -23-

```

avec l'APPEND usuel. Voici deux exemples d'analyse:

```

>phrase(le.chien.avale.la.viande.nil,d);
d=PH.(GN.(ART.le.nil).(NOM.chien.nil).nil).(GV.(VB.avale.nil).
  (GN.(ART.la.nil).(NOM.viande.nil).nil).nil).nil
>phrase(le.chat.mange.nil,d);
d=PH.(GN.(ART.le.nil).(NOM.chat.nil).nil).(GV.(VB.mange.nil).nil).nil
>

```

et ce sont bien les arbres cherchés. En génération, voici toutes (!) les phrases acceptables:

```

>phrase(p,d);
p=le.chien.mange.nil d=
  PH.(GN.(ART.le.nil).(NOM.chien.nil).nil).(GV.(VB.mange.nil).nil).nil
p=le.chien.grignotte.nil d=
  PH.(GN.(ART.le.nil).(NOM.chien.nil).nil).(GV.(VB.grignotte.nil).nil).nil
p=le.chien.digere.nil d=
  PH.(GN.(ART.le.nil).(NOM.chien.nil).nil).(GV.(VB.digere.nil).nil).nil
p=le.chien.avale.nil d=
VOUS VENEZ D'APPUYER SUR LA TOUCHE ESC
>

```

Ne jamais oublier la touche ESC lorsque l'ennui gagne... Il y a plusieurs manières d'étendre ce minuscule programme:

- en l'optimisant tout d'abord. Il est possible d'éviter les appels à APPEND, qui coutent du temps en générant toutes les scissions possibles (en particulier après que la solution soit trouvée), en décrivant une liste comme

différence de deux autres listes. Ce procédé est par exemple employé dans le manuel d'exemples (p. 48) à-propos du problème arithmétique de Baxter, auquel nous vous renvoyons. Il est aussi proposé dans l'ouvrage de Clark & McCabe, qui a inspiré la grammaire qui nous intéresse. Nous en dirons un mot dans le prochain thème.

- en étendant la grammaire elle-même (adverbes, adjectifs, subordonnées ...).

La liste comme différence de deux listes

Prenons, ainsi que le font Clark & McCabe, l'exemple de la fonction REVERSE. En LISP, la version naïve s'écrit:

```
(def reverse (L) (and L (append (reverse (cdr L)) (list (car L)))))
```

soit, en détaillant au maximum:

```
(def reverse (L)
  (cond
    ((null L) nil)
    (t (let ((x . y) L))                ; par filtrage de Le_Lisp
        (append (reverse y) (cons x nil))))))
```

et cette dernière version se traduit immédiatement en PROLOG:

```
-----
reverse(nil,nil) ->;                                -1-
reverse(x.y,L)   -> reverse(y,y1) append(y1,x.nil,L); -2-
-----
```

et alors:

```
>reverse(1.2.3.nil,L);
L=3.2.1.nil
>
```

Sans entrer dans trop de détails, il se trouve qu'il est possible d'optimiser l'appel récursif en LISP, en le rendant "terminal" (ce qui équivaut quasiment à une itération pour l'interpréteur Le_Lisp). La technique relève d'une "transformation de programme" assez simple, décrite par exemple dans l'ouvrage sur LISP publié chez Nathan. La version obtenue utilise alors une "variable-tampon" pour y stocker les résultats intermédiaires:

```
(def apprev (L L1)                                ; L1 = variable-tampon
  (cond
    ((null L) L1)
    (t (let ((x . y) L)
        (apprev y (cons x L1))))))
```

et l'appel se fera par:

```
(apprev '(1 2 3) nil)
```

pour inverser la liste (1 2 3). Cette version est plus performante, à cause de la disparition de l'appel à APPEND, cher en PROLOG à cause des nombreux retours-arrière qu'il occasionne. Son analogue PROLOG s'obtient par traduction à vue:

```
-----
apprev(nil,L,L)   ->;                                -1'-
apprev(x.y,L1,L2) -> apprev(y,x.L1,L2);              -2'-
-----
```

et, là aussi, l'appel se fera par :

apprev(L,nil,L2)

pour obtenir dans L2 l'inverse de la liste L. Notez que l'argument L2 ne change jamais, c'est la liste-résultat (l'inverse); elle est scindée en deux parties, la partie-gauche étant le morceau à inverser, et la partie-droite le morceau inversé déjà obtenu. Voici l'évolution de l'appel "apprev(1.2.3.nil,nil,L2)" :

1 2 3	apprev(1.2.3.nil,nil,L2)
-----------	--------------------------

2 3 1	apprev(2.3.nil,1.nil,L2)
---------------	--------------------------

3 2 1	apprev(3.nil,2.1.nil,L2)
---------------	--------------------------

3 2 1	apprev(nil,3.2.1.nil,L2)
-----------	--------------------------

le cas générique :

L L1	apprev(L,L1,L2)
------------	-----------------

Comme on peut le constater, on obtient à chaque étape l'interprétation géométrique suivante de la relation "apprev(L,L1,L2)" :

apprev(L,L1,L2) signifie que l'inverse de L est L2 - L1

si l'opération de soustraction sur les listes est définie en prenant APPEND comme addition :

$$L = L2 - L1 \quad \Longleftrightarrow \quad L2 = L + L1 \quad \Longleftrightarrow \quad \text{append}(L,L1,L2)$$

Par exemple :

$$\text{apprev}(2.3.\text{nil},1.\text{nil},3.2.1.\text{nil}) \Longleftrightarrow (3 \ 2) = (3 \ 2 \ 1) - (1)$$

Il y avait donc une manière "naturelle" d'obtenir ce nouveau programme, qui consistait à représenter le résultat - l'inverse de la liste L - comme "différence" de deux listes U - W. Ecrivons que :

apprev(x.y,U,W) -> ?...

en pensant "l'inverse de x.y est U - W si ...":

$\begin{array}{l} \text{inv}(x.y) \\ \langle * * * * * \langle \text{---}W\text{---} \rangle \rangle \\ \langle \text{---}U\text{---} \rangle \end{array}$	si	$\begin{array}{l} \text{inv}(y) \\ \langle * * * * * x \langle \text{---}W\text{---} \rangle \rangle \\ \langle \text{---}U\text{---} \rangle \end{array}$
--	----	--

i.e. si l'inverse de y est U - x.W :

apprev(x.y,U,W) -> apprev(y,U,x.W);

ce qui, avec le fait "apprev(nil,U,U) -> ;" est bien le programme obtenu dans l'ouvrage cité, qui applique cette technique à l'élimination d'APPEND dans le programme du thème précédent.

III

Perspectives - Bibliographie

Il est toujours délicat de dire "tel type de programmation est meilleure que telle autre". De la programmation pour quoi faire ? Fabriquer du logiciel efficace ou former des élèves n'est assurément pas la même chose. Il me paraît assez spécieux de dire que dans l'enseignement de la programmation, le langage n'a pas d'importance; il permet de fournir au débutant (et pas seulement à lui) un cadre de pensée dans lequel il puisse formuler ses idées aussi confortablement que possible.

LISP apporte avec lui l'idée de régir tout le contrôle par le biais de fonctions. La syntaxe est réduite au minimum, et le langage est extensible à volonté par le programmeur. Pouvoir insister sur le fait qu'un programme n'est pas autre chose - au fond - qu'une fonction composée, est un fantastique outil pédagogique. Mais voir ce principe réellement à l'œuvre en programmant, voilà qui est encore plus satisfaisant, et vaut tous les discours sur la bienfaisance de la modularité. L'élève formé à LISP (ou à LOGO avec quelques réserves) dispose d'un certain nombre de réflexes qui le rendront exigeant à la fois sur la façon de concevoir un programme, et sur le choix du langage lui-même. Être capable d'évaluer le confort de programmation qu'apporte un langage est aussi quelque chose d'intéressant en soi.

PROLOG nous a introduit dans un autre univers conceptuel. Précurseur des futurs langages "déclaratifs", il a une méthode bien à lui: vous voulez calculer ceci ? déclarez-moi donc ce que vous savez sur le sujet. Le programme doit être perçu comme un ensemble de connaissances (des objets connus et des relations entre des objets génériques). Là aussi, la syntaxe est minimale, et l'effort devra porter sur la manière de déclarer ses connaissances pour fabriquer un "micro-monde". A l'heure où l'Intelligence Artificielle fait parler d'elle dans les journaux du soir, et où il est permis d'espérer qu'il s'agit plus que d'une mode, il serait dommage de ne pas en parler à des élèves, futurs utilisateurs de gigantesques banques de données. Ne voir dans l'activité de programmation que la simple constitution d'une telle banque, est peut-être (osons-le: sans doute) dans le droit fil de notre futur.

L'aspect "fonctionnel" de LISP, et celui "déclaratif" de PROLOG me paraissent finalement, après cette année expérimentale, très bénéfiques pour la formation des élèves. Souhaitons seulement pouvoir disposer bientôt de versions plus sophistiquées (graphiques par exemple) disponibles sur les micro-ordinateurs du marché. Remercions à cet effet le G.I.A. de Marseille-Luminy et l'INRIA de Rocquencourt pour leurs travaux respectifs sur PROLOG II et Le_Lisp, disponibles bientôt sur des dizaines de machines, parmi les plus diffusées.

Bibliographie

Elle est loin d'être exhaustive.

Sur LISP:

- "LISP" par G. Kiremitdjian et J.P. Roy (Nathan, décembre 84)
- "LISP: Mode d'emploi" par C. Queinnec (Eyrolles, septembre 84)
- "Le_Lisp 80: Le manuel de référence" par J. Chailloux (CNDP)

sur PROLOG

- "PROLOG II", les 3 manuels cités à la fin de II.2
- "PROLOG, langage de l'Intelligence Artificielle" par A. Colmerauer
La Recherche, septembre 1984
- "Logic for problem solving" by R. Kowalski (North-Holland)
- "Programming in PROLOG" by W.F. Clocksin & C.S. Mellish (Springer)
- "How to solve it with PROLOG" by Coelho, Cotta & Pereira
commander à (Ministerio da habitacao e obras publicas,
Laboratorio nacional de engenharia civil,
Lisbonne, Portugal)
- "Micro-PROLOG: programming in Logic" by K.L. Clark & F.G. McCabe
(Prentice-Hall)

Sur l'Intelligence Artificielle

- "La Sième génération" par E. Feigenbaum et P. McCorduck (Inter-Editions)
 - "Machines who think" by P. McCorduck (Freeman & Co)
 - Le numéro de décembre 1984 de la revue "Education et Informatique"
 - "Les systèmes experts" par M.O. Cordier (La Recherche, janvier 1984)
 - "Artificial Intelligence" by E. Rich (McGraw-Hill)
-

PUBLICATIONS DE L'I.R.E.M DE PARIS VII

- 4 GROUPE FRANCAIS/MATHEMATIQUES (TOME 1)
- 10 CALCUL MENTAL
- 11 CALCUL 1ER CYCLE
- 14 DE LA TEMPERATURE RESULTANTE A L'ANGLE SOLIDE
- 15 GROUPE FRANCAIS/MATHEMATIQUES (TOME 2)
- 21 ALMANATHEMATIQUE
- 22 FRISES .
- 23 RECURRENCE
- 24 CLIMATISONS LES MATHEMATIQUES
- 25 LES FORCES EN STATIQUE
- 26 BIBLIOGRAPHIE POUR L'ENSEIGNEMENT DES MATHEMATIQUES
- 27 LE NOMBRE D'OR
- 28 ROTATION/ANGLES
- 29 LA VITESSE .
- 32 LES ANGLES
- 34 METROLOGIE
- 35 ACTIVITES SPATIALES
- 36 TRAINS EPICYCLOIDaux
- 39 POT POURRI
- 40 GROUPE DE RECHERCHE AVIATION/ENSEIGNEMENT (N° 1)
- 41 RESEAUX
- 43 CINEMATIQUE RELATIVISTE
- 45 PETITES VARIATIONS OU L'ART DE DERIVER SANS LE SAVOIR
- 46 NOMBRES A L'ECOLE ELEMENTAIRE
- 47 MATERIAUX POUR LOGO
- 48 MESURE DES LONGUEURS ET DES AIRES
- 49 DEVOIRS A LA MAISON POUR LE PREMIER CYCLE
- 50 GROUPE DE RECHERCHE AVIATION/ENSEIGNEMENT (N° 2)
- 51 INSTRUMENTS DE GEOMETRIE

Reproduction de textes anciens

- I - DISME .
- II - GEOMETRIE ELEMENTAIRE
- IV - DICTIONNAIRE MATHEMATIQUE (TOME 2)

NOUVEAUX PROGRAMMES DES COLLEGES ET LYCEES

QUESTIONNAIRE POUR APPRENDRE

BROCHURES INTER-I.R.E.M

N° 2 : TECHNOLOGIE ET MATHEMATIQUES EN F

N° 3 : QUELLES ACTIVITES POUR QUEL APPRENTISSAGE

BULLETIN INTER-I.R.E.M N° 22 (catalogue publications I.R.E.M)

THESE D'ALINE ROBERT

DEUG/SSM

BULLETIN DE LIAISON DES PROFESSEURS DE MATHEMATIQUES

Editeur : I.R.E.M

Directeur/Responsable de la publication : François COLMEZ

Tirage : 200 exemplaires

dépôt légal : 2-88612-025-6

SEPTEMBRE 1984

I.R.E.M PARIS VII - Tour 55/56 - 3ème étage - 2 place Jussieu 75005 PARIS