PROLOG

VERSION TO7, TO7-70, TO9, MO5

Réalisation MICLOG
en collaboration avec CRISS-GRENOBLE
et avec l'aide de l'A.D.I.

Photo couverture : PATRICK BERTOLOTTI

COPYRIGHT © FRANCE IMAGE LOGICIEL 1986
Tous droits de reproduction, d'adaptation et de traduction
réservés pour tous pays.

I - PRÉSENTATION GÉNÉRALE

1 - OBJECTIFS 5

2 - PRINCIPES 6

1 - OBJECTIFS

Ce logiciel est destiné à toute personne désirant s'initier à la programmation en PROLOG, qu'elle ait ou non des connaissances en informatique.

Il fonctionne sur tout micro-ordinateur TO7 (extension mémoire), TO7-70, TO9 ou MO5.

Programmer en langage PROLOG, c'est découvrir les principes et les notions de base qui génèrent ce nouveau domaine, très prometteur, qu'est l'Intelligence Artificielle. Domaine totalement différent par l'esprit de celui de l'informatique dite « traditionnelle », il donne lieu à une démarche de programmation toute différente et ouvre la voie à de nouvelles applications, non encore traitées par l'informatique. Qui n'a pas entendu parler de bases de données relationnelles, de systèmes experts, de reconnaissance des formes ou du langage ?...

En utilisant PROLOG, vous serez à même de créer et de tester vos propres programmes, que vous soyez informaticien ou non.

2 - PRINCIPES

L'articulation du manuel correspond aux objectifs à atteindre. Nous espérons qu'il répondra à votre attente.

Nous l'avons divisé en deux grandes parties :

— introduction à l'Intelligence Artificielle,
— programmation en PROLOG.

L'utilisateur choisira, selon sa formation, les parties qu'il désire approfondir.

Il vous est conseillé, en particulier si vous n'êtes pas informaticien, de lire attentivement la partie consacrée à l'Intelligence Artificielle, afin de mieux apprécier ensuite les avantages de la programmation en PROLOG. De plus, cette approche de l'Intelligence Artificielle permet de se resituer dans le contexte du monde informatique, et de saisir le « pourquoi » d'une telle démarche.

La partie relative à la programmation en PROLOG se décompose comme suit :

— dans un premier chapitre, une explication de la mise en marche et du fonctionnement général du logiciel PROLOG sur TO7-TO7/70-TO9-MO5. C'est à ce chapitre que vous vous référerez pour une première utilisation du produit ;

— le chapitre 2 vous invite à une première rencontre avec PROLOG. Cette partie d'initiation introduit, de façon simple, les principes généraux et l'esprit de PROLOG ;

— le chapitre 3 : « Comment programmer en PROLOG ? » décrit plus précisément la syntaxe de base de PROLOG, c'est-à-dire les éléments utilisés pour la programmation en PROLOG, et leur signification. Il introduit progressivement ces différentes notions (avec des exemples à l'appui), et permet de comprendre, en suivant les grandes étapes de résolution d'un programme simple, ce que représente un programme écrit en langage PROLOG ;

— le chapitre 4 est un manuel de références, regroupant toutes les commandes et tous les prédicats prédéfinis utilisables en PROLOG et leur syntaxe, classés par ordre alphabétique.

Enfin, en annexe, vous trouverez :

— des exemples de programmes PROLOG ;
— un lexique ;
— un index des noms cités ;
— une liste des messages d'erreurs et leur signification ;
— une liste des caractères de contrôle.

II - INTRODUCTION A L'INTELLIGENCE ARTIFICIELLE

1 - LA SITUATION DE DÉPART : L'INFORMATIQUE TRADITIONNELLE 9

1.1 - Principes 9

1.2 - Avantages et inconvénients 11

2 - L'APPORT DE L'INTELLIGENCE ARTIFICIELLE 13

2.1 - Les origines 13

2.2 - Les objectifs 13

2.3 - Les principes 14

2.4 - Les domaines d'application de l'Intelligence Artificielle 15

2.4.1 - Les bases de données relationnelles .. 16
2.4.2 - Les systèmes experts 17
2.4.3 - La compréhension des langages naturels 20
2.4.4 - La reconnaissance des formes 22
2.4.5 - L'avenir par l'Intelligence Artificielle 23

3 - UN LANGAGE D'INTELLIGENCE ARTIFICELLE PROLOG 25

3.1 - Les origines 25

3.2 - PROLOG et la logique 26

3.2.1 - Le calcul des prédicats 26
- Les objets 26
- Les propositions 27
- Mise sous forme clausale d'une proposition 32
- Une formalisation pour les clauses 36
3.2.2 - Démonstration de théorèmes : le Principe de Résolution 37
3.2.3 - Rapports entre PROLOG et la logique . 40

L'Intelligence Artificielle ? Un terme qui fait peur, car il allie une caractéristique propre à l'homme à un adjectif qui peut sembler contradictoire voire effrayant... Serait-on en passe de perdre ce qui fait le propre de l'homme ? Essayons d'abord de comprendre ce que recouvre exactement le terme d'« Intelligence Artificielle ». Et pour cela, remontons aux origines.

1 - LA SITUATION DE DÉPART : L'INFORMATIQUE TRADITIONNELLE

1.1 - PRINCIPES

A l'origine, l'informatique a été pensée pour résoudre des problèmes concrets et spécifiques. Ainsi, elle a tout d'abord été utilisée dans les domaines de la gestion et du calcul, où les besoins d'un outil performant se faisaient sentir. On a créé des langages, plus ou moins orientés vers des domaines précis (COBOL et sa capacité à manipuler un grand nombre d'informations pour la gestion, FORTRAN et son efficacité dans la représentation et le traitement des nombres, etc.). Avec le temps, les domaines d'application se sont diversifiés, et par là même, les langages.

Mais quels qu'ils soient et quelle que soit leur spécificité, ces langages fonctionnent tous selon le même principe de base : la programmation **impérative.**

Voyons un peu ce que cela signifie. Pour résoudre un problème par l'informatique traditionnelle, il faut exposer celui-ci à l'ordinateur sous forme d'un programme, c'est-à-dire d'une succession d'opérations (ou instructions) portant sur des données (numériques ou non, stockées en mémoire). Une fois que ce programme est enregistré en mémoire, donc qu'il a acquis une existence physique propre et qu'il est connu du système, il peut être exécuté.

Rappelons brièvement la composition de l'architecture interne d'un ordinateur. L'ordinateur comprend d'abord une mémoire où sont stockées les données et les instructions du programme. Il dispose également d'une unité centrale ou processeur, destiné à permettre l'exécution d'un programme. Différents équipements périphériques (écran, clavier, disques magnétiques, etc.) complètent ce dispositif et permettent d'améliorer la communication entre l'ordinateur et l'utilisateur.

Les programmes sont constitués d'opérations, ou plus généralement d'instructions, c'est-à-dire d'ordres donnés à la machine. Chaque instruction correspond à une opération élémentaire que l'ordinateur sait réaliser, comme par exemple l'addition de tel nombre à tel autre, la comparaison de deux nombres, la lecture ou l'écriture d'une donnée, etc.

Le nombre et la performance des instructions qu'un programme peut exécuter dépendent du **langage machine** propre à chaque modèle d'ordinateur. En effet, il faut savoir que les instructions que vous donnez en langage plus ou moins clair (mais de toute façon formalisé) à l'ordinateur ne peuvent être comprises par celui-ci que si elles sont préalablement recodées (en code binaire). Le langage machine est, en fait, le seul que peut « comprendre » l'ordinateur. Ainsi, quel que soit le

langage utilisé pour la programmation, il y aura toujours, à un moment ou à un autre, mais de toute façon avant l'exécution du programme, traduction automatique du programme en langage machine.

Ceci vient du fait que l'unité centrale (ou processeur) de l'ordinateur, responsable du traitement à l'exécution, ne permet le traitement que de deux situations (correspondant à des états électriques) qui, en se combinant, donnent lieu à une multitude d'autres possibilités.

Vous comprendrez peut-être mieux, en prenant la simple analogie suivante. Considérons que chaque modèle d'ordinateur est constitué, de façon très simplifiée, d'une série d'interrupteurs (ou bits), dont le nombre varie selon le modèle. Comme vous le savez, un interrupteur n'a que deux positions. Il est soit ouvert (que nous noterons 0), soit fermé (que nous noterons 1). Plus on utilise d'interrupteurs, plus on obtient de combinaisons différentes. Si N est le nombre d'interrupteurs, on a 2^N combinaisons possibles.

Exemples :

2 interrupteurs = 4 combinaisons 00
01
10
11

3 interrupteurs = 8 combinaisons 000
001
010
011
100
101
110
111

Du nombre d'interrupteurs utilisés dépend le nombre d'instructions utilisables. En effet, chaque instruction est codée en machine sous la forme d'une des combinaisons possibles. Ainsi le langage machine définit l'ensemble des instructions qu'il est capable de comprendre et d'exécuter.

Lorsqu'un programme est exécuté, l'unité centrale lit d'abord la première instruction du programme (après l'avoir codée en langage machine), réalise l'opération qui lui est demandée, et qui, en général, a pour effet de modifier l'une des données du programme (le résultat est renvoyé dans la mémoire centrale), puis passe à l'instruction suivante, etc. Le traitement des instructions s'effectue ainsi jusqu'à épuisement des instructions du programme, de façon séquentielle, selon le principe énoncé par John Von Neumann en 1946.

Pour rendre l'ordinateur encore plus performant, et éviter de le reléguer au rang de simple automate qui ne ferait qu'exécuter mécaniquement, on utilise des instructions dites *conditionnelles* et *inconditionnelles*. Les instructions conditionnelles, comme leur nom l'indique, ont une action qui dépend des résultats précédemment obtenus (si le résultat est... alors on fait... sinon on fait...). De la même façon, les instructions inconditionnelles forcent le programme à

effectuer telle ou telle instruction, qui n'est pas forcément la suivante dans le programme. Ces instructions ont pour effet de « dérouter » l'ordinateur vers une autre partie du programme.

Ce principe de programmation impérative, en pas à pas, est donc utilisé par tous les langages de programmation traditionnelle. Si sa renommée n'est plus à faire, nous pouvons cependant nous interroger sur son universalité : ce type de programmation peut-il tout résoudre ?

1.2 - AVANTAGES ET INCONVÉNIENTS

Comme nous l'avons vu, un bon point d'abord : s'il existe un très grand nombre de langages, c'est pour essayer de résoudre de façon toujours plus performante des problèmes différents. Ainsi cette programmation s'avère très efficace et souvent irremplaçable pour les domaines classiques bien connus, tels que la gestion, le calcul numérique...

Mais la programmation impérative pose également un certain nombre de problèmes.

● Tout d'abord, on ne pourra réaliser puis exécuter un programme que si l'on a clairement mis au point précédemment une méthode de résolution, c'est-à-dire un *algorithme* : il faut organiser l'enchaînement des instructions à exécuter pour atteindre un but prédéfini. C'est le programmeur qui effectue la plus grosse part du travail : il donne la « recette » pour aboutir au résultat demandé et l'ordinateur applique « docilement » ses désirs, c'est-à-dire qu'il va traiter les données associées, chose souvent ardue à faire manuellement, compte tenu du nombre de données. L'ordinateur est ici utilisé en priorité pour sa rapidité de calcul.

Ainsi, pour réaliser le programme qui donne le signe d'un nombre X, il faudra traduire en langage de programmation impérative, l'algorithme suivant : si X < 0 alors signe = « - » sinon signe = « + ».

Tout ceci peut paraître paradoxal. On s'attendrait à ce que l'ordinateur, outil conçu pour aider l'homme, soit cependant doté d'une certaine intelligence. Il n'en est rien. L'ordinateur ne fait strictement que ce qu'on lui demande et ordonne de faire. Il n'est capable ici d'aucune initiative, d'aucun raisonnement. Il faut lui dire « quoi » faire et « quand » le faire. Cette contrainte implique que le programmeur « se plie » à la machine et non l'inverse.

● Un autre problème tout aussi important relève de la portée des algorithmes et programmes ainsi conçus. Ceux-ci sont destinés à résoudre un problème unique. Ainsi, si nous voulons modifier la question posée, nous sommes obligés de modifier également le programme ou une partie de celui-ci, afin de l'adapter aux nouvelles exigences. Chaque type de question nécessite donc un algorithme différent.

Ainsi, par exemple, un programme gérant un fichier de personnes (nom, prénom, rue, ville), écrit pour traiter des questions de la forme "Quelle est la ville où habite X ?" sera totalement inefficace pour

résoudre cette autre question : "Quelles sont les personnes qui habitent (ont pour ville) Y ?". Pourtant, on voit bien que ces deux problèmes se ressemblent, sont liés implicitement. Mais dans un cas, le résultat est unique, dans l'autre, c'est un ensemble ; donc les traitements à effectuer vont être différents.

Ceci explique la situation actuelle de l'informatique, le coût du matériel baisse continuellement (les ordinateurs deviennent de moins en moins chers et de plus en plus puissants), contrairement à la réalisation de programmes (ou logiciels). Parallèlement, le développement de l'informatique (micro-informatique, informatique "domestique", ...) a créé des besoins nouveaux en matière de programmes, fiabilité et diversité des logiciels mis sur le marché deviennent exigences. Cela entraîne au niveau de la conception des logiciels, une programmation toujours plus importante et lourde, pour répondre à cette attente...

2 - L'APPORT DE L'INTELLIGENCE ARTIFICIELLE

2.1 - ORIGINES

Depuis les débuts de l'informatique, certains ont toujours pensé que l'ordinateur pourrait posséder des "facultés intellectuelles" supérieures ou égales à celles de l'homme. On tenait alors l'intelligence pour un concept simple. Nous avons vu ce qu'il en était, jusqu'à maintenant, l'ordinateur n'a fait qu'exécuter les ordres qui lui avaient été donnés. Il n'a fait preuve d'aucune intelligence (si ce n'est dans l'analyse syntaxique et sémantique qu'il effectue au cours de l'interprétation d'un programme).

Parallèlement au développement de l'informatique traditionnelle, les chercheurs en informatique se sont très vite, dès 1957, penchés sur ce problème. Comment rendre l'ordinateur intelligent? Et pour quels besoins? Mais d'abord fallait-il cerner le concept d'intelligence; lui donner une base théorique.

2.2 - OBJECTIFS

- Rendre l'ordinateur intelligent, c'est lui faire adopter une démarche proche de la pensée humaine. C'est lui permettre de traiter des connaissances qu'il a emmagasinées, pour aller au-delà de sa fonction d'automate. Ces connaissances ne sont pas des ordres. En fonction du problème posé, il les traitera pour en déduire, de façon plus ou moins intuitive (grâce à une gamme de mécanismes de raisonnement qui lui ont été incorporés), l'action à en tirer. Ici, on privilégie les bons principes d'organisation à la rapidité de calcul.

C'est exactement la même démarche que celle de l'homme qui, par exemple, arrive à reconnaître un visage (même si celui-ci est caché à moitié dans l'ombre, même si la personne porte des lunettes de soleil ou a changé de coiffure). En fonction du savoir et des connaissances qu'il a acquis, il en déduit l'action de reconnaître ou non le visage qui lui est présenté. Ainsi l'Intelligence Artificielle a-t-elle pour objectif d'analyser les comportements humains dans les domaines de la compréhension, de la perception et de la décision, dans le but de les reproduire de façon informatique.

- L'Intelligence Artificielle se propose également de répondre aux difficultés liées à la programmation classique. Comme nous l'avons vu (§ 1.2, p. 11), les exigences au niveau logiciel imposent une programmation de plus en plus lourde et diversifiée, afin de répondre à chacun des problèmes que l'on veut résoudre. L'Intelligence Artificielle, par l'esprit qu'elle veut utiliser, pourrait remédier à cela, en mettant au

point des logiciels capables de résoudre un grand nombre de problèmes : réaliser des logiciels très généraux, mais également "réutilisables" d'une machine à l'autre, mais aussi d'une application à l'autre. Ainsi, simplifiera-t-on, en l'améliorant considérablement, le travail et le confort de l'utilisateur.

● L'informatique traditionnelle est, on l'a vu, très performante dans la plupart des domaines classiques. Mais il est des domaines pour lesquels il n'existe a priori aucune méthode standard connue, aucun algorithme de résolution. Il en est ainsi pour de nombreuses activités de l'homme, qui ne peuvent être résolues qu'en se référant à des données et à un *contexte* connu et précis. L'Intelligence Artificielle a pour but d'ouvrir l'informatique à ces domaines, pour lesquels on ignore les mécanismes intellectuels employés, en mettant au point des machines spécialisées dans le traitement des connaissances. La démarche sera, pour un problème donné, d'essayer un chemin tout en gardant la possibilité d'en essayer d'autres, si celui qui paraissait prometteur n'a pas conduit rapidement à une solution.

Exemple :
Un programme qui résout des équations du second degré ne peut pas être considéré comme un programme d'Intelligence Artificielle (puisqu'on connaît une méthode menant à coup sûr vers la ou les solutions). Par contre, un programme jouant aux échecs en relève, puisqu'on ne connaît à l'avance aucune méthode conduisant au meilleur coup à jouer, quelle que soit la situation considérée.

2.3 - PRINCIPES

● Le premier point, non des moindres, a été, et est toujours, de cerner la base théorique de l'intelligence. Sans cela, impossible de suivre les objectifs précités. D'où des questions fondamentales. Comment apprend-on ? Comment parvient-on à appliquer nos connaissances accumulées, notre expérience, notre intuition, à des situations sans cesse renouvelées ?

Nous allons voir que, bien que nous ne sachions pas définir l'intelligence en général, nous pouvons cependant mettre en relief un certain nombre de caractères qui la définissent : la capacité d'abstraction ou de généralisation, la capacité de faire des analogies entre différentes situations, la faculté de s'adapter à des situations toujours nouvelles ou de corriger ses erreurs afin d'améliorer ses performances futures, etc. Supposons, par exemple, que l'on vous présente une tasse en inox et que l'on vous demande si elle peut faire office de tasse à café. Il est sûr que vous répondrez "non", si vous avez déjà eu le malheur de vous brûler les doigts au contact d'un récipient en métal rempli d'un liquide bouillant. Mais si vous n'avez pas le choix et que vous êtes obligé de boire votre café dans ce type de tasse, vous saurez que vous devrez tout simplement prendre vos précautions, en entourant la tasse d'une protection.

Cet exemple simple prouve combien il est difficile de formaliser ce sens commun, si intuitif et ambigu. Il montre que l'action engagée dépend fondamentalement des données et du contexte qui l'environnent... C'est là l'un des problèmes primordiaux quant au développement de l'Intelligence Artificielle. Mais les recherches vont bon train et l'avenir dans ce domaine semble des plus prometteurs.

● En Intelligence Artificielle, les problèmes à résoudre seront le plus souvent du type : comment exploiter au mieux un ensemble d'actions permettant d'aboutir à un but précis ?

La démarche d'un chercheur en Intelligence Artificielle sera donc la suivante : à partir d'une activité réputée intelligente, il émet des hypothèses (ou théories) concernant les informations et la démarche utilisée lors de la réalisation d'une telle activité, incorpore cette démarche dans un programme, et observe le comportement et les performances de ce programme. Ceci conduit à affiner la théorie de départ, qui conduit à son tour à modifier le programme, etc.

— d'un côté, la machine devra être en possession des connaissances de base nécessaires à la résolution d'un type de problèmes. Ces connaissances seront **déclarées** à l'ordinateur par l'utilisateur, comme des faits pertinents sur un domaine, un ensemble d'actions possibles. Une des caractéristiques des programmes d'Intelligence Artificielle est qu'ils manipulent des symboles (les connaissances) autres que numériques. Ceci rend les programmes plus clairs et plus lisibles, et contraste fortement avec l'idée généralement établie que l'ordinateur ne sait comprendre et manipuler que des chiffres (répandue par la représentation interne conventionnelle par "0, 1", voir § 1.1, p. 9) ;

— de l'autre, l'ordinateur traitera les connaissances par rapport à la problématique posée, et en déduira la validité ou non de la ou des réponses qu'il a pu envisager. Une autre caractéristique des programmes en Intelligence Artificielle est de séparer le plus nettement possible les connaissances du programme des mécanismes d'utilisation de ces connaissances (très important au niveau méthodologique).

2.4 - DOMAINES D'APPLICATION DE L'INTELLIGENCE ARTIFICIELLE

L'Intelligence Artificielle est un domaine neuf, en perpétuelle mutation et dont le champ d'action est encore plus ou moins bien défini. Son développement actuel, alors que l'informatique traditionnelle atteint à peine sa maturité, montre les immenses perspectives de progrès que l'on peut en attendre. Mais les bouleversements vont être progressifs comme on l'a vu, puisque le vaste domaine en cause est encore en partie inconnu. Les spécialistes tracent progressivement les grandes lignes, consolident et formalisent les premières approches de manipulation de la connaissance par l'ordinateur.

Cependant, on peut visualiser ces principes, en s'intéressant aux premières applications véritablement opérationnelles, ou qui le deviendront dans un avenir plus ou moins proche, connues sous le nom de bases de données, systèmes experts, compréhension du langage naturel, reconnaissance des formes. Ces exemples matérialisent efficacement les principes de base de l'Intelligence Artificielle.

2.4.1 - Les bases de données relationnelles

Les bases de données relationnelles permettent de travailler sur des données et sur leurs relations. Les données traitées ne sont plus simplement des éléments indépendants, mais un ensemble de données géré et associé à une même application. Les données sont reliées entre elles par la relation : Attribut/Objet/Valeur.

Exemples :

« père » de « Jean » est « Paul ».

« salaire » de « Jean » est « 6500 F ».

De ce fait, on peut questionner la base de données sous différents angles. Si on prend la première relation, on pourrait demander : "Qui est le père de Jean ?" ou "De qui Paul est-il le père ?".

Cette caractéristique des bases de données s'avère très intéressante lorsqu'on veut pouvoir manipuler confortablement des données, sans avoir, à chaque fois, à modifier la programmation en conséquence. Si nous reprenons l'exemple vu au § 1.2 (p. 11), nous voyons que le traitement par base de données nous serait très utile. La programmation traditionnelle ne pouvait pas nous permettre de gérer un fichier (nom, prénom, rue, ville) et de pouvoir poser simultanément des questions du type : "Quelle est la ville où habite X ?" et "Quelles sont les personnes qui habitent la ville Y ?". Ici, le problème se résoudrait facilement.

Le principe consiste donc à établir les relations pertinentes entre les données que l'on traite, afin de pouvoir permettre ensuite de poser toute sorte de questions sur ces données. Les données reliées entre elles constituent des *faits*. Un ensemble de faits s'appelle une *base de données*. A chaque fois que l'on pose une question, le système parcourt la base de données pour voir si un fait peut lui correspondre et retourne la réponse (voir chapitre 3, § 2.1, p. 64).

Les bases de données sont constamment et massivement mises à profit aujourd'hui. Elles permettent une gestion simple et souple des données traitées. De plus, elles donnent des résultats spectaculaires, car elles permettent un dialogue quasi-naturel qui donne une illusion de grande compréhension entre l'humain et la machine. Voyez leur utilisation dans la consultation de comptes bancaires, la vente par correspondance...

2.4.2 - Les systèmes experts

Les systèmes experts ont pour objectif, comme leur nom l'indique, d'analyser les comportements d'experts humains, en quelque domaine que ce soit (diagnostic médical, prospection minière, géologie, recherche de formules chimiques, conseil d'utilisation des systèmes informatiques, interprétation de sonogrammes, etc.), dans le but de les reproduire sur ordinateur. C'est un travail patient, souvent difficile, car les experts ne sont pas des machines. Lorsqu'ils acceptent de collaborer, ils ont souvent du mal à expliquer le "pourquoi" de leurs raisonnements, de leurs décisions. Il est à remarquer, de plus, que cette activité ouvre la voie à un nouveau métier de l'informatique : les ingénieurs "cogniticiens" ou ingénieurs de la connaissance, dont le rôle est d'acquérir le vocabulaire, les connaissances et les "trucs" de l'expert sur son domaine, en vue de les reformuler dans un système expert.

L'objectif des systèmes experts est de permettre plusieurs types d'utilisation :

— tout d'abord, ils peuvent venir en aide à des non-spécialistes qui ont besoin d'un conseil dans un domaine pointu. L'exemple le plus courant est celui du médecin généraliste qui pourrait consulter un système expert relatif à une spécialisation, avant de prendre une décision pour son client.

— ils peuvent également avoir une faculté pédagogique, permettant à tout candidat d'acquérir une formation dans une technique particulière.

— enfin, ils peuvent permettre de conserver l'expertise rare d'éminents spécialistes, qui risque de disparaître avant qu'ils n'aient pu la transmettre à leurs successeurs.

C'est une des applications opérationnelles de la recherche en Intelligence Artificielle. Pourquoi ? Parce que, comme nous allons le voir, elle regroupe tous les objectifs et principes exposés ci-dessus.

Tout d'abord, le comportement d'un expert est *indépendant* du domaine dans lequel il travaille. Seules ses connaissances changent. Ainsi, la réalisation d'un logiciel système expert, si elle est bien faite, permettra son utilisation dans n'importe quel domaine d'expertise. On pallie ainsi aux inconvénients de la programmation classique (un programme/une question).

• COMMENT TRAVAILLE UN EXPERT HUMAIN ?

Lorsqu'il se trouve face à un problème, il possède en fait une description partielle de la situation. A partir des faits qui lui sont connus, il émet des hypothèses qu'il teste par rapport au contexte et à son environnement de connaissances, revient en arrière pour émettre d'autres hypothèses qu'il teste, etc. Sa longue expérience lui permet également d'utiliser des "trucs" et son savoir déjà acquis. De cette démarche, il déduit une conclusion en fonction de la validité ou non des tests qu'il a effectués.

• PRINCIPES D'UN SYSTÈME EXPERT

Comme on l'a vu, il est impossible d'appliquer la notion d'algorithme et de programmation impérative à l'attitude d'un expert humain. En effet, la connaissance de l'expert est modulaire, c'est-à-dire qu'il ne l'utilise jamais totalement, mais plutôt par fragments, disponibles à tout moment, et choisis en fonction de la situation à laquelle il est confronté. Nous ne pouvons parler ici en terme d'algorithme, où toutes les données prévues sont requises et utilisées impérativement ; et où la communication ne peut se faire qu'en donnant des ordres. De là l'intérêt d'utiliser les moyens offerts par la recherche en Intelligence Artificielle. On remplace les ordres par des règles de production.

La démarche de l'expert humain serait amoindrie, si elle ne laissait aucune trace du cheminement effectué pour aboutir au résultat. Il est en effet important pour l'expert, de savoir "comment" et "pourquoi" il en est arrivé à ce résultat. En conservant la démonstration qu'il vient d'effectuer, il améliore considérablement son raisonnement et l'enseignement à en tirer.

Ce principe de cumulation du savoir et de la connaissance doit être pris en compte dans les systèmes experts. Le but est de pouvoir rechercher des faits qui ne sont pas enregistrés dans la base de connaissances, mais que l'on pourra déduire des faits existants.

• FONCTIONNEMENT D'UN SYSTÈME EXPERT

La plupart des systèmes experts sont composés d'une base de faits ou de connaissances, et d'une base de règles de production. Ces bases sont utilisées pour traiter votre problème, en fonction des connaissances dont vous disposez (le système les obtient en vous posant une série de questions de plus en plus fines, relativement à l'expertise en cours, et permettant d'aboutir à un diagnostic).

— **La base de faits :** elle permet de représenter les connaissances expertes sur un domaine. Ces connaissances sont déclarées au système de façon simple, lisible (une phrase par connaissance) et par conséquent modulaire ; ce qui permet son utilisation par fragments. Elle est conçue par et pour l'expert humain (bien que cette opération soit souvent délicate à effectuer, vu les "a priori" que peuvent éprouver les experts face à ce projet "d'imitation" de leur travail par l'informatique).

Elle doit être indépendante de la base des règles de production, mais cependant interprétable et utilisable par elle. En fait, la base de faits correspond à un "emmagasinement" brut des connaissances acquises dans un domaine.

— **La base des règles de production,** ou règles "si-alors" ou règles "situation-action", ou démonstrateur de théorèmes ou moteur d'inférence : elle se propose de trouver des faits nouveaux à partir de la base de connaissances et de la règle du "*Modus Ponens*". La règle du Modus Ponens nous dit que "si p est vrai et si on peut déduire q de p (noté $p \rightarrow q$) alors q est vrai". Elle donne donc les relations qui peuvent

exister entre certains faits exprimés et permet d'introduire des nouveaux faits dans la base.

Exemple :

soit les bases suivantes

base de faits initiale	:	A, B.
base des règles	:	(R1) C → D.
		(R2) E → F.
		(R3) A → C.
		(R4) B et D → G.

Il existe deux stratégies de démonstration :

— **Le chaînage avant :** en partant de la base de faits initiale, on va essayer de déduire des faits nouveaux, par utilisation de la base des règles. Rappelons que les éléments de la base sont des vérités et qu'on utilise, pour la démonstration, la règle du Modus Ponens. Parcourons la base de faits. On sait que A et B sont vrais. On va chercher dans la base de règles, toutes les règles dont la partie gauche est soit A, soit B. D'après la règle du Modus Ponens, on en déduit que la partie droite de ces règles est vraie et on introduit alors son contenu, qui est donc un nouveau fait, dans la base de faits. On poursuit cette démarche jusqu'à saturation.

Ici, on obtient :

(R3)	A → C	base de faits : A, B, C.
(R1)	C → D	base de faits : A, B, C, D.
(R4)	B et D → G	base de faits : A, B, C, D, G.

Remarque : Ici, on n'a pas pu utiliser directement la règle (R4), car nous ne connaissions pas le fait D. Il a fallu d'abord le déduire de (R3) et de (R1). La règle (R2) n'a pas été utilisée.

— **Le chaînage arrière :** c'est le cheminement inverse. Supposons que nous cherchions à démontrer que le fait G est vérifié. Il serait inutile d'utiliser le chaînage avant, qui travaille sur toute la base. Nous voulons ici démontrer une seule chose : c'est le *but* recherché.

Pour cela, nous allons travailler par étapes successives, en essayant d'utiliser toutes les règles qui ont pour partie droite (conséquence) le but recherché. Deux cas sont possibles. Le ou les faits de la partie gauche de la règle sont tous satisfaits dans la base initiale. Dans ce cas, le but est atteint. Sinon, on se crée des nouveaux buts, qui sont les faits inconnus de la partie gauche et on recommence ce cycle avec les nouveaux buts considérés. A la fin, soit on a réussi à démontrer tous les sous-buts, et dans ce cas, le but initial est vérifié ; sinon c'est l'échec.

Ici, cherchons à démontrer si le fait G est vérifié :

— on considère la règle (R4), et on définit B et D comme nouveaux buts. Le "et" utilisé ici signifie bien évidemment que les deux faits B, D doivent être vérifiés simultanément. B étant dans la base de faits, seul D est considéré comme nouveau but,

— on considère alors la règle (R1) qui nous donne comme nouveau but C,

— de (R3), on déduit A comme nouveau but. Or A est dans la base de faits donc le but initial est atteint, donc G est un fait.

Par contre, il serait impossible de vérifier le fait F. De la règle (R2), nous définirions un nouveau but E. Mais là, rien à faire : E n'est pas dans la base de faits et n'existe nulle part dans la partie droite d'une règle. Donc si nous considérons la base telle qu'elle est actuellement, nous ne pourrions en tirer aucune connaissance de E et de F. A vous d'imaginer ce qu'on pourrait rajouter dans la base.

- **CONCLUSION**

Deux caractéristiques essentielles différencient les systèmes experts des traitements en informatique traditionnelle, et constituent l'intérêt primordial de l'utilisation pratique de tels systèmes :

— les systèmes experts permettent d'expliquer son comportement à l'expert, de façon logique.

— ils peuvent recevoir de nouvelles connaissances de l'expert sans aucune nouvelle programmation, puisque celles-ci leur sont données dans un langage déclaratif et modulaire.

2.4.3 - La compréhension des langages naturels

Le problème le plus complexe de l'Intelligence Artificielle réside peut-être dans le désir de donner à l'ordinateur la possibilité de comprendre le langage naturel humain. La compréhension du langage naturel rendrait inévitablement la communication avec l'ordinateur plus aisée (c'est ce que nous avons déjà vu pour le dialogue à tendance "utilitaire" instauré dans les bases de données, entre l'homme et la machine).

Au lieu d'obliger l'utilisateur à devenir "informaticien" pour pouvoir utiliser un système informatique, pourquoi n'essayerions-nous pas, au contraire, de rendre l'ordinateur plus "humain" ? C'est l'axe de recherche le plus novateur de l'Intelligence Artificielle aujourd'hui. Mais les obstacles sont nombreux. Peut-on résoudre par la logique formelle de l'Intelligence Artificielle des problèmes aussi complexes ? Le langage humain est en effet bien souvent ambigu : dépendant d'un contexte, mais se référant également au "non-dit" et aux croyances et objectifs de la personne qui parle.

Exemple :
Supposons que quelqu'on fasse la demande suivante à son assistant, ou même à un système de données en langage naturel : *"Pouvez-vous me donner des informations sur les X lieux qui utilisent notre logiciel PROLOG ? J'aimerais les consulter par départements."* Cette formulation devrait être rapidement comprise par tout auditeur qui est dans le contexte, surtout s'il est familier avec les sociétés d'édition de

logiciels. Par contre, la compréhension et l'interprétation de cette demande par l'ordinateur sont beaucoup moins évidentes.

"Pouvez-vous me donner des informations..." : une réponse sensée à cette requête est "oui". Mais ce n'est pas ce que veut le demandeur ici. Le fait de savoir si l'ordinateur a la capacité de fournir ou non des informations ne l'intéresse pas. La requête n'est pas à prendre au sens premier, mais dans un sens plus profond auquel est associé le reste de la question. Pour cela, l'ordinateur devrait tout d'abord avoir acquis et compris les règles générales de la communication humaine, ainsi que les conventions qui y sont liées.

"... des informations sur les X lieux..." : cette partie de phrase révèle également quelques ambiguïtés. Que veut exactement le demandeur ? Des informations sur X installations quelconques ou sur celles plus spécifiques liées à un type d'utilisation ? L'ordinateur est-il capable de décider sans une connaissance du contexte pour une situation précise ? Cette connaissance du contexte doit être partagée entre le demandeur et l'auditeur pour que le problème puisse être résolu. Plus le contexte est précisé, moins le problème présentera d'ambiguïtés dans sa formulation et plus il aura des chances d'être résolu.

La demande est d'autant plus ambiguë et vague qu'on ne sait pas quel type d'informations est attendu. Le nom et l'adresse des sociétés et des utilisateurs de PROLOG ? Les applications qu'ils traitent ? Les remarques qu'ils peuvent avoir formulées ? Dans le cas d'une situation humaine, l'auditeur se fera certainement une idée de ce que le demandeur a "derrière la tête", ou pourra demander des précisions pour affiner la question.

Pour mener à bien cette demande, on voit que l'auditeur a intérêt à bien cerner les objectifs réels du demandeur. Ainsi, il aura une vision beaucoup plus précise de la situation, ce qui lui permettra de donner des réponses plausibles, et d'écarter rapidement les réponses non pertinentes.

En conclusion, l'auditeur doit posséder un haut niveau de connaissances pour comprendre et déduire que l'article "les" de la deuxième partie de la requête se réfère aux lieux précités ; que "*j'aimerais...*" n'est pas ici une formule littéraire, etc.

Ce type de connaissances n'est évidemment codifié nulle part, et l'objectif est donc maintenant qu'il devienne une grande part du répertoire de base de tout système de compréhension des langages naturels. Inversement (et c'est une question que l'on peut soulever, puisqu'elle est très débattue en ce moment), la mise en place de tels systèmes ne passe-t-elle pas tout d'abord par la formation de l'être humain à savoir poser les questions pertinentes et à savoir bien les poser ? Cela voudrait dire, et on se retrouve face aux problèmes fondamentaux de base, qu'une fois de plus l'homme serait obligé de se "plier" aux contraintes de la machine. Le débat est ouvert...

En tout cas, on peut déjà envisager la portée et les perspectives qu'engage une telle révolution dans le domaine de l'informatique et de l'accès à l'informatique. Ce domaine commence à porter ses fruits en

bureautique, selon diverses directions : traduction automatique, documentation, conversation en bon français entre la machine et l'usager, examen et analyse linguistique et logique de la validité de phrases...

• UN ASPECT SPÉCIFIQUE : LA COMPRÉHENSION DE LA PAROLE

Ce problème est encore plus ardu que celui de la compréhension du langage naturel écrit (ou tapé au clavier), et cela pour plusieurs raisons :

— le "signal" résultant de l'émission de parole n'est pas pur. Il comporte toutes sortes de "bruits" (liés à l'environnement) qui ne sont donc pas des informations pertinentes ;

— la prononciation d'une même phrase n'est pas parfaite et universelle. Il est bien évident qu'elle varie d'un individu à l'autre ;

— la prononciation d'un locuteur donné est elle-même variable, suivant l'état de celui-ci (psychologique, physiologique...) ;

— la prononciation d'une syllabe, voire d'un mot, n'est pas la même suivant que ces derniers sont isolés ou en contexte ;

— la frontière entre les mots successifs d'une phrase n'apparaît pas clairement sur le signal vocal. Il peut très bien y avoir des silences au milieu d'un mot, et aucun silence entre deux mots consécutifs ;

— des mots peuvent avoir la même prononciation mais une signification totalement différente : c'est le problème des homonymes (par exemple : mer, mère, maire).

Par définition, le système de compréhension reçoit en entrée un groupe d'éléments phonétiques correspondant à la transcription de la phrase prononcée. Il résulte de ce qui a été dit ci-dessus, que la compréhension comporte encore un certain indéterminisme (actuellement, le taux d'erreurs de compréhension est, dans le meilleur des cas, de l'ordre de 30 %).

2.4.4 - La reconnaissance des formes

Rendre l'ordinateur intelligent passe également par sa capacité à reconnaître des formes, à posséder une vision intelligente comparable à celle de l'homme.

Connecter une caméra à un ordinateur, en vue de transformer une scène en un tableau de points, dits "pixels", dont on mémorise les caractéristiques dans l'ordinateur, est devenu chose commune. Mais ces données mémorisées sont brutes, c'est-à-dire que l'ordinateur est incapable d'en déduire ce qu'elles représentent. Ce passage des données brutes à la reconnaissance des objets représentés nécessite, il est vrai, une démarche comparable à celle de la vision de l'homme, démarche fondamentale mais complexe à définir.

Les systèmes biologiques de vision intègrent nombres d'étapes progressives et hiérarchisées, avant d'aboutir à la reconnaissance des

formes. La reconnaissance se fait par parties : tout d'abord, on identifie les "primitives", comme par exemple les contours, les lignes, puis on extrait de notre connaissance culturelle l'information qui y correspond, comme par exemple la configuration d'un visage. Une dernière étape permet de présenter l'information (qui n'est pas encore une image au sens propre du terme) à un haut niveau du cerveau, pour être enfin interprétée en tant qu'image visionnée et reconnue (ou non). Bien sûr, la reconnaissance est une opération extrêmement rapide. On évalue à 45 millisecondes au maximum, le temps nécessaire à un être humain pour reconnaître un visage.

On ne prétend pas évidemment donner à l'ordinateur toutes les possibilités offertes à l'homme en ce domaine, mais, en s'en inspirant, on voudrait lui permettre la reconnaissance de certains types contours, rotation, position respective des objets. Ces capacités ne peuvent être acquises simplement en affectant une image à une bibliothèque de calibres (sauf dans certains domaines d'application très limités), car il y a trop d'ambiguités dans une image. Les lignes constituant un objet ne sont pas toujours toutes visibles. Certaines d'entre elles peuvent être amoindries par la présence d'une ombre ; cachées par d'autres objets qui se trouvent devant (nous présentant une image partiellement connue de l'objet) ; non apparentés simplement parce qu'elles ne se trouvent pas sur la surface qui nous est présentée de l'objet.

Pour pallier à ces problèmes, la vision humaine utilise une vision globale, alors que l'ordinateur travaille au point par point. De ce fait les recherches s'orientent vers un système de vision globale. Des résultats déjà opérationnels ont été obtenus dans le domaine de la robotique. Les robots de la dernière génération sont ainsi "capables" de raisonner sur les contours et les positions respectives des pièces qu'ils doivent manipuler.

2.5 - L'AVENIR PAR L'INTELLIGENCE ARTIFICIELLE

L'Intelligence Artificielle propose une nouvelle voie à l'informatique ; celle de l'intelligence de l'ordinateur, qui se "plierait" ainsi à l'homme, en utilisant la même démarche que lui.

Le parti en jeu peut paraître insensé, mais les recherches effectuées, et les applications déjà opérationnelles qui en découlent, montrent que ce domaine est en voie de devenir un domaine à part entière. L'Intelligence Artificielle est en train de devenir un métier, qui devrait devenir quantitativement significatif dans les années à venir, et qui commence à se structurer. On compterait aujourd'hui en France, une dizaine de sociétés spécialisées en Intelligence Artificielle et travaillant sur le logiciel, toutes créées depuis 1980, et en général par des chercheurs. De plus, un certain nombre de grands utilisateurs, constructeurs ou SSII (Sociétés de Service et d'Ingénierie en Informatique) travaillent pour leur propre compte dans ce domaine. Aux Etats-Unis, on compterait déjà une centaine de sociétés.

Au dire des nouveaux convertis, quand on attrape le virus Intelligence Artificielle, c'est pour la vie ou presque. Quel plaisir de formuler une problématique, dans un langage proche de sa façon de s'exprimer !

Mais il faut rester réaliste. Si ce domaine est appelé à un grand avenir, ce ne sera pas au dépend de l'informatique traditionnelle, qui, on l'a vu, reste et restera toujours très performante pour des applications précises. Ces deux méthodes et philosophies de programmation seront indéniablement complémentaires l'une de l'autre.

Recherches logicielles et matérielles vont donc bon train et constituent l'un des rendez-vous technologiques les plus importants de demain.

Nous avons surtout parlé de l'aspect logiciel (informatique déclarative, un programme/des questions) et pas du tout de **l'aspect matériel** dans ce chapitre. Il se trouve que cet aspect du problème relève encore de l'inconnu.

Comme nous l'avons vu, le fonctionnement général d'un ordinateur et sa structure interne sont toujours ceux définis par John von Neumann, en 1946 (§ 1.1, p. 9). C'est-à-dire que l'ordinateur exécute toujours les programmes en séquentiel. Certes, sa structure a considérablement évolué au fil des temps, et les progrès tiennent surtout à la miniaturisation, la performance et la rapidité des composants électroniques. Là, les progrès ont suivi un rythme exponentiel et on peut se demander si les limites physiques absolues d'évolution ne risquent pas d'être atteintes rapidement.

Pourtant les besoins en traitement de l'information sont pratiquement illimités. Ainsi, le nouveau domaine de l'Intelligence Artificielle aurait besoin de machines qui, outre leur rapidité de traitement, seraient mieux adaptées au traitement des connaissances, par opposition au traitement des données. C'est-à-dire rendre véritablement les machines intelligentes, en créant une structure interne comparable à celle des neurones du cerveau.

Or, nos neurones, même s'ils ont des performances relativement modestes, compensent cet inconvénient en se partageant le travail. Ainsi, dans le mécanisme de la vision, un grand nombre de neurones traitent simultanément les données captées par la rétine ; c'est-à-dire que le cerveau travaille en "parallèle". Malheureusement, ce n'est pas le cas des ordinateurs actuels, qui travaillent toujours en séquentiel.

D'où des projets des plus fous et ambitieux ; alors que nous croyions être arrivés au sommet de la compétence technologique, le nouveau pari est désormais de concevoir des ordinateurs "parallèles", capables de traiter, en même temps, différents sous-ensembles d'un même problème. C'est ce que cache le mystérieux ordinateur de la "*cinquième génération*" envisagé par les Japonais, puis les Américains.

Gageure ou proche réalité, ceci est à suivre ! Mais il est évident que si ce but est atteint, on aura créé des machines véritablement intelligentes, alors qu'actuellement cette intelligence est simulée par des logiciels.

3. - UN LANGAGE DE L'INTELLIGENCE ARTIFICIELLE : PROLOG

3.1. - ORIGINES

La machine n'est pas encore intelligente, on l'a vu. On compense cette lacune par des logiciels simulant l'intelligence. Or qui dit logiciels, dit programmation donc utilisation d'un langage approprié. Ces langages de l'Intelligence Artificielle doivent leur existence à l'introduction, "révolutionnaire" dans le milieu informatique, du concept de *traitement des listes*, en 1956 (voir § 3.1.1.3, p. 86). Pour la première fois, on pouvait considérer les ordinateurs non plus simplement comme de gigantesques "broyeurs de nombres", mais également comme de véritables manipulateurs de symboles.

De là est né le langage LISP, en 1959, spécialisé dans la formulation et la manipulation d'expressions représentant des déclarations simples et dans l'évaluation de fonctions mathématiques ou informatiques. Mais un de ses inconvénients, en particulier pour le développement de logiciels complexes, réside dans le fait que *tout* doit être défini explicitement (comme en informatique traditionnelle).

Puis apparaissent une nouvelle série et un nouveau type de langages, *les langages "déclaratifs"*, par lesquels le système devenait **capable de trouver la solution par lui-même,** à partir du moment où le problème lui avait été décrit de façon suffisamment fine. La tâche du programmeur devient simple :

— exprimer le problème ;

— définir les concepts traités ;

— déclarer les connaissances nécessaires à la résolution de cette problématique.

PROLOG est de ceux-là. Sa renommée n'est plus à faire, puisqu'en particulier, le rapport des Japonais concernant les "ordinateurs de la cinquième génération", stipule que ces ordinateurs du futur seront conçus pour "comprendre" ce langage, ou du moins un langage inspiré de PROLOG ("PROgrammer en LOGique").

Comment est né PROLOG ?

C'est Alain COLMERAUER (professeur d'informatique à la faculté des Sciences de Luminy à Marseille) qui l'a mis au point, dans les années 1970. L'objectif était double :

— suivre les nouveaux principes des langages déclaratifs ;

— utiliser un développement récent de la logique mathématique pour la mise au point d'un mécanisme de déduction : **le Principe de Résolution,** méthode de démonstration automatique de théorèmes, énoncée

par J.A. Robinson. Nous allons voir rapidement dans le paragraphe suivant, quels sont ces rapports établis entre PROLOG et la logique.

3.2 - "PROLOG" ET LA LOGIQUE

Comment s'est formé PROLOG ? A partir des principes de la logique.

Rappelons d'abord ce que représente le terme "logique". La logique est une façon de représenter des énoncés et de vérifier leur validité ou non, de manière absolument formelle. La logique formelle intéresse depuis toujours les mathématiciens et les philosophes, mais c'est seulement à partir de 1958 qu'on l'utilise pour représenter le raisonnement ou la prise de décision. La logique permet donc, à partir de la représentation de propositions et de relations entre ces propositions, de déduire de nouvelles propositions. C'est une partie de la logique appelée *Calcul de Prédicats.*

Ce langage formel permet d'exprimer un très grand nombre et une très grande variété de propositions, ce qui en fait un outil très efficace pour l'Intelligence Artificielle, donc en particulier pour PROLOG. Avant de voir son application au domaine qui nous concerne, nous allons définir ce langage, montrer comment il est utilisé pour exprimer des propositions et en déduire de nouvelles. Ces concepts fondamentaux de la logique formelle ont une grande importance pour la compréhension de PROLOG et de l'Intelligence Artificielle.

3.2.1 - Le calcul des prédicats

Le calcul des prédicats manipule des propositions portant sur des objets, et permettant d'exprimer des relations entre ces objets. Avant d'exprimer une proposition, il faut savoir comment décrire les objets concernés.

● **LES OBJETS :**

Les objets sont représentés par des *termes* (que nous retrouverons en PROLOG), ayant l'une des trois formes suivantes :

— **un symbole constant :** on considère alors que l'objet désigne un concept bien particulier et unique. Quel que soit le contexte dans lequel est utilisé l'objet, il a toujours la même valeur, la même signification ;

— **un symbole variable :** l'objet n'a pas de valeur précise et peut représenter plusieurs choses différentes à des instants différents, selon le contexte dans lequel on l'utilise. Cela nous permet de ne pas nommer expressément l'objet dont on parle. Il introduit la notion de quantificateurs, qui sera exposée plus loin. On le représente en général par l'une des lettres de la fin de l'alphabet (u, v, w, x, y, z) ;

— **un terme composé :** il est composé de deux parties; un symbole fonctionnel et un ensemble de termes ordonnés, qui sont ses arguments, noté à sa suite entre parenthèses. La signification du terme composé revient à lier les arguments entre eux par la relation énoncée dans le symbole fonctionnel.

Exemple :
Avec le symbole fonctionnel "distance" et deux arguments, on crée un terme composé distance (a, b), dont la signification est la distance qui sépare le point a du point b.

Toutes les fonctions mathématiques sont des termes composés.

On remarque que les arguments sont eux-mêmes des termes (c'est-à-dire qu'ils peuvent être soit une constante, soit une variable, soit un terme composé). Ainsi, on peut créer des termes composés plus complexes :
- distance (a, destination), la signification dépend du point de destination à définir,
- fois (5, plus [2,3]), la signification est 5* (2 + 3).

• LES PROPOSITIONS

Pour exprimer une proposition, on utilise un ou des symboles de prédicats qui permettent d'identifier la ou les relations, et des termes qui permettent de nommer les objets en relation. On distingue deux types de propositions :
— **Les propositions atomiques :** elles sont composées d'un symbole de prédicat et d'un ensemble ordonné de termes, qui sont ses arguments, placé à sa suite entre parenthèses. Ce sont des propositions élémentaires, non divisibles (correspondant aux prédicats en PROLOG).

Nous avons en général plusieurs possibilités pour décrire une relation par une proposition atomique. Par exemple, on pourra représenter la phrase "La voiture est rouge" soit par un prédicat à :
- un argument, comme dans ROUGE (Voiture),
- deux arguments, comme dans COULEUR (Voiture, Rouge),
- trois arguments, comme dans VALEUR (Couleur, Voiture, Rouge), etc.

Le choix effectué dépend du sens et des détails que l'on veut apporter à la relation.

Vous pouvez vous demander ce qui différencient les propositions des termes composés. Rappelez-vous qu'un terme représente un objet, donc une réalité, une propriété de notre univers. Par contre une proposition exprime une relation entre ces objets. Cette relation peut être *vraie* ou *fausse*, selon les objets que l'on considère. Exprimer une proposition, c'est donc essayer de réunir plusieurs objets. Une proposition logique possède donc une valeur de vérité : elle est soit vraie (notée V), soit fausse (notée F), mais ne peut être les deux à la fois. Par contre un terme composé est posé comme vrai a priori.

— **Les propositions composées :** elles sont obtenues en reliant plusieurs propositions atomiques ensemble, grâce aux connecteurs logiques "non", "et", "ou", "implique", "est équivalent à". Cela permet de travailler sur des propositions plus complexes et certainement plus intéressantes. De plus, elles permettent souvent d'énoncer des propositions à partir de phrases qui, même simples, ne peuvent être transcrites simplement par des propositions atomiques.

Arrêtons-nous ici quelque peu sur ces principes. Dans tout ce qui suit, A et B seront deux propositions atomiques quelconques.

— **Le connecteur de négation "non" :** à toute proposition notée A, on associe une nouvelle proposition notée $\neg A$ (non A) suivant le principe du tiers-exclu. L'une des deux propositions A, $\neg A$ est vraie et une seule. Cela veut dire que si A est vrai alors $\neg A$ est faux et inversement. En fait, c'est un "faux" connecteur, puisqu'il ne sert pas à "connecter" deux propositions. Visualisons cela sous forme d'un tableau de vérité.

A	$\neg A$
V	F
F	V

Exemple :

A : "L'homme est mortel"
$\neg A$: "L'homme est immortel".

— **Le connecteur de conjonction : "et" : $A \wedge B$:** on associe à la conjonction de ces deux propositions, une troisième appelée conjonction logique, notée ($A \wedge B$) ou (A et B). Cette proposition est vraie dans le seul cas où les propositions A et B sont simultanément vraies, ce que montre le tableau de vérité suivant :

A	B	$A \wedge B$
V	V	V
V	F	F
F	V	F
F	F	F

Exemple :
La phrase "Pierre possède une voiture rouge" peut être représentée par la proposition composée :

POSSÈDE (Pierre, Voiture) $\wedge$ COULEUR (Voiture, Rouge),

où le prédicat POSSÈDE établit une relation entre une personne et un objet, et le prédicat COULEUR, une relation entre un objet et une couleur.

— **Le connecteur de disjonction : "ou" : (A v B) :** on associe à la disjonction de ces deux propositions, une troisième appelée disjonction logique, notée (A v B) ou (A ou B). Cette proposition est vraie dès que l'une des propositions A ou B, ou les deux sont vraies. Elle n'est donc fausse que si les deux propositions sont fausses simultanément. En pratique, on n'examine jamais la seconde proposition si l'on sait que la première est vraie.

A	B	A v B
V	V	V
V	F	V
F	V	V
F	F	F

Exemple :
La phrase : "Pierre joue au ballon ou au tennis" pourra être représentée par la disjonction :

(JOUE (Pierre, Tennis) v JOUE (Pierre, Ballon)).

— **Le connecteur d'implication : A => B :** on associe à l'implication de ces deux propositions, une troisième notée (A => B) de la façon suivante :

((A => B) correspond à (A) v B)

C'est-à-dire qu'elle permet de représenter des propositions "si-alors" (si A alors B).

Dessinons le tableau de vérité de cette proposition composée, pour voir les conclusions que l'on peut en tirer :

A	1A	B	A => B
V	F	V	V
V	F	F	F
F	V	V	V
F	V	F	V

Nous en déduisons que la proposition (A => B) est vraie dans tous les cas sauf lorsque A est vrai et que B est faux. Cela revient à dire que

pour démontrer que la proposition (A => B) est vraie, il suffira de démontrer que si A est vrai alors B est vrai.

Exemple :
La phrase "Si cette voiture appartient à Pierre, alors elle est bleue" peut s'écrire :

APPARTIENT (Voiture, Pierre) => COULEUR (Voiture, Bleu).

— **Le connecteur d'équivalence : A <=> B :** on associe à l'équivalence de ces deux propositions, une troisième (A <=> B), de la façon suivante :

(A <=> B) correspond à (A => B) ∧ (B => A),
c'est-à-dire à (¬ A v B) ∧ (¬ B v A).

Elle permet de formuler des propositions du type "si et seulement si".

En nous aidant du tableau de vérité précédent, on peut déduire celui-ci :

A	B	A => B	B => A	A <=> B
V	V	V	V	V
V	F	F	V	F
F	V	V	F	F
F	F	V	V	V

Lorsque la proposition (A <=> B) est vraie, on dira que A équivaut à B ou que les propositions A et B sont équivalentes. Pour démontrer que les propositions A et B sont équivalentes, il suffira de démontrer que les propositions A et B sont de même nature (c'est-à-dire soit simultanément vraies, soit simultanément fausses).

On s'aperçoit, au vu des connecteurs => et <=>, qu'on peut associer un grand nombre de connecteurs et de propositions, pour former des propositions composées.

— **Les quantificateurs :** si nous limitions nos propositions à celles que nous pouvons écrire en utilisant les constructions précédentes, et si nous n'utilisions jamais de variables dans les termes, nous resterions dans le sous-ensemble du calcul des prédicats appelé *calcul propositionnel*, performant pour représenter des domaines simples. Nous arriverions vite à cette constatation : ce langage manque de puissance, car il ne nous permet pas de parler d'ensembles d'éléments, c'est-à-dire de plusieurs individus différents à des instants différents, mais vérifiant tous la même propriété (comme, par exemple "tous les chats sont gris").

Le calcul des prédicats nous permet d'aller plus loin, par l'utilisation de variables, comme nous l'avons vu précédemment, et de quantifica-

teurs. Cette extension du calcul des propositions nous permettra de décrire des propositions à propos d'objets du monde réel, et de pouvoir particulariser ceux-ci à un moment ou à un autre.

On distingue deux quantificateurs dans le calcul des prédicats :

• *Le quantificateur universel, noté* ∀

Soit une variable quelconque, et soit P un symbole de prédicat quelconque à un argument. La syntaxe : ∀ x, P(x) signifie que quelque soit l'élément x que l'on prend dans l'ensemble, la proposition P(x) est toujours vérifiée ; ou en d'autres termes, tout élément X de l'ensemble vérifie P(x). Placé devant une proposition, il voudra dire que *tous* les éléments de l'ensemble vérifient la proposition.

Exemples :

(∀x, HOMME <x> => MORTEL <x>)

signifie que pour tout x élément de l'univers, si x est un homme alors x est mortel, ou plus simplement « Tout homme est mortel ».

(∀x, CHAT <x> => COULEUR <x, gris>)

signifie que « Tous les chats sont gris ».

• *Le quantificateur existentiel, noté* ∃

C'est la négation du quantificateur universel. Dire que quelque chose n'est pas vérifié pour *tous* les éléments de l'ensemble est équivalent à dire qu'il existe *au moins un* élément de l'ensemble pour lequel la proposition n'est pas vérifiée. Placé devant une proposition, il voudra dire qu'*au moins un élément de l'ensemble vérifie la proposition. Ainsi,* ∃ x, P(x) signifie qu'il existe au moins un objet, noté x, qui vérifie la proposition P (x).

Exemple :

(∃ x, FEMME [x] => MÈRE [x])

signifie qu'il existe au moins un élément de l'univers, tel que si c'est une femme, alors c'est une mère. Cette proposition sera vérifiée ici pour tout le sous-ensemble de l'univers composé des mères de famille. La quantification porte ici sur une implication.

(∀x, ∀ y (∃ z, PÈRE (x,z) ^ PÈRE (z,y)) => GRAND-PÈRE (x,y))

signifie que s'il existe au moins un élément de l'univers (ici un et un seul) qui soit à la fois fils d'un individu et père d'un autre (en associant au premier argument du prédicat père le nom du père et au deuxième le nom du fils), alors le premier individu cité est le grand-père du deuxième.

La version du calcul des prédicats utilisés ici est dite de *premier ordre*, car elle ne permet pas la quantification sur des symboles fonctionnels ou de prédicats. Ainsi, une formule telle que (∀ P, P(A)) n'est pas autorisée dans le calcul des prédicats de premier ordre.

— Les règles d'inférence : elles permettent de créer de nouvelles propositions à partir de propositions déjà existantes. Les deux principales règles d'inférence sont :

• le Modus Ponens défini au § 2.4.2. (p. 17).
• la substitution consistant à remplacer une variable quantifiée par une constante.

Exemple :

$(\forall x, (P(x)) => P(A)$

• MISE SOUS FORME CLAUSALE D'UNE PROPOSITION

Nous possédons désormais tous les éléments de base pour travailler en logique, les termes et les propositions reliant ces termes entre eux. Mais nous nous apercevons que certains éléments sont redondants ; nous l'avons vu en particulier pour les connecteurs "=>" et "<=>", qui peuvent s'exprimer en fonction des autres connecteurs "et", "ou", "non". De plus les connecteurs "et", "ou" ne sont que l'opposé l'un de l'autre ; de même pour les quantificateurs ∀ et ∃.

A ce stade, on se demande alors s'il ne serait pas nécessaire de supprimer l'utilisation de certains d'entre eux. Cela ce ferait sans inconvénient :

- d'une part, en combinant les connecteurs restants et pertinents, on pourrait sans difficulté retrouver les autres,
- d'autre part, cela faciliterait la formalisation de notre démarche : en ne pouvant nous exprimer que d'une seule manière pour décrire des propositions, le raisonnement serait plus facile à suivre et plus du tout ambigu.

Transformer n'importe quelle proposition en *formalisation unique*, tel est le but de la *mise sous forme clausale*, dont le résultat est important pour comprendre PROLOG. Nous verrons que le résultat obtenu se réduit à l'utilisation d'un nombre très restreint de connecteurs. On n'utilise plus alors que les connecteurs de disjonction et de conjonction.

La mise sous forme clausale d'une proposition se décompose en six étapes :

— 1re étape : éliminer les implications. Si notre proposition comporte une ou des "=>" et/ou "<=>", on peut les éliminer en les remplaçant par leur équivalent donné plus haut. On rappelle que :

$(A => B)$ est identique à $((\neg A) \vee B)$
$(A <=> B)$ est identique à $((\neg A) \vee B) \quad ((\neg B) \vee A)))$.

Exemples :

$(\forall x, HOMME(x) => MORTEL(x)$
devient $(\forall x, \neg (HOMME(x)) \vee MORTEL(x))$.

$(\forall x, \forall y, (FEMME(x) \wedge PARENT(x,y)) => MÈRE(x,y))$
devient $(\forall x. \forall y, \neg (FEMME(x) \wedge PARENT(x,y)) \vee MÈRE(x,y))$.

— 2e étape : déplacer les négations vers l'extérieur. Si la formule n'est pas une proposition atomique, la négation qui la précède est

élargie à toutes les propositions qui la constituent, selon les règles suivantes :

$\neg (A \wedge B)$ est identique à $((\neg A) \vee (\neg B))$.
$\neg (\exists x, P(x))$ est identique à $(\forall x, \neg (P(x)))$.
$\neg (\forall x, P(x))$ est identique à $(\exists x, \neg (P(x)))$.

Ainsi, toutes les négations désormais présentes ne se réfèrent plus qu'à des formules atomiques. On donne le nom de *littéral* à toute proposition atomique, qu'elle soit ou non précédée d'un connecteur de négation. Désormais les littéraux sont les éléments de base de nos propositions.

Exemples :

$(\forall x, \forall y, \neg (R(x) \wedge S(x,y)) \vee R(y))$
devient $(\forall x, \forall y, \neg (R(x) \vee \neg (S(x,y)) \vee R(y))$.

$(\text{FEMME}(\text{Anne}) \wedge \text{MÈRE}(\text{Anne},x))$
devient $(\neg (\text{FEMME}(\text{Anne})) \vee \neg (\text{MÈRE}(\text{Anne}, x)))$.

— 3e étape : "skolemisation" ou élimination des quantificateurs existentiels. Au lieu de dire qu'il existe au moins un objet qui vérifie une proposition ou en ensemble de propositions, nous allons affecter à cet objet un symbole constant, donc un nom unique, appelé *constante de Skolem*. Cette constante a les mêmes propriétés que celle de l'objet. De plus, une formule ne peut être interprétée que si sa version "skolemisée" est interprétable. La constante ne doit pas être identique au nom utilisé dans la formule comprenant le quantificateur $\exists$, car le quantificateur $\exists$ ne dit pas qu'un objet de nom donné vérifie la proposition, mais simplement qu'un tel objet existe.

Exemple :

$\exists x, (\text{HUMAIN}(x) \wedge \text{AMI}(x,\text{Pierre}))$
devient $(\text{HUMAIN}(a1) \wedge \text{AMI}(a1,\text{Pierre}))$:
la constante de Skolem "a1" représente une personne qui est un ou une ami(e) de Pierre.

Le problème se complique lorsque les propositions comprennent également des quantificateurs universels.

Ainsi, si nous "skolemisons" :
$(\forall x, \forall y (\exists z, \text{PÈRE}(x,z) \wedge \text{PÈRE}(z,y)) \Rightarrow \text{GRAND-PÈRE}(x,y))$
on obtient :
$(\forall x, \forall y (\text{PÈRE}(x,h1) \wedge \text{PÈRE}(h1,y)) \Rightarrow \text{GRAND-PÈRE}(x,y))$.

Or cette formulation ne correspond pas à celle que nous voulons donner. Ici, nous avons affirmé que tout individu, reconnu comme étant père, était le père d'un même individu, nommé "h1" ; et que, de plus, "h1" était le père de tout "fils" de l'univers... Et cela, parce que le $\exists$ est dans la portée du quantificateur $\forall$.

En présence de quantificateurs universels, on doit indiquer que "ce qui existe" dépend de ce qui est représenté par la variable universelle. Pour cela, on introduit la notion de fonction.

Exemple :

(∀x, ∀y (PÈRE (x,h1 (x)) ∧ PÈRE (h1 (x),y)) => GRAND-PÈRE (x,y)).

Le symbole de fonction "h1" montre alors que la fonction de père dépend de l'individu que l'on considère.

— **4e étape : déplacer les quantificateurs universels vers l'extérieur.** Cette opération est simple et sans effet sur la signification générale de la formule. On place les quantificateurs en début de formule. On les appelle des *préfixes*. Ils sont suivis d'une formule sans quantificateur, appelée *matrice*.

Exemple :

(∀x, HOMME (x) ∧ AMI (x,Pierre)) ∧ (∀y, FEMME (y) ∧ AIME (y,Pierre))

revient à écrire :

(∀x, ∀y, (HOMME (x) ∧ AMI (x,Pierre)) ∧ (FEMME (y) ∧ AIME (y,Pierre)))

où ∀ x, ∀ y est le préfixe,

(HOMME (x) ∧ AMI (x,Pierre)) ∧ (FEMME (y) ∧ AIME (y, Pierre)) est la matrice.

Dès que les quantificateurs ∀ ont été placés devant la formule, c'est comme s'ils ne servaient plus à rien, et on peut les omettre. En effet, il suffit de se rappeler que toutes les variables utilisées désormais leur doivent l'existence.

— **5e étape : distribuer les "et" sur les "ou".** A ce stade de la démarche de mise sous forme clausale, les propositions sont déjà fortement transformées et simplifiées (bien que leur sens ne soit pas modifié, si on tient compte des conventions qui ont été prises). Les quantificateurs ont disparu, les connecteurs encore utilisés ne sont plus que les "et" et les "ou" (ainsi que les "non", mais ils font partie des littéraux). Nous allons désormais transformer une formule en une succession de "et" entre des éléments qui sont soit des littéraux, soit des groupes de littéraux reliés par des "ou", selon les règles suivantes :

(A ∧ B) v C est identique à (A v C) ∧ (B v C).
(A v (B ∧ C) est identique à (A v B) ∧ (A v C).

On dit que l'on met la formule sous *forme normale conjonctive*.

Exemples :

(A v (B ∧ (C v D)))

devient ((A v B) ∧ (A v (C v D))).

((A v (B ∧ (C v ¬ D))) v E)

devient (((A v B) ∧ (A v (C v ¬ D))) v E).

— **6e étape : mise en clauses.** De façon générale, les formules dont nous disposons maintenant sont composées d'une suite de "et" (∧) reliant soit des littéraux, soit des groupes de littéraux reliés eux-mêmes par des "ou" (v).

Sans entrer dans le détail, c'est-à-dire sans distinguer pour l'instant les types d'éléments reliés par des "et", on peut affirmer que le sens dans lequel on vérifie la proposition n'a pas d'importance, puisque la proposition sera vraie, à partir du moment où *tous* les éléments qui la composent seront vrais. On comprend aisément qu'écrire $A \wedge (B \wedge C)$ est équivalent à $(A \wedge B) \wedge C$, ou même mieux à $A \wedge B \wedge C$. Nous pouvons donc supprimer les parenthèses, puisqu'elles n'ont plus aucune utilité de priorité ici. De plus le "et" devient implicite, puisque c'est le seul connecteur qui puisse relier les éléments. Il devient lui aussi inutile. Ceci permet de simplifier encore notre formule, en la notant sous forme d'une *collection*, où l'ordre n'a plus d'importance, notée (A, B, C). Toute formule de la collection est appelée *clause*, dans le calcul des prédicats. Une clause correspond, par définition et comme nous l'avons constaté précédemment, à un littéral ou à un ensemble de littéraux reliés uniquement par des "ou". En suivant la même démarche que pour le connecteur "et" dans les collections, on s'aperçoit que si les clauses contiennent des "ou", ceux-ci deviennent implicites puisque ce sont les seuls utilisables ici. De plus les parenthèses sont inutiles, puisque l'ordre dans lequel on interprète la formule n'a aucune importance. Il suffit de trouver un littéral évalué à vrai pour que la proposition soit vérifiée. Dans le même souci de simplification et de clarification de la formule, on représentera les clauses par une collection de littéraux, implicitement disjoints. Ainsi, $\forall A \vee ((B \vee C) \vee (D \vee \forall E))$ correspond à la collection de littéraux ($\forall$A, B, C, D, $\forall$ E).

Nous voici arrivés à la mise sous forme clausale d'une formule initiale. Celle-ci est composée d'une collection de clauses, formées elles-mêmes d'une collection de littéraux. On rappelle qu'un littéral est soit une proposition atomique, soit la négation d'une proposition atomique. Si la formulation de la proposition initiale a été considérablement simplifiée et clarifiée, il n'empêche qu'elle n'a rien perdu de son interprétation et de sa signification, à partir du moment où nous nous souvenons des conventions qui ont été utilisées pour aboutir à cette mise sous forme clausale.

En guise de conclusion, essayons de suivre en totalité, la démarche de transformation en forme clausale, à partir d'un exemple :

($\forall$ x, ($\forall$ y, ÉTUDIANT (y,x) => AIME (y,logique)) => BONPROF (x))

dit que si tous les étudiants de quelqu'un aiment la logique, alors cette personne est un bon professeur.

- dans un premier temps, supprimons les implications présentes :
  ($\forall$ x, ¬ ($\forall$ y, ¬ ÉTUDIANT (y,x) v AIME (y,logique)) v BONPROF (x)).
- élargissons la négation à l'intérieur de la proposition composée :
  ($\forall$ x, ($\exists$ y, ÉTUDIANT (y,x) $\wedge$ ¬ AIME (y,logique)) v BONPROF (x)).

Désormais, la proposition n'est composée que de littéraux.

— skolemisation ou élimination du quantificateur existentiel. Nous sommes obligés d'introduire une fonction e1 (x) associée à la variable x, pour dire que ce qui existe dépend du quantificateur universel.

(∃ x, (ÉTUDIANT (e1 (x), x) ∧ ¬ AIME (e1 (x),logique)) v BONPROF (x)).

— déplacement du "∃" vers l'extérieur. Ici, on l'élimine simplement puisqu'il est déjà en tête :

(ÉTUDIANT (e1 (x),x) ∧ ¬ AIME (e1 (x),logique)) v BONPROF (x)).

— l'étape suivante permet de mettre la formule sous forme *normale conjonctive* (c'est-à-dire de la représenter par une succession de "∧" entre littéraux, ou groupes de littéraux reliés uniquement par des "v") :

(ÉTUDIANT (e1 (x),x) v BONPROF (x)) ∧ (¬ AIME (e1 (x), logique) v BONPROF (x)).

— l'étape finale conduit à la construction de deux clauses, formées chacune de deux littéraux :

ÉTUDIANT (e1 (x),x), BONPROF (x).
¬ AIME (e1 (x),logique), BONPROF (x).

- **UNE FORMALISATION POUR LES CLAUSES.**

Le mieux serait désormais d'aboutir à une normalisation d'écriture des formes clausales. On a vu qu'une forme clausale est composée d'une collection non ordonnée de clauses. On choisit de la représenter en écrivant chaque clause l'une après l'autre, dans n'importe quel ordre (puisque celui-ci n'a pas d'importance).

Allons un peu plus loin maintenant. Une clause est constituée d'une collection de littéraux, qui sont soit des propositions atomiques, soit des négations de propositions atomiques. Jusqu'à maintenant, on ne les a pas différenciés.

En vus d'une normalisation, on adoptera les conventions suivantes :

— on choisit de séparer et de différencier les littéraux "positifs" des littéraux "négatifs" ;

— les littéraux positifs, s'ils existent, seront écrits en tête de la clause, séparés les uns des autres par des points-virgules ;

— le séparateur entre les littéraux positifs et négatifs est le ":" ;

— les littéraux négatifs, s'ils existent, seront placés après le séparateur ":". Ils seront écrits sans le "¬" (puisque leur emplacement détermine leur "signe"), et seront séparés par des virgules ;

— la clause se termine par un point.

Ainsi pour l'exemple précédent, la formule clausale sera représentée par :

ÉTUDIANT (e1 (x),x) ; BONPROF (x) :.
BONPROF (x) : AIME (e1 (x),logique).

Vous vous demandez peut-être quel est l'impact des formes clausales sur le programmation en PROLOG. Pour le comprendre, il faut se rappeler que PROLOG s'inspire du Principe de Résolution ou méthode de démonstration automatique de thérorèmes.

3.2.2 - Démonstration de Théorèmes : le Principe de Résolution

Lorsqu'on dispose d'une série de propositions, que peut-on et que veut-on en faire? Notre but est de pouvoir en déduire des conséquences qui peuvent être intéressantes, c'est-à-dire de nouvelles propositions.

Par convention, les propositions posées comme vraies au départ seront appelées *axiomes* ou *hypothèses*, les propositions que nous pourrons en déduire seront appelées *théorèmes*.

La même théorie est utilisée en Mathématiques où, à partir d'hypothèses bien choisies, on parvient à déduire des théorèmes, c'est-à-dire de nouvelles propositions vérifiées.

Sans entrer dans les détails (ce n'est pas notre but ici) traçons les grandes lignes des principes de la démonstration de théorèmes, et surtout du Principe de Résolution, proposé par J.A. Robinson.

● De tout temps, on s'est intéressé au problème de résolution et de démonstration de théorèmes. Cette démarche devenant de plus en plus "automatique", de nombreuses recherches se sont orientées, dès les années 1960, sur la possibilité de programmer donc d'automatiser ce processus.

● Une des découvertes importantes de l'époque fut sans aucun doute l'énoncé du *Principe de Résolution*, permettant de mécaniser cette démarche : en choisissant correctement les axiomes que l'on veut utiliser, on peut prouver des théorèmes automatiquement.

● Le processus de résolution, quand on peut l'appliquer (c'est-à-dire quand les axiomes sont pertinents), est basé sur l'utilisation de la forme **clausale** des propositions. Il permet, à partir d'une partie de clauses dites "parents", de déduire une clause dérivée. Ainsi, avant d'utiliser tout principe de résolution, il faut mettre les propositions, à condition qu'elles soient bien formées, donc qu'elles aient une syntaxe correcte (Well Formed Formules: WFF), sous leur forme clausale.

● Le principe de résolution est basé sur plusieurs propriétés, que nous allons énoncer brièvement (nous aurons l'occasion d'y revenir).

— **Le processus d'unification :** Ce processus consiste à essayer de faire coïncider, sur un ensemble de clauses, un ou plusieurs littéraux de chaque clause.

Soient deux clauses. Si une même formule atomique figure simultanément dans la partie gauche d'une des clauses et dans la partie

droite de l'autre, on peut en dériver une nouvelle clause, par assemblage des deux clauses initiales et élimination des formules atomiques dupliquées (on sait que ces formules sont la négociation l'une de l'autre, puisqu'elles sont placées d'un côté et de l'autre du séparateur " : ", donc on peut les éliminer).

Exemples :
Des deux clauses :

A ; B ; C : D, E, H
D ; F ; G : A, B

On en déduit une nouvelle (par élimination des formules atomiques dupliquées A, B et D) :

C ; F ; G : E, H.

On peut appliquer l'unification à plusieurs clauses, et cela en procédant simultanément, par paires de clauses.

Soient les clauses :

A ; B
B : C
C : D

a) On déduit d'abord, des deux premières clauses, une clause dérivée : A : C.
b) En unifiant la clause dérivée à la troisième clause initiale, on en déduit la clause : A : D.

Nous ne sommes pas ici entrés dans la signification des littéraux employés et des termes qu'ils contiennent. Évidemment, suivant le type de termes utilisé, l'unification est plus ou moins facile à traiter ; en particulier lorsque les clauses possèdent des symboles variables. Nous reviendrons sur ce traitement dans la partie pratique relative à la programmation en PROLOG.

— **Le processus de démonstration par l'absurde :** Si nous nous contentions d'utiliser le processus vu ci-dessus, nous arriverions peut-être rapidement à la conclusion que ce Principe de Résolution n'est pas assez performant. En effet, il se peut que nous arrivions au terme des unifications possibles, et que le mécanisme de déduction en soit stoppé, alors qu'il s'avère, de toute évidence, qu'on pourrait tout de même en déduire une conséquence.

Une autre propriété de la Résolution consiste à permettre la reformulation de notre problème, afin d'en garantir la résolution. Elle est basée sur le fait qu'une résolution est toujours possible.

ATTENTION :

CELA NE VEUT PAS DIRE QUE DE TOUTE HYPOTHÈSE, ON POURRA TOUJOURS DÉDUIRE UNE CONSÉQUENCE.
MAIS, AU CONTRAIRE, QUE SI ON NE PEUT TIRER AUCUNE CONCLUSION, AUCUNE INTERPRÉTATION D'UN ENSEMBLE DE FORMULES, APPELÉES ALORS FORMULES INCONSISTANTES, ALORS LA RÉSOLUTION SERA CAPABLE DE NOUS LE SIGNIFIER, EN NOUS RENVOYANT COMME CONSÉQUENCE LA CLAUSE VIDE, QUI EST L'EXPRESSION LOGIQUE MÊME DE LA CONTRADICTION (ELLE REPRÉSENTE UNE PROPOSITION QUI NE PEUT ÊTRE VRAIE).

Ainsi, supposons que nous voulions prouver une proposition. Le principe consiste à rajouter aux hypothèses (après s'être assuré qu'elles sont consistantes, c'est-à-dire qu'elles permettent d'en déduire notre proposition), les clauses correspondant à la négation de ce que l'on veut prouver. La résolution en déduira la clause vide, puisque notre proposition découle des hypothèses et que nous avons introduit la contradiction dans l'énoncé.

Par définition, la Résolution déduit la clause vide si et seulement si la proposition que nous voulons démontrer, et que nous avons énoncé par sa négation, découle des hypothèses consistantes.

C'est ce qu'on appelle en Mathématiques *le raisonnement et la démonstration par l'absurde*. On part du principe que ce que l'on veut démontrer est faux et on aboutit à une contradiction.

— **Les clauses de Horn :** Le seul inconvénient de la Résolution, à cette phase, est qu'elle ne nous dit pas comment elle choisit ses hypothèses, ni comment et sur quoi elle pratique le principe d'unification pour aboutir à une déduction. Or, pour pouvoir utiliser ce principe de façon automatique, il faut aboutir à une formalisation unique, simple et utilisable. De nouvelles recherches ont été menées pour affiner le Principe de Résolution avec en particulier l'énoncé des clauses de Horn, qui est, on le verra, à la base de la syntaxe de PROLOG.

Par définition, une *clause de Horn* est une clause qui possède *au plus* un littéral positif. Ces clauses sont donc relativement simples et débouchent sur un système effectivement programmable de "démontreur de théorèmes".

Les clauses de Horn sont de deux types :

- soit elles ont un littéral positif et un seul. On les appellera clauses de Horn *avec tête*,
- soit elles n'en ont aucun. On les appellera les clauses de Horn *sans tête*.

Exemples :

MERE(x,y) : FEMME(x), PARENT(x,y) est une clause avec tête
: MERE(x,y) est une clause sans tête.

3.2.3 - Rapport entre PROLOG et la logique

Ce chapitre peut vous avoir semblé complexe, mais il était nécessaire de l'introduire, pour se faire une idée de la logique et de l'esprit PROLOG. Nous vous demanderons simplement de retenir les points suivants qui rappellent les rapports qui peuvent exister entre PROLOG et la logique :

— la syntaxe de base de PROLOG s'inspire de celle qui est utilisée en calcul des prédicats ;

— résoudre un problème en PROLOG, c'est utiliser un "démontreur de théorèmes", qui permet, à partir d'hypothèses énoncées, c'est-à-dire de connaissances vérifiées sur un domaine :
- de poser des questions à partir des hypothèses préétablies,
- d'en tirer des conclusions intéressantes ;

— les propositions énoncées en PROLOG sont très proches de leurs formulations sous forme de clauses de Horn. Les hypothèses correspondent en fait aux clauses de Horn avec tête, et le but à atteindre (un seul à la fois) à la clause sans tête ;

— un système PROLOG est donc basé, plus précisément sur un démontreur de théorèmes par Résolution pour clauses de Horn. Nous verrons ultérieurement cette correspondance dans les étapes d'exécution d'un programme en PROLOG.

PROLOG N'EST-IL QUE LOGIQUE ?

Il est indéniable que non, et cela pour le plus grand confort de l'utilisateur !

En effet, il serait peut pratique de ne pouvoir travailler que par Résolution, alors qu'à certains moments, nous pouvons avoir à effectuer des contrôles, traiter plus spécialement des termes ou des clauses, effectuer des actions dans le but d'améliorer la gestion des programmes. La logique nous permet de formaliser un raisonnement, nous offre un système de programmation clair, concret et déclaratif. Mais, à certains moments, comme nous l'avons vu, nous devrons agir sur cette logique, pour effectuer des traitements spécifiques.

PROLOG, conscient de cette "dualité", a été conçu pour prendre en compte ces constats. Ainsi, si sa syntaxe est claire et déclarative, comme le veut la logique, PROLOG permet aussi des traitements plus impératifs.

Après cette introduction au nouveau monde de l'intelligence Artificielle, et à sa formalisation en PROLOG, nous voilà prêts pour la programmation en PROLOG.

III - PROGRAMMER EN PROLOG

1 - MISE EN ROUTE DU SYSTÈME 45

1.1 - Branchement du matériel 45

1.2 - Démarrer PROLOG 45

1.3 - Modes de fonctionnement 46

1.3.1 - Taper une commande 46
1.3.2 - Énoncer ou saisir une clause 54
1.3.3 - Lancer un programme 60

1.4 - La gestion de fichiers de programmes 60

1.4.1 - Principes 60
1.4.2 - Sauvegarder un fichier 61
1.4.3 - Lire un fichier 62

2 - PREMIÈRE RENCONTRE AVEC PROLOG ... 64

2.1 - Les faits 64

2.2 - Les variables 68

2.3 - Les conjonctions 71

2.4 - Les règles 77

3 - COMMENT PROGRAMMER EN PROLOG ? . 83

3.1 - Les principes de base : les éléments de la syntaxe et leur représentation en mémoire 83

3.1.1 - Les termes 83
3.1.1.1 - Les constantes 84
3.1.1.2 - Les variables 85
3.1.1.3 - Les termes composés ou structures 86
3.1.1.4 - Les textes 93

3.1.2 - Les programmes 95
3.1.2.1 - Les prédicats 95
3.1.2.2 - Les clauses 96
3.1.2.3 - Les sous-programmes 100
3.1.2.4 - Les prédicats prédéfinis ou primitives 108

3.2 - Résolvante de programme :
les étapes de résolution 130

4 - MANUEL DE RÉFÉRENCES 143

PRÉFACE

Ce logiciel a été conçu pour vous initier à la programmation en PROLOG. Ainsi, vous allez pouvoir programmer et tester vos propres applications.

● Mais sachez qu'avant cela, vous allez devoir, si ce n'est déjà fait, comprendre les principes et les éléments de base de la programmation en PROLOG, et en assimiler les conventions et l'esprit.

En effet PROLOG, comme tout langage, possède son vocabulaire et sa grammaire, et l'on ne peut prétendre connaître un langage sans avoir, au préalable, acquis ces éléments de base. Même ici, la programmation ne s'improvise pas.

Une fois cette opération effectuée et assimilée, la programmation en PROLOG ne posera normalement pas de problèmes majeurs, puisqu'elle se réfère à un traitement de connaissances, dans une démarche qui a le mérite de nous être familière.

● Ce préambule s'avère d'autant plus nécessaire que, lorsque vous allez mettre en route le système, vous allez vous retrouver directement maître du clavier et des commandes.

PROLOG est un langage *conversationnel*, c'est-à-dire qu'il permet d'établir un certain type de dialogue entre l'ordinateur et vous.

Vous utiliserez le clavier pour envoyer vos "connaissances" à l'ordinateur qui vous répondra en vous affichant les résultats de vos demandes par l'intermédiaire de l'écran (ou de l'imprimante, si vous l'utilisez).

PROLOG va attendre que vous lui énonciez, de façon formelle, le problème que vous voulez résoudre (en lui déclarant toutes les connaissances nécessaires à la résolution du problème) et que vous lui posiez les "bonnes" questions, pour trouver et afficher les réponses correspondantes.

N'ayez pas d'inquiétude, en suivant le chapitre d'utilisation de PROLOG, vous vous apercevrez que chaque nouvelle notion que vous rencontrerez est accompagnée d'un ou de plusieurs exemples, que vous pourrez tester sur votre ordinateur. Petit à petit, les exemples seront plus conséquents, pour aboutir progressivement à la programmation totale d'une problématique. Alors, vous pourrez programmer vos propres applications.

● L'articulation modulaire et indépendante de ce chapitre doit vous aider dans cette démarche :

— le premier chapitre "Mise en route du système" énonce tout ce qu'il faut savoir avant d'utiliser PROLOG, (tant au niveau branche-

ment du matériel. qu'au niveau de l'utilisation du clavier et des divers périphériques), vous permettant une utilisation optimale de l'environnement de PROLOG sur TO7, TO7/70, TO9, MO5 ;

— le chapitre intitulé "Première rencontre avec PROLOG" est plus particulièrement destiné aux utilisateurs qui n'ont jamais programmé dans ce langage. Il introduit de façon progressive les principes du langage PROLOG ;

— le chapitre intitulé "Comment programmer en PROLOG " définit plus explicitement les principes de base, la syntaxe et la représentation en mémoire des éléments du langage. Chaque principe et chaque nouvelle notion seront accompagnés d'exemples. Ces définitions vous permettront d'assimiler de façon progressive les notions de termes (éléments de base du langage ou "mots"), les notions de prédicats, clauses, prédicats prédéfinis, permettant de composer des "phrases" à partir des termes du langage, et regroupées pour former un programme, puis enfin la notion de résolution ;

— le chapitre intitulé "Manuel de références" regroupe toutes les commandes, tous les prédicats prédéfinis PROLOG et leur syntaxe, classés par ordre alphabétique. Ce chapitre permet aux programmeurs avertis de retrouver rapidement une définition ou une syntaxe.

1 - MISE EN ROUTE DU SYSTÈME

1.1 - BRANCHEMENT DU MATÉRIEL

Si vous venez d'acquérir votre micro-ordinateur, quelques précisions sur sa mise en place et les branchements à effectuer vous seront peut-être nécessaires.

Regardez bien votre micro-ordinateur. Il se présente comme un clavier de machine à écrire avec, sur ses côtés et à l'arrière, diverses prises. Quel que soit le modèle dont vous disposez, le principe d'installation est le même.

Examinons tout d'abord, les connexions dont il dispose, et les opérations à effectuer :

— relié directement au micro-ordinateur, un câble de liaison, muni d'une prise PERITEL. Ce câble sera branché directement sur votre téléviseur, si celui-ci possède également une prise PERITEL. Dans le cas contraire, renseignez-vous auprès de votre fournisseur pour savoir comment l'adapter ;

— une prise pour l'alimentation en courant électrique continu de votre ordinateur. Pour le MO5, cette prise, possédant un adaptateur, sera branchée à la droite du câble de liaison PERITEL.

A ce stade, vous possédez la configuration de base nécessaire au bon fonctionnement de votre micro-ordinateur. Cependant, vous avez certainement remarqué la présence d'autres prises. Elles vont vous permettre d'améliorer votre configuration selon vos besoins, en autorisant en particulier l'utilisation de nombreux périphériques d'Entrée/Sortie :

— une prise pour la connexion d'un magnétophone à cassettes, permettant de sauvegarder et de charger des programmes sur cassettes. Le magnétophone devient nécessaire, lorsque l'on compte programmer un peu sérieusement ;

— une cavité permettant d'introduire des cartouches de mémoire morte (ROM), contenant des logiciels variés (langage, jeu, ...) ;

— des sorties extension permettant en particulier de raccorder une imprimante, pour avoir des "traces-papier" de vos travaux, d'utiliser une extension mémoire (nécessaire sur TO7) ou de connecter un lecteur de disquettes. Celles-ci se trouvent à l'arrière de l'appareil.

1.2 - DÉMARRER PROLOG

Le logiciel PROLOG, que vous venez d'acquérir, se présente sous la forme d'une cartouche.

Une fois que vous avez vérifié que les branchements entre les différents composants étaient bien établis,vous pouvez démarrer PROLOG, en effectuant la démarche suivante :

— insérez la cartouche PROLOG dans la cavité prévue à cet effet.

— allumez votre téléviseur. Si vous utilisez un adaptateur PERITEL, suivez les consignes du fournisseurs pour vous placer sur le bon canal de télévision ;

— allumez ensuite votre micro-ordinateur :
- sur TO7, TO7/70, une page d'en-tête s'affiche : appuyez sur la touche **1**.
- sur TO9, une page d'en-tête s'affiche : appuyez sur la touche **0**.
- lorsque le logiciel PROLOG est lancé, le signe "$" est affiché en haut de l'écran, l'ordinateur est prêt à programmer avec vous en PROLOG.

Il est conseillé de faire cette manipulation (allumer le téléviseur *puis* le micro-ordinateur) toujours dans cet ordre. Inversement, lorsque vous aurez fini de travailler sur votre micro-ordinateur, commencez par éteindre celui-ci, puis ensuite seulement la télévision.

1.3 - MODES DE FONCTIONNEMENT

Programmer en PROLOG permet de traiter des connaissances : les énoncer sous forme de *clauses*, mais aussi essayer de les utiliser dans une méthode de *résolution*, afin d'en déduire de nouvelles connaissances.

C'est également permettre à l'utilisateur d'avoir une action un peu plus impérative, lorsqu'il veut, par exemple, donner des *commandes* qui ne sont pas spécialement logiques (voir § 3.2.3, p. 40). Ainsi, il peut vouloir lire un programme stocké sur cassette ou sur disquette, lister un programme, modifier une des connaissances, donc une des saisies, qu'il a déclarées : autant d'opérations des plus classiques en informatique.

On distingue trois modes de fonctionnement :

— donner des commandes impératives ;

— énoncer ou saisir une clause (donc déclarer au système ses connaissances sur un domaine) ;

— lancer un programme (ou demander une résolution).

PROLOG vous permet ainsi de choisir votre mode de travail, selon vos besoins.

1.3.1 - Taper une commande

Une commande permet, en PROLOG, de lire ou sauver des fichiers sur cassettes ou sur disquettes, de détruire ou modifier une ou

plusieurs clauses stockées en mémoire, de modifier certains paramètres de fonctionnement, etc. Bref, elles obligent l'interpréteur à effectuer certaines actions, dans le but d'améliorer la gestion des programmes.

En pratique, pour taper une commande, il faut que le curseur se trouve en début de ligne, précédé du symbole "$". Cela veut dire que l'interpréteur attend une action de votre part. C'est dans cette situation que vous vous trouvez, lorsque vous démarrez PROLOG. C'est le **mode commande.**

● Le mode commande travaille en *mode "insertion"*, c'est-à-dire que toute touche affichable (lettres, chiffres, caractères spéciaux) est affichée à l'emplacement du curseur, avec éventuellement déplacement de la lettre qui se trouvait à l'emplacement du curseur et de toutes les suivantes, d'une position vers la droite (à condition que la ligne ne soit pas déjà pleine, comme nous allons le voir).

● Le mode commande travaille *par ligne*, c'est-à-dire qu'il ne prend en compte que les 39 premiers caractères tapés (sachant qu'une ligne contient 40 caractères et que le premier est occupé par le symbole "$"). En fin de ligne, le curseur est stoppé et vous ne pouvez aller plus loin (c'est-à-dire continuer en passant à la ligne suivante).

Par contre, vous pouvez vous déplacer sur la ligne, en actionnant les touches du clavier suivantes :

— **Flèche gauche** ← : le curseur se déplace d'une position vers la gauche, sans détruire le caractère sur lequel il se trouve.

— **Flèche droite** → : effet identique au précédent, mais le curseur se déplace vers la droite.

— **EFF** EFF : effacement du caractère sur lequel est positionné le curseur. Tous les caractères qui se trouvaient après sont décalés d'une position vers la gauche.

— **RS (flèche recourbée)** : effacement du caractère qui se trouve avant la position du curseur. Tous les caractères suivants sont décalés d'une position vers la gauche.

Mais rappelez-vous que les modifications ne sont prises en compte que sur les 39 premiers caractères. En particulier, si la ligne est déjà remplie, vous ne pourrez y insérer directement un caractère. Il vous faudra d'abord en effacer un autre, pour lui faire une place.

Ces définitions ou "contraintes" n'ont en fait pas grande importance ici, puisque les commandes sont en général, courtes à énoncer. Mais nous verrons qu'elles peuvent être très gênantes lorsqu'on utilise ce mode pour d'autres actions. Mais ne vous inquiétez pas : PROLOG a plus d'un tour dans son sac !

Une commande est validée par appui sur la touche ENTREE ou RETURN. Dès lors, l'action demandée est exécutée par l'ordinateur, jusqu'à ce que réapparaisse à l'écran le symbole "$". Il se peut que le résultat ne soit pas celui attendu. Soit vous avez fait une simple faute de vocabulaire dans l'énoncé de la commande (la

conséquence sera expliquée dans le paragraphe suivant), soit vous avez mal géré les paramètres de la commande (signalé par un message d'erreur). Dans tous les cas, pour modifier la commande, il faut la retaper, vu les principes énoncés ci-dessus.

Les commandes doivent être saisies en majuscules. Les commandes disponibles sur PROLOG et leur syntaxe sont les suivantes :

- **COMMANDES DE GESTION DES FICHIERS DE PROGRAMMES** (voir § 1.4, p. 60).

- **COMMANDES DE MANIPULATION ET D'ÉDITION DU PROGRAMME EN MÉMOIRE**

CLEAR
Efface tout le programme qui se trouve en mémoire (donc toutes les clauses que vous aviez saisies). Réinitialise l'interpréteur PROLOG.

CLS ou **CLS (<couleur des caractères>, <couleur de fond>)**
Efface l'écran, sans pour autant le faire disparaître de la mémoire et éventuellement change les couleurs de caractères et de fond, qui sont données sous forme de nombre entre 0 et 7, selon la convention habituelle sur TO7, MO5.

DEL (<nom-d'un-paquet-de-clauses>)
Détruit toutes les clauses d'un paquet (ou sous-programme) dont le nom est donné en paramètre (visualisé par la commande STAT : le nombre de clauses mémorisées est donc diminué). La commande sera plus ou moins longue à s'effectuer selon la taille du paquet.

Exemple :
DEL(PERE) : détruit toutes les clauses de nom PERE.

DEL (<nom-d'un-paquet-de-clauses>, <rang-de-la-clause>)
Détruit la clause d'un paquet (dont le nom est donné en premier paramètre) et dont le rang dans le paquet est indiqué en deuxième paramètre (correspondant, lors de l'action de la commande LIST, au numéro placé devant la clause entre les deux signes "=").

Exemple :
DEL(PERE, 5) : détruit la 5e clause de nom "PERE".

DOFF
Supprime l'affichage des solutions trouvées pour une résolution, permettant à l'utilisateur de formater (par programme) leur présentation à sa convenance.

DON
Rétablit l'affichage systématique des solutions.

ED, ED (<nom-du-paquet-de-clauses>), CH (<nom-du-paquet-de-clauses>) : ces commandes spécifiques seront examinées dans le paragraphe suivant.

LIST
Liste à l'écran la totalité du programme en mémoire. Chaque clause mémorisée est précédée d'un numéro entre deux signes "-" indiquant son rang dans le paquet, et d'un nombre entre <> indiquant le nombre de termes qu'elle occupe en mémoire. L'appui sur n'importe quelle touche du clavier provoque l'arrêt momentané du défilement (pour reprendre, répéter cette manipulation). L'appui sur la touche **RAZ** provoque l'abandon de la commande.

LIST (<nom-d'un-paquet-de-clauses>)
Ne liste que les clauses du paquet dont le nom est donné en paramètre.

Exemple :

LIST(PERE) : liste toutes les clauses de nom "PERE".

PRINTON
Demande à éditer sur l'imprimante tout ce qui sera ultérieurement affiché à l'écran, que ce soit le résultat d'une commande ou l'exécution d''un programme.

PRINTOFF
Annule l'effet de la commande précédente.

TRON
Permet de suivre le déroulement des résolutions ultérieures, en affichant les différentes évaluations successives des clauses à évaluer (mode trace). L'appui sur n'importe quelle touche du clavier provoque l'arrêt momentané de la trace (pour la reprendre, répéter cete opération). L'appui sur la touche **RAZ** fait sortir du mode trace (la résolution continue en mode normal). Un nouvel appui sur la touche **RAZ** provoque l'abandon de la résolution.

TROFF
Annule l'effet de TRON.

• COMMANDES D'EXAMEN DES STRUCTURES MÉMOIRE ET DE LEUR OCCUPATION

L'interpréteur gère un ensemble de structures mémoire qui ont deux fonctions principales :

— représenter le programme sous forme interne, par des structures "statiques",
— permettre de faire une résolution, par des structures "dynamiques".

Ces structures se comportent comme des piles (fonctionnant par empilement successifs, à chaque nouvelle clause saisie).

L'ensemble de ces structures est accessible par la commande STAT, qui vous permet de visualiser :

— d'une part la taille de chacune de ces structures (-1-MÉMOIRE). La rubrique MÉMOIRE donne l'adresse (en hexadécimal) du premier et du dernier octet de chaque structure, dont l'occupation

maximale de chaque zone. Elle a une capacité totale et maximale de 32 Ko (nécessitant une extension mémoire sur TO7) ;

— d'autre part la place mémoire occupée par votre programme, dans les structures précitées, c'est-à-dire le nombre d'éléments occupés sur le nombre total de libres (-2-OCCUPATION). Le nombre d'éléments par structure s'obtient en divisant la taille de la structure (en octets) par la taille d'un élément (voir ci-dessous). Par définition, quand il n'y a pas de programme en mémoire, l'occupation des différentes structures est de 0, sauf pour les termes où il y a 4 termes stockés initialement.

La commande STAT lance un "garbage" (récupérateur mémoire) sur les TERMES et le TEXTE. Si vous venez de détruire des clauses de votre programme, la mémoire correspondante est récupérée : STAT annonce désormais le nombre exact de termes, et le nombre exact (si le texte détruit se trouvait au sommet de la pile) ou maximum de texte présent.

On distingue donc :

— **les structures "statiques"** qui contiennent le programme :

- le dictionnaire contenant tous les noms de prédicats ou d'atomes créés par l'utilisateur, mémorisés sur 13 octets (DICO),
- la "pile" des clauses : chaque clause est associée à un élément de cette pile (occupant 5 octets) qui représente l'ensemble de la clause (CLAUSES),
- la "pile" des termes (TERMES) : chaque terme composant d'une clause y est mémorisé (sur 4 octets) sauf les termes particuliers de type "texte",
- la "pile" des textes : chaque terme de type "texte" d'une clause y est mémorisée (TEXTE).

— **Les structures "dynamiques"** : utilisées lors de la résolution (qui doivent occuper au moins 3 Ko) :

- la "pile" des nœuds de résolution, dont chaque élément occupe 12 octets (NŒUDS),
- la "pile" des substitutions, dont chaque élement occupe en moyenne 7 octets, suivant le nombre de variables instanciées (PILE),
- le compteur d'unification (sur 16 bits) donnant nombre d'unifications réussies sur le nombre total essayé, lors de la dernière résolution. Il est donc automatiquement remis à zéro lorsqu'il atteint 65535 (2^{16}-1).

Toutes ces structures et leur contenu seront plus amplement détaillées dans le paragraphe sur les éléments de la syntaxe PROLOG et leur représentation en mémoire (§ 3.1.2.3). Les piles sont considérées comme pleines lorsqu'elles ne possèdent plus qu'un élément de libre (entraînant une erreur en mémoire).

Exemple d'effet de la commande STAT, après mise sous tension du matériel (il n'y a donc pas de programme en mémoire). Les adresses mémoire diffèrent suivant la machine.

Sur TO7/70 :

```
$ STAT
STATISTIQUES
-1-MÉMOIRE
DICO 6500 A 74FF
CLAUSES 7500 A 7CFF
TERMES 7D00 A C0FF
TEXTE C 100 A C4FF
NŒUDS C500 A CCFF
PILE CD00 A DFFF
-2-OCCUPATION
DICO 0 SUR 315
CLAUSES 0 SUR 409
TERMES 4 SUR 4352
TEXTE 0 SUR 1024
NŒUDS 0 SUR 170
PILE 0 SUR 694
-3-
UNIFICATION : 0 SUR 0
```

Sur MO5 :

```
$ STAT
STATISTIQUES
-1-MÉMOIRE
DICO 2500 A 34FF (4096 octets ou 4 Ko)
CLAUSES 3500 A 3CFF (2048 octets ou 2 Ko)
TERMES 3D00 A 80FF (17 Ko)
TEXTE 8100 A 84FF (1024 octets ou 2 Ko)
NŒUDS 8500 A 8CFF (2048 octets ou 2 Ko)
PILE 8D00 A 9FFF
-2-OCCUPATION
DICO 0 SUR 315
CLAUSES 0 SUR 409
TERMES 4 SUR 4352
TEXTE 0 SUR 1024
NŒUDS 0 SUR 170
PILE 0 SUR 694
-3-
UNIFICATION : 0 SUR 0
```

DIC

Permet de lister à l'écran, le contenu exact du dictionnaire, donc tous les noms d'atomes et de prédicats définis par l'utilisateur. L'appui sur toute touche du clavier arrête momentanément le défilement (pour reprendre, répéter cette opération). L'appui sur la touche **RAZ** provoque l'abandon de la commande.

Les noms sont listés dans leur ordre d'apparition dans le programme et suivis du nombre de fois où ils apparaissent, et d'un caractère spécifiant leur type (C pour nom de tête de clause, A pour atome, un nom de prédicat ou de structure, V pour affectation d'une valeur à un atome).

SIZE (N1, N2, N3, N4, N5)

Nous avons remarqué que la rubrique "MÉMOIRE" donnait la taille mémoire maximum des structures. Cela veut dire, en particulier, que si un programme est trop important, il ne pourra peut-être pas être mémorisé intégralement, si sa taille dépasse les limites. La commande SIZE (N1, N2, N3, N4, N5) permet d'"étendre" par structure la taille mémoire initialement prévue.

Mais attention : elle a aussi pour effet de remettre à zéro toute la mémoire (donc par définition d'**effacer le programme** qui pouvait s'y trouver). Il est donc recommandé d'effectuer cette manipulation au début, avant la saisie du programme et en fonction de la taille supposée de celui-ci.

La commande SIZE comprend 5 paramètres obligatoires N1, N2... N5, qui sont des entiers donnant le nombre de blocs de 256 octets alloués à chacune des structures indiquées, selon la correspondance suivante :

- N1 : taille du dictionnaire (sachant qu'un nom du dictionnaire occupe 13 octets) ;
- N2 : taille de la pile de clauses, qui donne le nombre maximum de clauses stockables (sachant qu'une clause occupe 5 octets) ;
- N3 : taille de la pile des termes, qui donne le nombre maximum de termes du programme (sachant qu'un terme occupe 4 octets) ;
- N4 : taille de la zone de stockage des textes (sachant qu'un texte occupe un terme par caractère / un terme pour le texte). Si on veut gagner de la place en mémoire et si le programme ne contient pas de texte, on donne à N4 la valeur 0 ;
- N5 : taille de la zone des nœuds de résolution (sachant qu'un nœud occupe 12 octets).

Le reste de la mémoire est alloué à la pile.

La limite de l'extension mémoire par SIZE est fixée par SIZE (20, 20, 20, 50,1).

Exemple :

SIZE (16, 16, 64, 4, 16)

alloue 4096 octets (4 Ko) au dictionaire, 4096 octets (4 Ko) à la pile des clauses, 16384 octets (16 Ko) aux termes, 1024 octets (1 Ko) pour les textes et 4096 octets (4 Ko) pour les nœuds de résolution.

En réactionnant la commande STAT, vous verrez que votre demande a bien été prise en compte :

Sur TO7 :

```
$ STAT
STATISTIQUES
-1-MÉMOIRE
DICO 6500 A 74 IFF
CLAUSES 7500 A 84FF
TERMES 8500 A C4FF
TEXTE C500 A C8FF
NŒUDS C900 A D8FF
PILE D900 A DFFF
-2-OCCUPATION
DICO 0 SUR 315
CLAUSES 0 SUR 819
TERMES 4 SUR 4096
TEXTE 0 SUR 1024
NŒUDS 0 SUR 341
PILE 0 SUR 256
-3-
UNIFICATION : 0 SUR 0
```

Sur MO5 :

```
$ STAT
STATISTIQUES
-1-MÉMOIRE
DICO 2500 A 34FF
CLAUSES 3500 A 44FF
TERMES 4500 A 84FF
TEXTE 8500 A 88FF
NŒUDS 8900 A 98FF
PILE 9900 A 9FFF
-2-OCCUPATION
DICO 0 SUR 315
CLAUSES 0 SUR 819
TERMES 4 SUR 4096
TEXTE 0 SUR 1024
NŒUDS 0 SUR 341
PILE 0 SUR 256
-3-
UNIFICATION : 0 SUR 0
```

1.3.2 - Énoncer ou saisir une clause

Cette opération est à la base de la programmation en PROLOG. Nous verrons dans le chapitre suivant, comment définir et écrire des clauses, de façon à ce qu'elles correspondent à la syntaxe PROLOG. Ici, intéressons-nous plutôt à la façon dont on peut les "rentrer" dans l'ordinateur.

Une première méthode consiste à utiliser le **mode commande** vu ci-dessus : on tape directement la clause, à partir du curseur placé en début de ligne et précédé du symbole "$". Mais nous avons vu les limites de ce mode, particulièrement gênantes ici, vu que les clauses que nous énoncerons auront souvent plus de 39 caractères.

Une fois que la clause est validée par appui sur la touche **ENTRÉE**, elle est analysée par l'interpréteur. Si une erreur est constatée, un message est affiché et vous devez retaper intégralement la clause. Sinon, l'ordinateur vous indique la place qu'elle occupe en mémoire (voir § 3.1, p. 83) par la mention "X TERMES". Toute clause tapée selon ce mode est rajoutée au programme, à la fin du paquet de clauses du même nom.

Ce mode de fonctionnement pose en outre un autre petit problème d'ambiguïté, car il permet de taper soit une commande, soit une clause. En effet, partons du constat que la liste des commandes disponibles est figée et possède son vocabulaire propre. Nous verrons plus loin que les clauses, à partir du moment où elles sont syntaxiquement correctes, sont énoncées par une suite de mots, dont le sens n'est pas vérifié par l'interpréteur. Cela veut dire, en particulier, que si vous faites une faute d'orthographe dans la frappe d'une commande, celle-ci n'est pas reconnue en tant que telle par l'interpréteur. Par contre, si sa syntaxe est valide, l'interpréteur la considérera comme une nouvelle clause qu'il rajoutera au programme en mémoire.

Exemple :
Si on tape SAV ('O : X') pour SAVE ('O : X') (sauvegarde du programme en mémoire, sur disquette, dans un fichier de nom X), on crée en fait une nouvelle clause SAV ('O : X').

Dès qu'une erreur de manipulation de ce type est constatée, on a intérêt à détruire cette clause "parasite", par utilisation de la commande de destruction de clauses DEL (<nom-de-cause>).

- **LE MODE ÉDITEUR**

Il serait fastidieux (et parfois même impossible) d'écrire un programme PROLOG seulement en mode saisie de commandes qui, on l'a vu, fonctionne par ligne. Imaginez la peine que cela vous donnerait, surtout pour écrire de longs programmes et les mettre au point...

Comme nous l'avons annoncé, PROLOG a plus d'un tour dans son sac.

PROLOG possède un "éditeur-page" qui permet de saisir, modifier un programme ou une partie de programme de quelques lignes à quelques dizaines de lignes (la seule limite étant la taille mémoire disponible donnée par la commande STAT et modifiable par la commande SIZE (N1, N2, N3, N4, N5) avant la saisie du programme). Si on dépasse la limite, un message s'affiche. Utiliser l'éditeur, c'est — comme on va le voir — faciliter votre travail de saisie et de mise au point.

L'éditeur permet de saisir simultanément une ou plusieurs clauses. Il utilise pour cela les structures mémoire NŒUDS et PILE (utilisées normalement à la résolution). Il vous est conseillé de saisir les clauses par petits paquets, afin d'éviter le débordement de l'éditeur.

Celui-ci présente une "fenêtre" de 20 lignes, de couleur bleue. Le curseur est positionné dans le coin supérieur gauche de la fenêtre.

COMMENT APPELER L'ÉDITEUR ?

L'éditeur possède trois fonctions différentes appelées chacune par une commande propre :

- saisir une ou des nouvelles clauses, commande ED,
- modifier un paquet de clauses, tout en gardant l'ancienne version, commande ED (<nom-du-paquet-de-clauses>),
- modifier définitivement un paquet de clauses, commande CH (<nom-du-paquet-de-clauses>).

Une fois la fonction choisie, le principe d'utilisation de l'éditeur est le même.

COMMENT CHOISIR SA FONCTION ET POUR QUOI FAIRE ?

— **La commande ED :** Elle permet de rentrer simplement dans l'éditeur, sans référence à une clause ou un paquet de clauses précis. Une page vierge bleue, qui ne couvre pas tout l'écran, s'affiche. Le curseur est positionné dans le coin gauche et attend que vous saisissiez une ou plusieurs clauses.

— **La commande ED (<nom-d'un-paquet-de-clauses>) :** Sa formulation ressemble à la précédente, sauf qu'elle spécifie un paquet de clauses qui ont le même nom (le même prédicat). Elle a pour effet d'afficher la même fenêtre bleue que précédemment, mais remplie cette fois-ci des clauses demandées. Cette commande permet de créer un nouveau paquet de clauses à partir du paquet proposé, sans pour autant détruire l'ancienne version. Cela veut dire qu'en sortant de l'éditeur, les anciennes clauses restent inchangées, et que les nouvelles sont rajoutées au programme (ce que vous pouvez visualiser par la commande LIST). Cela permet de faciliter la saisie des programmes, lorsque des paquets de clauses peuvent se ressembler.

— **La commande CH (<nom-d'un-paquet-de-clauses>) :** Cette commande permet la modification d'un paquet de clauses en le remplaçant par un nouveau. C'est ce qui fait la différence avec la commande précédente : en sortant de l'éditeur, les clauses modifiées sont, cette fois-ci, détruites et remplacées par les nouvelles. Cette commande sera en particulier utile pour la mise au point des programmes, lorsqu'on s'aperçoit qu'une clause doit être énoncée de façon différente.

Attention : assurez-vous que la modification que vous allez apporter n'est pas trop importante, afin que l'éditeur puisse la saisir intégralement (sinon, il y a débordement mémoire et seul ce que vous venez de taper est conservé).

UNE FOIS DANS L'ÉDITEUR

Selon la fonction de l'éditeur que vous avez sélectionnée et appelée, vous vous retrouvez ou non devant un paquet de clauses. De plus, selon le cas, votre objectif est soit de saisir un texte, soit de modifier un texte existant, soit les deux.

Quelle que soit la fonction retenue, la manipulation de l'éditeur est la même :

— l'éditeur vous permet d'énoncer tout clause, même si elle a plus de 39 caractères (on rappelle que la caractéristique principale de l'éditeur est d'utiliser le mode "page" et non le mode "ligne"). C'est-à-dire que s'il arrive au 39e caractère (donc qu'il rencontre la fin de la ligne), il passera automatiquement à la ligne suivante, lorsque vous taperez le caractère suivant,
— l'éditeur travaille toujours en mode *insertion*, c'est-à-dire que tout caractère affichable est affiché à l'emplacement du curseur, en s'"insérant" entre le caractère précédent et les caractères compris entre la position du curseur et la fin de la ligne. Ces derniers sont donc décalés d'une position vers la droite.

Attention : pour que l'insertion soit possible, lors d'une modification (prolonger une clause ou insérer des caractères dans une clause), il faut que la ligne ne soit pas déjà pleine (c'est-à-dire qu'au moins le dernier caractère de la ligne soit le caractère "espace"). Sinon utilisez la touche fonction **INS** ;

— il se peut que l'éditeur déborde si les clauses que vous venez de taper occupent trop de place (mais celles-ci ne sont pas détruites).

Une fois qu'un texte a été saisi et qu'on veut le modifier par les commandes ED (<nom-du-paquet-de-clauses>) ou CH (<nom-du-paquet-de-clauses>), on s'aperçoit que l'éditeur a lui-même remis en page la formulation, en la découpant, afin de vous faciliter le travail de correction !

LES TOUCHES DE FONCTION UTILISABLES

Toutes les touches de fonctions utilisées en mode de saisie de commandes sont valables ici.

Certaines ont exactement la même fonction : ←, →, **EFF**, ↵ : leur action s'effectue toujours par rapport à une ligne (voir mode commande).

D'autres ont été modifiées :
— **la touche** **ENTRÉE** permet simplement de passer à la ligne suivante et non plus d'actionner l'analyseur syntaxique. Nous pouvons le comprendre simplement. L'analyse ne se fait plus ici par ligne, mais par texte saisi. Donc elle sera effectuée à la fin de la saisie, lorsque nous sortirons de l'éditeur,
— **la touche** **RAZ** : son action permet toujours d'effacer la ligne sur laquelle se trouve le curseur. Mais elle entraîne aussi la remise en page de votre texte. Toutes les lignes qui se trouvaient au-dessous sont remontées d'une rangée, afin qu'aucun "trou" n'apparaisse. Elle se comporte donc comme la fonction **EFF** utilisée pour un caractère. En fait, la ligne effacée n'a pas totalement disparu. Vous pourrez la "rappeler" en actionnant la touche **ACC** .

D'autres touches fonctions sont spécifiques à l'éditeur :

— **la touche** **INS** : sa fonction dépend de la position courante du curseur :

— le curseur est positionné en début de ligne. Parallèlement au mode insertion de caractères dans une ligne, la touche **INS** permet ici l'insertion d'une ligne dans une page. A partir de la position du curseur, une ligne blanche est insérée entre celle spécifiée par le curseur et celle qui la précède. La dernière ligne de la fenêtre disparaît de l'écran, mais elle n'est pas perdue pour autant. Elle peut être rappelée en faisant défiler le texte (voir ci-après),
— le curseur est positionné sur un caractère d'une ligne, la touche **INS** permet de "couper" la ligne, c'est-à-dire d'éditer tous les caractères à partir du curseur sur la ligne suivante (insérée automatiquement). Cette fonction est très pratique pour le travail de correction,

— **la touche** **ACC** : elle permet de rappeler la *dernière* ligne supprimée par action de la touche **RAZ**. La ligne rappelée s'affiche à la place de la ligne sur laquelle se trouve le curseur, en s'insérant entre la ligne précédente et la ligne courante antérieure. Selon le même principe que pour la fonction **INS**, toutes les lignes à partir de la ligne courante antérieure sont décalées d'une rangée vers le bas. Cette fonction est très pratique en particulier pour déplacer une ligne existante. Elle l'est également lorsqu'on a des clauses de même "format" à saisir. Il suffit d'en saisir une, de la détruire par **RAZ** et de la rappeler autant de fois que l'on veut par action de la touche **ACC**. Il suffira ensuite de rentrer ou modifier les termes changeants.

Mais rappelez-vous que son action ne se réfère qu'à la dernière ligne détruite. Elle n'a pas son pendant en mode insertion, si ce n'est la frappe manuelle du caractère détruit.

FAIRE DÉFILER UN TEXTE

De la même façon que l'on peut déplacer le curseur sur une ligne, l'éditeur permet de faire défiler un texte soit vers le haut soit vers le bas. Cette option est très intéressante, surtout lorsque notre texte fait plus de 20 lignes et qu'on ne peut donc le voir intégralement à l'écran.

Une première constatation s'impose. Ce n'est pas parce qu'une ligne du texte n'est pas visible qu'elle n'existe pas ou plus. C'est simplement à cause de la "hauteur" de l'éditeur. Pour vous en assurer, l'éditeur vous offre d'utiliser les touches ↓ (défilement du texte du haut vers le bas) et ↑ (défilement du texte du bas vers le haut) :

— **touche ↓** : lorsqu'on arrive en bas de la fenêtre initiale (soit par la touche flèche inférieure ↓, soit par le saut de ligne), la ligne courante devient donc la dernière visible. En réutilisant ce processus (si on a besoin de saisir sur la ligne suivante), la première ligne de la fenêtre est "chassée" de l'écran vers le haut (elle n'est cependant pas perdue), la suite du texte est décalée d'une ligne vers le haut pour aménager une ligne blanche en bas de l'écran. S'il existe des lignes qui avaient été "chassées" auparavant vers le bas, elles réapparaissent successivement par appui sur la touche ↓. C'est ce qui permet de faire défiler le texte du haut vers le bas. Son action répétée peut entraîner la saturation de l'éditeur (erreur débordement mémoire) ;

— **touche ↑** : son action est l'inverse de la précédente : si on est positionné sur la première ligne visible de la fenêtre, une action sur la touche ↑ permet de faire redescendre les lignes qui avaient été "chassées" auparavant vers le haut. Cette touche n'a plus aucune action, bien sûr, à partir du moment où la première ligne de la fenêtre correspond effectivement à la première ligne du texte. Ce processus permet de faire défiler un texte du bas vers le haut.

COMMENT SORTIR DE L'ÉDITEUR ?

Nous avons vu tout à l'heure qu'aucune analyse du texte n'était effectuée quand nous étions encore dans l'éditeur, qui, par définition, travaille sur un ensemble de lignes.

Mais alors, quand s'effectue l'analyse du texte ?

— **la touche STOP** : Une fois que nous avons demandé à sortir de l'éditeur par la touche STOP, l'analyse du texte source saisi est automatiquement déclenchée. Chaque clause est compilée et analysée. Si une erreur de syntaxe (ou autre) est détectée, un message s'affiche en dessous de la fenêtre de l'éditeur. Celui-ci est automatiquement rappelé, de telle sorte que la première ligne du texte qu'il affiche, corresponde à la première ligne où a été détectée une erreur. Cela veut dire que toutes les clauses précédentes étaient

correctes. De plus, le curseur est positionné un peu après l'erreur, de façon à mieux la localiser. Dans ce cas, corrigez l'erreur (en vous aidant des touches fonction décrites précédemment), puis relancez la sortie de l'éditeur. Si aucune erreur n'est détectée, l'interpréteur affiche le nombre de termes mémorisés et se replace en mode commande ;

— les touches **CNT** et **Z** : si vous désirez abandonner l'édition, c'est-à-dire ne pas prendre en considération le texte saisi ou ne pas modifier votre programme, il suffit d'appuyer simultanément sur les touches **CNT** et **Z**. Vous vous retrouvez alors dans la situation de départ, en mode commande.

PRÉCAUTION D'UTILISATION DE L'ÉDITEUR

L'éditeur vous permet de saisir des clauses de manière simple, mais il faut prendre quelques précautions pour éviter de perdre des clauses du programmeur-source.

1) Lorsqu'il il y a un certain nombre de lignes de programme déjà saisies (soit que vous les ayez tapées, soit que vous ayez rappelé la saisie d'un paquet de clause) ces lignes sont stockées en mémoire (dans les zones dites "dynamiques" qui ne sont utilisées que lors de la résolution), il peut donc y avoir, à un moment, un débordement de la mémoire. Dans ce cas l'éditeur affiche un message (au bas de l'écran) et vous demande de quitter l'éditeur. Ce qu'il est recommandé de faire, pour ne pas risquer de perdre ce que vous avez tapé.

2) Une fois que vous avez tapé une ou plusieurs clauses, vous allez sortir (par **STOP**) pour faire l'analyse syntaxique de cette clause. Il se peut qu'il n'y ait pas assez de place en mémoire pour stocker le programme mis au format interne après analyse. Dans ce cas un message d'erreur est affiché et vous vous retrouvez dans l'éditeur avec le curseur indiquant la première clause qui n'a pas été stockée (comme dans le cas d'une erreur de syntaxe). Les clauses qui ont été analysées et converties en format interne ont disparues de l'éditeur. Comme il n'y a plus de place mémoire, la seule alternative qui vous reste est de sortir de l'éditeur en appuyant sur **CNT** et **Z** ce qui vous fait perdre toute la partie saisie qui restait dans l'éditeur.

Pour éviter ce genre de mésaventure, il est recommandé de "gérer" la mémoire en :

— appelant la commande **STAT**, avant de rentrer dans l'éditeur, pour compresser la mémoire utilisée, et voir quel espace il vous reste (les termes surtout),

— ne pas éditer d'une traite de gros programmes, mais au contraire éditer par petit bout (paquet de clauses par paquet de clauses) en surveillant l'occupation de la mémoire grâce à la commande **STAT**.

1.3.3 - Lancer un programme

Lancer un programme en PROLOG, c'est demander une résolution, en énonçant un but à l'interpréteur, à partir des clauses que vous avez saisies. C'est un type de commande particulier, qui possède une syntaxe précise, ressemblant beaucoup à celle d'une clause : la tête de la clause est affectée du symbole "?", le corps de la clause énonce le ou les buts à atteindre.

En conclusion, la syntaxe d'une demande de résolution correspond à une synthèse des deux modes vus précédemment. On peut considérer que le symbole "?" est une commande, et que le reste correspond à une partie de clause.

Demander une résolution entraîne généralement l'affichage des solutions à l 'écran, par l'interpréteur. Vous pouvez arrêter momentanément ce défilement, par appui sur une touche quelconque du clavier. Pour reprendre l'affichage, il suffit d'appuyer une nouvelle fois sur une touche.

L'appui sur la touche **RAZ** provoque l'abandon de la résolution.

1.4 - LA GESTION DES FICHIERS DE PROGRAMMES

Dès que vous allez commencer à programmer en PROLOG, vous allez rapidement éprouver le besoin de garder quelque part une "trace" de vos programmes.

Les deux commandes ci-dessous permettent, d'une part de sauvegarder la totalité du programme qui se trouve en mémoire, sur un fichier cassette ou disquette, d'autre part de pouvoir relire ensuite ces fichiers et réutiliser vos programmes. Ces commandes sont donc liées à la possession d'un magnétophone à cassettes ou d'un lecteur de disquettes, correctement connectés.

1.4.1 - Principes

Pour pouvoir sauver un programme puis le récupérer ensuite, il faut bien évidemment lui donner un nom, pour le différencier des autres.

La syntaxe du nom de fichier est la même qu'en BASIC. Le nom doit comporter au plus huit caractères. Si on a besoin d'une extension pour le nom, celle-ci aura au plus trois caractères et sera séparée du nom de fichier par un point. L'extension permet, en général, d'identifier plus précisément le type du programme. Par défaut, l'extension est PRG.

Nous verrons que chaque fois que nous avons besoin de préciser le nom d'un fichier ou programme, celui-ci doit être mis entre apostrophes.

1.4.2 - Sauvegarder un fichier

La sauvegarde permet d'enregistrer sur un support le programme contenu en mémoire. Ce support peut être une cassette ou une disquette. Il est conseillé de sauvegarder un programme avant de l'exécuter. Le programme est sauvé sous forme de programme source décompilé, c'est-à-dire comme lorsqu'on demande un listing. De plus, la sauvegarde s'effectue paquet de clauses par paquet de clauses. Un paquet de clauses trop important pourra nécessiter des sauvegardes successives. Pendant la sauvegarde, un ou des caractères "=" s'affichent à l'écran (ils visualisent la progression du travail). Une fois la sauvegarde effectuée, le symbole "$" réapparaît à l'écran.

• SAUVEGARDE SUR CASSETTE

La syntaxe est SAVE ('C :<nom du fichier>') ou SAVE ('<nom du fichier>) le C est facultatif.

Avant de valider la commande par **ENTREE**, positionnez votre cassette à l'endroit où vous voulez effectuer l'enregistrement en notant le numéro au compteur.

Appuyez simultanément sur les touches "Play" et "Enreg.Rec" : un voyant rouge s'allume. Vous pouvez alors lancer votre commande. Une fois celle-ci effectuée, stoppez votre magnétophone.

Exemple :
SAVE ('C : FICH 1') : sauvegarde du fichier FICH 1.PRG sur cassette.

• SAUVEGARDE SUR DISQUETTE

La commande est SAVE ('<X> : <nom du fichier>').

<X> représente le numéro du lecteur de disques utilisés (de 0 à 4). Vous pouvez valider directement la commande, le lecteur de disquettes prenant soin de gérer lui-même l'emplacement de sauvegarde du fichier, en fonction de la place disponible sur la disquette.

Exemple :
SAVE ('0 : LETTRE.TXT') : sauvegarde du fichier LETTRE.TXT sur la disquette se trouvant sur le lecteur 0.

● Avant de sauvegarder des fichiers sur une disquette, il faut initialiser celle-ci par la commande FORMAT(X) où X est le numéro du lecteur utilisé (de 0 à 4).

● Il est conseillé de ne pas trop charger les disquettes (pour éviter tout problème).

● Pour connaître le catalogue des fichiers sauvegardés sur une disquette, on utilise la commande DIR(X) où X est le numéro du lecteur (par défaut, c'est le lecteur 0 qui est utilisé).

Le catalogue se présente sur deux colonnes, selon le format suivant : <nom de fichier> <extension> A A <taille en Ko>. A A indique que les fichiers sont de type texte.

● Pour détruire un fichier sur disquette, on utilise la commande KILL('<X> : <nom du fichier>') ou <X> est le numéro du lecteur. Cela entraîne l'affichage du catalogue de fichiers.

1.4.3 - Lire un fichier

Cette commande permet de lire un programme préalablement sauvegardé. Les clauses lues sont compilées et rajoutées aux clauses déjà présentes en mémoire.

Lorsqu'on veut relire un fichier, il faut que la mémoire allouée par SIZE soit au moins aussi grande que celle qui a été utilisée pour le sauvegarder.

Dès que le programme a trouvé le fichier recherché, son nom est affiché, sans apostrophes mais entre parenthèses, sur la ligne suivante.

Il essaye alors de le lire, tout en le compilant : dès qu'un paquet de clauses est complet, il est compilé et la place mémoire qu'il occupe est affichée (X TERMES).

● LECTURE D'UN FICHIER SUR CASSETTE

La syntaxe est LOAD
ou LOAD (<nom du fichier>).

Appuyez sur la touche "Play" de votre magnétophone et lancez la commande.

Si le nom du fichier n'est pas précisé, le programme lit le premier fichier source rencontré sur la cassette et essaye de le compiler.

Sinon, le programme lit séquentiellement la cassette jusqu'à ce qu'il rencontre le fichier demandé. On comprendra aisément que cette manipulation sera d'autant plus rapide qu'on aura pris soin de "mémoriser" au compteur du magnétophone, lors de la sauvegarde, la position relative du fichier sur la bande.

Exemple :
LOAD ('C : MEDICAL') : recherche et chargement en mémoire du fichier MEDICAL.PRG.

● LECTURE D'UN FICHIER SUR DISQUETTE

La syntaxe est LOAD ('<X> : <nom du fichier>').

<X> est un nombre représentant le numéro du lecteur de disques utilisé (de 0 à 4).

Le fonctionnement est le même que précédemment, sauf qu'ici, l'emplacement du fichier sur la disquette est retrouvé automatiquement et directement par le lecteur.

Exemple :
LOAD ('0 : FICHIER 2') : chargement en mémoire du fichier FICHIER2.PRG, qui se trouve sur le lecteur 0.

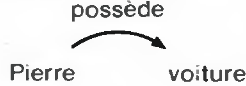
Précautions d'usages :
Lorsque vous créez un programme PROLOG celui-ci est stocké en mémoire (comme un programme BASIC) pour éviter de le perdre (suite à une coupure de courant ou à de l'électricité statique) faites des sauvegardes fréquentes.

2 - PREMIÈRE RENCONTRE AVEC PROLOG

2.1 - LES FAITS

PROLOG est un langage créé pour permettre de formuler et de résoudre des problèmes portant sur des objets et des relations entre ces objets, définissant des réalités, des vérités de notre univers.

C'est ce que nous faisons, quand nous affirmons, par exemple, que "Pierre possède cette voiture". Cela veut dire que nous créons une relation de possession entre l'objet "Pierre" et l'objet "voiture".



Cette opération de définition de relations entre objets s'appelle, en PROLOG, *la déclaration de faits*. On utilise les conventions suivantes :

- tous les noms sont écrits en majuscules, et n'ont pas plus de 8 caractères,
- on écrit le nom de la relation en premier, appelée *prédicat*,
- les objets concernés par la relation sont placés entre parenthèses derrière le prédicat, et séparés les uns des autres par des virgules. Les noms des objets mis entre parenthèses sont appelés *arguments* du prédicat,
- un fait doit toujours se terminer par un point ".".

Un ensemble de faits s'appelle en PROLOG une *base de données*. Elle constitue la base de connaissance élémentaires sur un domaine : tous les faits présents sont des réalités du domaine.

Remarques :

A. L'ordre dans lequel on énonce les arguments d'une relation détermine son sens. Ici, Pierre possède cette voiture, mais de là à dire que cette voiture possède Pierre !!!...

Si nous choisissons de représenter cette relation par :

POSSÈDE (PIERRE, VOITURE),

cela veut dire que nous avons choisi arbitrairement d'écrire "celui qui possède" en premier et "l'objet de possession" en second. Ce choix pour l'écriture de la relation POSSÈDE devra être impérativement suivi, si nous avons d'autres relations de possession à écrire. Ici POSSÈDE est un prédicat à deux arguments.

B. De la même façon que nous avons utilisé une convention arbitraire pour déterminer un ordre pour les objets, nous devons également affecter un sens précis à ces objets ou arguments.

Ici ce sens semble évident, mais si on considère que le nom des objets et des relations est lui aussi arbitraire, on s'aperçoit que la précision des significations est primordiale. On s'arrangera toujours, par esprit de clarté, pour donner des noms qui nous aident à nous rappeler leur signification première. Mais sachez que l'on aurait très bien pu représenter notre fait POSSÈDE (PIERRE, VOITURE), par A (B, C), en se souvenant que A signifie POSSÈDE, B signifie PIERRE et C signifie VOITURE.

C. Quand nous utilisons des noms, il faut donc décider comment les interpréter, en particulier lorsque leur sens n'est pas évident. Ainsi, dans la relation "l'argent est rare", dont le fait correspondant est RARE (ARGENT), nous devons impérativement préciser ce que nous entendons par ARGENT.

- Nous pouvons choisir de dire qu'il se rapporte à notre monnaie courante. Ainsi, en PROLOG, lorsque nous dirons RARE (ARGENT), nous voudrons signifier que l'argent est dur à gagner.
- Mais nous pouvons également choisir d'interpréter ARGENT comme le métal du même nom. Dans ce cas, cela signifiera, par exemple, que les mines d'argent sont peu nombreuses ou peu fécondes.

ATTENTION :
NOUS SOMMES TOTALEMENT RESPONSABLES DES FAITS QUE NOUS DÉFINISSONS. CELA VEUT DIRE QUE NOUS POUVONS DÉCLARER DES FAITS QUI NE SONT PAS VRAIS DANS LA RÉALITÉ. NOUS POURRIONS ÉCRIRE PAR EXEMPLE CAPITALE (GRENOBLE, FRANCE) POUR DÉCLARER QUE GRENOBLE EST LA CAPITALE DE LA FRANCE, CE QUI EST FAUX. PROLOG EST CAPABLE DE L'ANALYSER ET DE CONSTATER L'ERREUR. C'EST A VOUS D'EXPRIMER PERTINEMMENT LES RELATIONS ENTRE OBJETS ET DE LEUR DONNER UN SENS.

D. Les relations que nous utiliserons seront plus ou moins détaillées, selon ce que nous voulons leur faire dire. Ainsi, lorsque nous disons "cette maison est jolie", nous établissons une relation "être jolie" qui concerne une maison. Nous ne disons pas *qui* trouve cette maison jolie, *ni pourquoi* elle la trouve jolie.

E. On ne peut rien dire de l'absence (ou de la non déclaration) de certains faits dans une base de données. On ne sait rien sur eux, leur absence ne signifie pas qu'ils sont automatiquement faux, mais simplement que les relations correspondantes n'ont pas été énoncées (parce qu'on a pensé qu'elles n'étaient pas intéressantes ou pertinentes dans le domaine).

Exemples de faits :

HOMME (JEAN)	Jean est un homme
PERE (JEAN, MARTINE)	Jean est le père de Martine
PARENTS (MARC, PIERRE, ANNE)	Les parents de Marc sont Pierre et Anne
DONNE (OLIVIER, BONBON, SAM) + (2, 3,5)	Olivier donne un bon à Sam 2 / 3 = 5

● LES QUESTIONS SUR LES BASES DE DONNÉES

Une fois que nous possédons une base de données, nous pouvons demander si des relations entre certains objets sont vérifiées. En PROLOG, une question, ou but, ressemble à un fait précédé du symbole "?:" (le point final est facultatif).

Ainsi, si nous posons la question : "Est-ce que Pierre possède cette voiture?, qui s'énonce en PROLOG :

?: POSSÈDE (PIERRE, VOITURE)

c'est dans le but de savoir si la relation existe dans la base. Le système va parcourir la base de données que vous avez introduite auparavant (du début à la fin), à la recherche du ou des faits qui peuvent correspondre.

Deux faits *correspondent* si leurs prédicats s'épèlent de la même façon (donc s'ils sont identiques) et si tous leurs arguments respectifs et de même rang sont les mêmes.

Si PROLOG trouve un ou des faits qui correspondent à celui de la question, il va répondre -SUCCES- pour oui, suivi du nombre de solutions trouvées, sinon, cela veut dire qu'il a parcouru toute la base et qu'il n'a rien trouvé, et il répondra -ECHEC-. La réponse s'affiche à l'écran, sur la ligne juste en dessous de votre question.

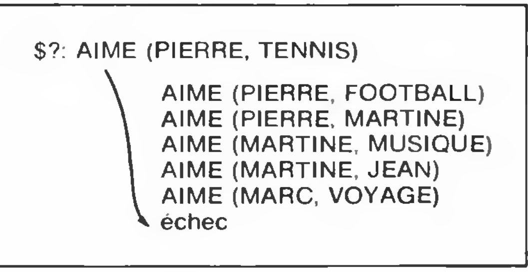
Soit la base de données suivante :

AIME (PIERRE, FOOTBALL)
AIME (PIERRE, MARTINE)
AIME (MARTINE, MUSIQUE)
AIME (MARTINE, JEAN)
AIME (MARC, VOYAGE).

Si nous déclarons ces faits au système, PROLOG pourra répondre aux questions que nous lui poserons :

$?: AIME (PIERRE, TENNIS)
--ECHEC--

En effet, PROLOG a parcouru toute la base de données, sans trouver de faits qui correspondent :



$?: AIME (MARTINE, JAZZ)
--ECHEC--

Même constatation :

$?: AIME (MARTINE,MUSIQUE)
--SUCCES--(1)--

PROLOG trouve un fait qui correspond dans la base (a). Puis il continue sa recherche jusqu'à la fin de la base (b), au cas où il trouverait un autre fait (cela voudrait dire que le fait exprimé dans la question est dupliqué dans la base) : en conclusion, il a trouvé une solution.

$?: POSSÈDE (PIERRE, MAISON)
--ECHEC--

La remarque à faire ici, est que la réponse "ECHEC" ne correspond pas toujours à un "non" catégorique, mais plutôt au "je ne sais pas" couramment utilisé dans la langue française. Dans notre base de données, il n'y avait aucune relation énoncée avec le prédicat "POSSÈDE", donc le système n'a pas pu *décider*, à partir des faits de la base.

Un dernier exemple, pour appuyer cette notion :

soit la base de données et les demandes de résolution suivantes :

HUMAIN (VERLAINE)
HUMAIN (ZOLA)
PARISIEN (ZOLA)

$?: PARISIEN (ZOLA)
--SUCCES--(1)--

$?: FRANÇAIS (ZOLA)
--ECHEC--

bien que géographiquement ce soit évident, le système ne peut pas décider.

Poser des questions uniquement de ce type, sur les faits que nous avons énoncé dans une base de données, n'est pas particulièrement intéressant, puisque cela revient à obtenir les informations que nous avons nous-mêmes introduites. Ce n'est heureusement pas l'unique but et l'unique ressource de la programmation en PROLOG, comme nous allons le voir maintenant.

2.2 - LES VARIABLES

Supposons que nous voulions savoir ce que Pierre aime. Nous pourrions, bien sûr, poser toute une série de questions du type : Est-ce que Pierre aime le football ?, Est-ce que Pierre aime la musique ?, Est-ce que Pierre aime Martine ?, etc. A chaque fois, le système nous répondrait par - SUCCÈS - ou - ÉCHEC - selon les résultats de son parcours dans la base de données. Mais quel travail fastidieux !

Au lieu de cela, nous préférerions pouvoir demander à PROLOG de nous dire lui-même tout ce que Pierre aime ; tous les objets vérifiant la relation dont AIME est le prédicat et PIERRE le premier argument. Dans la question, nous voudrions donc que le deuxième argument ne corresponde pas à un objet particulier, mais soit reconnu du système comme *n'importe quel objet que Pierre aime.* C'est le rôle des **variables,** objets bien spécifiques de PROLOG, qui permettent de représenter des objets que nous ne pouvons pas nommer. PROLOG distingue une variable d'un objet par le fait que *toute variable commence par le symbole* "*".

Ainsi, nous pourrions formuler notre question de la façon suivante :

$?: AIME (PIERRE, *X)

où *X représente "quelque chose que Pierre aime".

Quand PROLOG utilise une variable, celle-ci est soit :

-- **instanciée ;** c'est-à-dire qu'il existe, et qu'on a trouvé, un objet représenté par la variable. Par ce processus, on remplace donc une variable par un objet. Ainsi, on crée un exemplaire, un individu particulier d'un concept.

— libre ; c'est-à-dire non instanciée, ou en d'autres termes que ce qu'elle représente n'est pas encore connu.

Exemple :

Soit la base de données suivante :

```
POSSÈDE (PIERRE, VOITURE).
POSSÈDE (JEAN, VÈLO).
POSSÈDE (PIERRE, MAISON).
POSSÈDE (MARTINE, VOITURE).
POSSÈDE (MARTINE, BIJOU).
```

Si nous voulons savoir ce que Pierre possède, nous poserons la question :

```
$?: POSSÈDE (PIERRE. *X)
*X = VOITURE
*X = MAISON
--SUCCÈS--(2)--
```

Voyons plus précisément comment s'y prend PROLOG pour répondre à une question de ce type :

1 - La variable *X est d'abord libre. PROLOG parcourt la base de données à la recherche de faits qui correspondent à la question. C'est-à-dire à ceux dont le prédicat est POSSÈDE, le premier argument PIERRE et le deuxième argument un objet quelconque (puisqu'il correspond à la variable libre de la question).

2 - PROLOG effectuant sa recherche dans la base de données, dans l'ordre où celle-ci lui a été énoncée (c'est-à-dire de haut en bas), il va essayer de satisfaire la question ou but.

```
$?: POSSÈDE (PIERRE, *X)
   \
    POSSÈDE (PIERRE, VOITURE).
    POSSÈDE (JEAN, VÉLO).
    POSSÈDE (PIERRE, MAISON).          *X = VOITURE
    POSSÈDE (MARTINE, VOITURE).
    POSSÈDE (MARTINE, BIJOU).
```

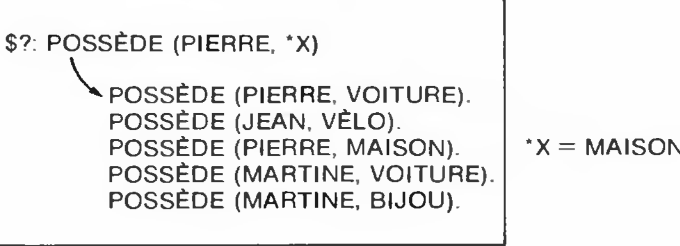
Le premier fait qu'il va trouver est POSSÈDE (PIERRE, VOITURE). PROLOG a trouvé ici un objet représenté par la variable VOITURE. On dit que la variable est instanciée par VOITURE. Nous venons donc de trouver un objet que Pierre possède, et PROLOG l'affiche à l'écran :

*X = VOITURE.

3 - Mais PROLOG ne va pas s'arrêter en si bon chemin. Il existe peut-être d'autres objets qui correspondant à la recherche. Ainsi PROLOG va-t-il continuer sa recherche, mais de façon intelligente. Au lieu de reparcourir intégralement la base de données, il va

reprendre sa recherche à l'endroit où il s'était arrêté. En effet, à chaque fois qu'il trouve une réponse pertinente, il prend soin d'en "marquer" l'endroit dans la base, signifiant ainsi la limite entre ce qui a déjà été étudié et ce qui n'a pas encore été parcouru.

A chaque fois qu'il reprend ainsi sa recherche, on dit qu'il essaye de *resatisfaire* la question, en repartant du dernier choix effectué. Pour cela, PROLOG doit "oublier" que *X représente VOITURE, afin d'essayer de lui donner une autre représentation. Il commence par *désinstancier* la variable *X, et reprend sa recherche avec une variable libre, à partir de la position du marqueur.



Le deuxième fait correspondant est POSSÈDE (PIERRE, MAISON). La variable *X est alors instanciée par MAISON, (elle est affichée à l'écran : *X = MAISON), et le marqueur est positionné sur ce fait.

4 - Selon le même processus, PROLOG essaye de resatisfaire la question. La recherche continue, mais à la fin du parcours, aucun autre fait n'a été trouvé. PROLOG arrête la recherche et affiche le nombre de résultats trouvés :

--SUCCÈS--(2)--

```
$?: POSSÈDE (PIERRE, *X)

        POSSÈDE (PIERRE, VOITURE).
        POSSÈDE (JEAN, VÈLO).
        POSSÈDE (PIERRE, MAISON).
        POSSÈDE (MARTINE, VOITURE).
        POSSÈDE (MARTINE, BIJOU).
        Fin de parcours de la base.
```

Si nous voulons maintenant savoir quels sont les individus qui possèdent une voiture, nous poserons la question suivante :

```
$?: POSSÈDE (*X, VOITURE)
*X = PIERRE
*X = MARTINE
--SUCCÈS--(2)--
```

Ainsi les variables nous permettent de travailler sur des objets ou ensembles d'objets inconnus au départ, et qui nous sont révélés par les réponses aux questions que nous pourrons poser.

2.3 - LES CONJONCTIONS

Supposons que nous voulions savoir, en compliquant un peu les choses, si plusieurs relations sont vérifiées simultanément.

Pour exemple, prenons la base de données suivante :

```
AMI (PIERRE, MARTINE).
AMI (MARTINE, ANIMAUX).
AMI (MARTINE, JACQUES).
AMI (MARTINE, PIERRE).
```

Nous voulons savoir si Pierre et Martine sont amis, en considérant que la relation d'amitié est quelque chose de réciproque.

La première idée qui vient à l'esprit est de demander si Pierre est un ami de Martine. PROLOG répondrait -SUCCÈS- ; puis de demander si Martine est amie de Pierre. PROLOG répondrait également -SUCCÈS-.

C'est-à-dire qu'on devrait poser deux questions, donc satisfaire deux buts, pour résoudre ce problème, et déduire des réponses si la relation initiale est vérifiée (elle ne le sera que si les deux citées précédemment le sont simultanément).

Comme ce type de combinaisons est fréquemment utilisé par les programmeurs en PROLOG, une notation particulière a été adoptée pour cela. Elle consiste à relier les différentes questions que nous avons eues à poser par le symbole "&" (pour ET), en créant ainsi une *conjonction de buts*. Une conjonction de buts est vraie si *tous* les buts qui la composent sont vrais *simultanément*. Ainsi la question précédente se formulerait :

```
$?: AMI (PIERRE, MARTINE) & AMI (MARTINE, PIERRE)
--SUCCÈS--(1)--
```

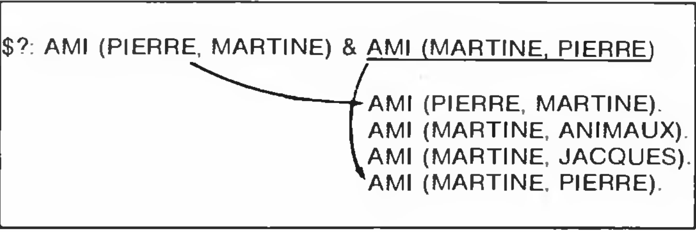
Comment procède PROLOG pour répondre à ce type de question ?

1 - PROLOG essaye d'abord de satisfaire le premier but de la conjonction (le plus à gauche), en cherchant si le fait correspondant existe dans la base (c'est le premier) :

```
$?: AMI (PIERRE, MARTINE) & AMI (MARTINE, PIERRE).

                                AMI (PIERRE, MARTINE)
                                AMI (MARTINE, ANIMAUX).
                                AMI (MARTINE, JACQUES).
                                AMI (MARTINE, PIERRE).
```

2 - PROLOG va essayer maintenant de satisfaire le but suivant, de la même façon ; il trouve également le fait dans la base (c'est le dernier) :



Tous les buts de la conjonction étant satisfaits, la conjonction est donc également satisfaite et vérifiée (le résultat est SUCCÈS). La recherche de PROLOG s'arrête une fois qu'il a parcouru toute la base de données pour chaque but (selon un processus que nous développerons dans l'exemple suivant), sans avoir trouvé d'autres solutions.

Prenons un autre exemple :

```
AIME (JEAN, ANNE).
AIME (ANNE, ANIMAUX).
AIME (JEAN, ANIMAUX).
AIME (ANNE, PIERRE).

$?: AIME (JEAN, ANNE) & AIME (ANNE, JEAN)
--ÉCHEC--
```

La réponse est "non" puisque s'il est un fait que Jean aime Anne, rien ne nous dit qu'Anne aime Jean. Or comme nous voulons ici savoir s'ils s'aiment mutuellement, la réponse est malheureusement négative.

Mais peut-être que Jean et Anne aiment quelque chose en commun ?

De l'utilisation simultanée des conjonctions et des variables, nous pouvons poser ce type de questions ; ce qui devient de plus en plus intéressant.

Cette question contient deux buts :

- trouver un objet *X qu'Anne aime.
- voir si cet objet *X est aimé de Jean.

Ce qui s'énonce en PROLOG :

```
$?: AIME (ANNE, *X) & AIME (JEAN, *X)
*X = ANIMAUX
--SUCCÈS--(1)--
```

N'oublions pas que la question est en quelque sorte hiérarchisée, par le fait que la recherche sur les buts va s'effectuer de gauche à droite, et que chaque but possède son *propre marqueur.*

Pour répondre à cette question, PROLOG procède de la façon suivante :

A - Il essaye d'abord de satisfaire le premier but :

- s'il trouve un fait qui corresponde, il en marque l'endroit dans la base de données, et instancie la variable par l'objet trouvé ;
- sinon, la recherche s'arrête et la réponse est --ÈCHEC-- : en effet, ce n'est pas la peine d'aller plus loin, si on n'a pas pu satisfaire le premier but.

B - S'il a pu satisfaire le premier but, il va essayer désormais de satisfaire le second, sachant que la variable est instanciée par un objet précis. Un principe est à retenir ici ; dès qu'une variable a été instanciée par un objet, toutes les variables de même nom apparaissant dans la demande de résolution le sont par le *même* objet. En repartant du début de la base de données, PROLOG va essayer de trouver un fait correspondant au deuxième but. S'il le trouve, c'est qu'il existe une solution satisfaisant les deux buts simultanément : la valeur de la variable est affichée, le marqueur du deuxième but est positionné à cet endroit dans la base de données.

C - Et la recherche continue, pour essayer de resatisfaire le deuxième but, à partir de son marqueur, en vue de trouver d'autres solutions. A un moment ou à un autre, toute la base de données a été parcourue pour le deuxième but. PROLOG va essayer alors de resatisfaire le premier but, en repartant du dernier choix effectué pour ce but. Pour ce faire, il désinstancie la variable, repart de la marque de son pointeur et procède comme précédemment en a, (sauf que le résultat ne pourra être l'échec général puisqu'à ce stade, PROLOG a déjà trouvé au moins une solution).

D - La recherche s'arrête quand toute la base de données a été parcourue pour le premier but (en effet, on ne peut plus le resatisfaire).

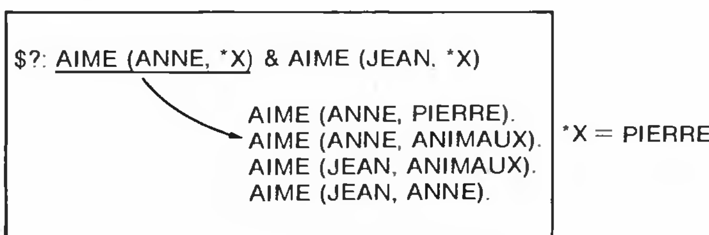
Visualisons cela sur l'exemple précédent, afin de clarifier les choses :

$?: AIME (ANNE, *X) & AIME (PIERRE, *X).

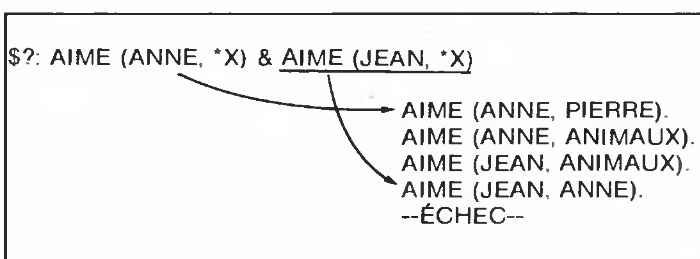
1 - On s'intéresse d'abord au premier but, contenant une variable qui, pour l'instant, est libre. Elle peut donc correspondre à n'importe quel objet, à partir du moment où celui-ci apparaît comme deuxième argument d'un fait de prédicat AIME et de premier argument ANNE.

PROLOG va parcourir la base de données, à la recherche du premier fait qui lui corresponde : celui-ci est AIME (ANNE, PIERRE).

Maintenant, la variable *X est instanciée par PIERRE. Et cela va plus loin ; cette instanciation est valable sur toute la question, c'est-à-dire que le deuxième but est maintenant AIME (JEAN, PIERRE). Avant d'essayer de satisfaire ce deuxième but, PROLOG va marquer dans la base l'endroit où il a trouvé le fait correspondant au premier but.



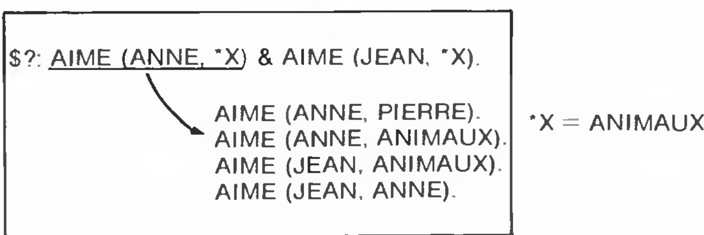
2 - Maintenant que le premier but a été satisfait, on essaye de satisfaire le second, AIME (JEAN, PIERRE). Le système parcourt la base de données depuis le début, à la recherche du fait correspondant : on s'aperçoit que le deuxième but échoue.



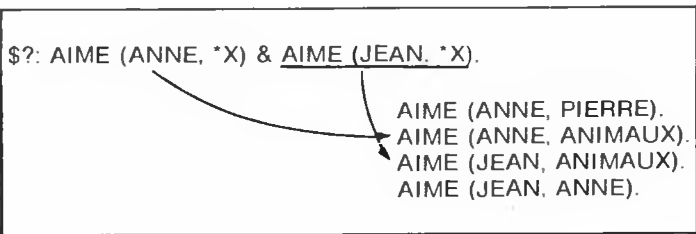
3 - Comme il n'y a plus rien à tirer du deuxième but, le système va l'oublier momentanément pour "revenir" s'intéresser au premier (donc au dernier choix effectué). C'est ce que l'on appelle **le retour arrière** (en anglais : **backtracking).** Ceci permet de rechercher d'au-

tres solutions, puisque dans un premier temps PROLOG a considéré le premier fait correspondant rencontré dans la base (il y en a peut-être d'autres).

PROLOG sait où il en était resté par son marqueur : on se retrouve en fait dans la situation du schéma 1). Avant de reprendre sa recherche, il va désinstancier la variable et donc "oublier" l'objet PIERRE, afin de chercher si un autre objet peut lui correspondre. Le prochain fait correspondant est AIME (ANNE, ANIMAUX). La variable *X est alors instanciée par ANIMAUX, le pointeur marque l'endroit dans la base de données où a été trouvé ce fait :



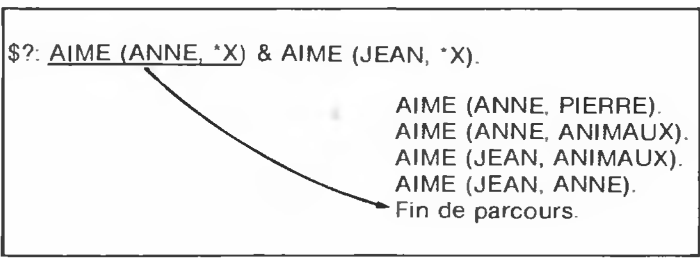
4 - Maintenant, on essaye de satisfaire le deuxième but, en sachant que *X est instancié par ANIMAUX. Ce but revient à chercher si un fait correspond à AIME (JEAN, ANIMAUX) dans la base de données : le but réussit. On a donc trouvé un objet qui vérifie à la fois le premier et le deuxième but, donc c'est une première solution, qui vous est annoncée à l'écran : *X = ANIMAUX.



5 - Le système va alors essayer de resatisfaire le deuxième but, c'est-à-dire trouver un autre fait qui corresponde. Ici, cela n'a pas grand sens, et il y a peu de chance pour qu'il trouve un autre fait (sauf si on a dupliqué le fait AIME (JEAN, ANIMAUX)). Mais sachez que dans des cas où les buts sont énoncés de façon plus complexe (avec plus de variables), cette démarche s'avère primordiale. Donc ici la tentative de resatisfaire le but échoue.

6 - Le système va essayer, une fois de plus, de resatisfaire le premier but (il va essayer de trouver une autre chose qu'Anne aime). Nous nous retrouvons au niveau de schéma 3). La recherche va repartir

du pointeur laissé pour ce but, après que la variable *X ait été désinstanciée. Aucun fait ne correspond plus au but. Ce processus de recherche est donc stoppé, puisqu'il n'y a plus de but à satisfaire. En conclusion, nous avons trouvé une solution pour la question initiale :



Nous voyons que, de l'ordre dans lequel nous énonçons les buts, dépend la façon dont travaille le système.

A titre d'exemple, essayez de voir comment les choses se passeraient si nous énoncions la question dans l'autre sens :

$?: AIME (JEAN, *X) & AIME (ANNE, *X).

● **CONCLUSION :**

Récapitulons ce qui se passe lorsqu'on demande une résolution portant sur une conjonction de buts :

- chaque but a un voisin de gauche (sauf le premier) et un voisin de droite (sauf le dernier) ;
- chaque but possède son propre marqueur ;
- PROLOG essaye de satisfaire chaque but l'un après l'autre, en procédant de la gauche vers la droite ;
- si le premier but échoue sans avoir jamais été satisfait, alors toute la conjonction échoue ;
- si un but est satisfait, PROLOG associe un marqueur à ce but dans la base de données (pour optimiser les recherches ultérieures). Si le but contenait une variable (libre), celle-ci est instanciée et toutes les occurences de la variable dans la conjonction sont instanciées par le même objet. Puis PROLOG tente de satisfaire le voisin de droite du but, en repartant du début de la base de données ;
- chaque fois qu'un but échoue, PROLOG essaye de resatisfaire son voisin de gauche, en partant de la position de son marqueur. Auparavant, il aura désinstancié (libéré) les variables qui avaient été instanciées précédemment, dans le but de pouvoir trouver un nouvel objet, en "oubliant" le précédent ;

- la manière dont PROLOG tente à plusieurs reprises de satisfaire et resatisfaire les buts d'une conjonction s'appelle le *retour arrière*.

2.4 - LES RÈGLES

Nous avons, jusqu'à maintenant considéré des relations très simples à énoncer. Or ce n'est pas toujours le cas. En effet, le plus généralement, un fait ne peut s'énoncer de façon élémentaire, car il dépend lui-même d'un ensemble d'autres faits. C'est ce que l'on appelle une **règle.** Une règle (exprimée en français par le mot "si") permet d'énoncer des définitions sur notre univers, ou en d'autres termes, des déclarations d'ordre général sur des objets et leurs relations.

● Des phrases telles que "Je vais à la piscine s'il fait beau" ou "Pierre a de bonnes notes s'il travaille" ne peuvent s'énoncer sous forme de faits élémentaires. Le fait d'aller à la piscine dépend du temps ; et le fait que Pierre ait des bonnes notes dépend du travail qu'il fournit. L'action à effectuer dépend donc de la situation. La relation qui unit ces faits introduit la notion de *règle*.

● En PROLOG, une règle se compose d'une tête (prédicat correspondant à l'action à effectuer) et d'un corps (relation ou conjonction de relations correspondant à la situation requise), reliés l'un à l'autre par le symbole ":" signifiant "si". Elle se termine par un point.

● La tête d'une règle sera satisfaite *si et seulement si* le corps de la règle l'est également. C'est-à-dire que, lors d'une demande de résolution portant sur une règle, le système tentera de satisfaire le but, ou la conjonction de buts, composant le corps de la règle, selon les principes vus au § 2.3, p. 71.

Exemples :

A - "Cette voiture est à Pierre si elle est de couleur rouge" s'énonce par la règle :

> POSSÈDE (PIERRE, VOITURE) : COULEUR (VOITURE, ROUGE).

B - "Pierre aime toutes les personnes qui aiment les animaux". Nous allons énoncer une règle :

> Pierre aime un "objet" *si* celui-ci est une personne *et* si cette personne aime les animaux.

Pour définir cette règle, nous allons utiliser une variable *PERSONNE et une conjonction de buts. En effet, nous ne voulons pas nommer cette personne, puisque nous voulons pouvoir obtenir *toutes* les personnes qui correspondent. De plus, nous voyons que nous avons besoin de deux relations pour définir le corps de la règle (la situation requise).

LES PROPRIÉTÉS D'UNE VARIABLE DANS UNE RÈGLE SONT LES SUIVANTES :

- toute variable de même nom utilisée dans une règle représente le même objet. C'est-à-dire qu'à partir du moment où elle a été instanciée, elle l'est sur toute la règle : c'est le principe de *cohérence* ;
- on peut autoriser une variable à représenter un objet différent à chaque utilisation *différente* de la règle.

La phrase précédente s'écrirait :

AIME (PIERRE, *PERSONNE) : HUMAIN (*PERSONNE) & AIME (*PERSONNE, ANIMAUX).

La tête de la règle correspond à : AIME (PIERRE, *PERSONNE).

Le corps de la règle correspond à : HUMAIN (*PERSONNE) & AIME (*PERSONNE, ANIMAUX).

La variable *PERSONNE est utilisée trois fois. Chaque fois que *PERSONNE est instanciée par un objet, c'est toutes les occurences *PERSONNE de la règle qui sont instanciées par le même objet, *dans la limite de la portée de la variable* (définie ici par la règle entière).

Par exemple, si *PERSONNE est instanciée par ANNE, dans la tête de la règle, les buts du corps de la règle seront HUMAIN (ANNE) et AIME (ANNE, ANIMAUX).

C - Compliquons un peu le problème. Supposons que nous ayons à exprimer la relation "être frère de". Par définition, une personne est le frère d'une autre *si* elle est de sexe masculin et *si* ces deux personnes ont les mêmes parents. Ainsi, nous allons exprimer la relation "être frère" par une combinaison d'autres relations qui la définissent, en énonçant une règle.

Ici, on pourrait dire :

X est le frère de Y *si* : X est un homme
X et Y ont les mêmes parents.

sachant que toute variable de même nom utilisée dans la règle représente toujours le même objet. Ici, cela n'aurait donc aucun sens de dire : Marc est le frère de Julie si Médor est un homme et si table et animaux ont les mêmes parents !

En PROLOG, on écrirait la règle "être frère de" par :

FRÈRE (*X, *Y) : HOMME (*X) & PARENTS (*X, *PÈRE, *MÈRE) & PARENTS (*Y, *PÈRE, *MÈRE).

Ce qui veut dire : *X est frère de *Y **si**
*X est un homme
*X a un père *PÈRE et une mère *MÈRE
*Y a le même père et la même mère que *X.

Le prédicat PARENTS a trois arguments *X, *Y, *Z tel que PARENTS (*X, *Y, *Z) signifie que les parents de *X sont *Y et *Z, avec *Y père de *X et *Z mère de *X.

Remarquez l'utilisation des conjonctions et des variables. Ici, on a utilisé quatre variables :

- d'une *X pour représenter l'objet inconnu "frère" ;
- une autre *Y pour représenter l'objet inconnu dont *X est le frère ;
- les variables *PÈRE et *MÈRE, non présents dans la tête de la règle, représentent les parents de *X (donc dépendent de lui) et également ceux de *Y (du moins, on l'espère). Elles sont traitées de la même manière que les autres variables, c'est-à-dire, en particulier, qu'au début elles sont libres.

En utilisant une base de données adéquate, essayons de traiter l'exemple de règle précédent :

HOMME (ZEUS).
HOMME (CASTOR).
HOMME (POLLUX).
FEMME (LEDA).
PARENTS (POLLUX, ZEUS, LEDA).
PARENTS (CASTOR, ZEUS, LEDA).
FRÈRE (*X, *Y) : HOMME (*X) & PARENTS (*X, *PÈRE, *MÈRE)
& PARENTS (*Y, *PÈRE, *MÈRE).

Nous remarquons que cette base est une partie de la généalogie et de la mythologie grecque. Que les puristes ne soient pas choqués, les prédicats HOMME et FEMME n'enlèvent rien au caractère divin que peuvent avoir ces personnages. Mais nous n'en parlerons pas ici, puisque ce n'est pas notre problème.

Exemple 1 :

Supposons que nous voulions savoir si Castor est le frère de Pollux, ce que nous énonçons en PROLOG par :

$?: FRÈRE (CASTOR, POLLUX).

Sachant que la base de données et la règle FRÈRE ont été déclarées au système, PROLOG va procéder de la manière suivante :

1 - Puisque la question posée correspond à la tête de l'unique règle FRÈRE énoncée (que PROLOG rencontre en parcourant la base de données), la variable *X va être instanciée par CASTOR et la variable *Y par POLLUX.

Désormais, toute variable *X rencontrée dans la règle représente CASTOR, de même pour *Y qui représente POLLUX. Le marqueur est alors positionné sur cette règle. PROLOG va alors tenter de satisfaire la règle en satisfaisant chacun des buts du corps de la règle l'un après l'autre (selon les principes déjà énoncés au § 2.3, p. 71), sachant maintenant qu'ils s'énoncent de la façon suivante :

HOMME (CASTOR).
PARENTS (CASTOR, *PÈRE, *MÈRE).
PARENTS (POLLUX, *PÈRE, *MÈRE).

2 - Le premier but à satisfaire est HOMME (CASTOR). Le système le rencontre en parcourant la base de données (c'est le deuxième), et

donc ce but est satisfait. PROLOG en marque la place dans la base de données et passe au but suivant.

3 - Le deuxième but est PARENTS (CASTOR, *PÈRE, *MÈRE). Les variables *PÈRE et *MÈRE sont libres. On va donc essayer de trouver un fait de la base qui corresponde (c'est-à-dire dont le prédicat est PARENTS et le premier argument CASTOR). Le premier fait de ce type rencontré est PARENTS (CASTOR, ZEUS, LEDA). PROLOG marque la place de ce but dans la base de données. De plus, il instancie les variables *PÈRE par ZEUS et *MÈRE par LEDA ; puis essaye de satisfaire le but suivant.

4 - Le troisième but s'énonce désormais PARENTS (POLLUX, ZEUS, LEDA). Le système va essayer de voir si le fait correspondant existe dans la base de données. Il le trouve en cinquième position. Ce but étant le dernier de la conjonction, comme il a réussi, le but entier réussit et le système a vérifié le fait FRÈRE (CASTOR, POLLUX).

5 - PROLOG essaye de resatisfaire chacun des buts et ne trouve pas d'autre solution : la recherche s'arrête. Le système annonce à l'écran que le but initial a été vérifié par --SUCCÈS--(1)--.

Exemple 2 :

Maintenant si nous voulons savoir de qui Castor est le frère, ou si Castor est le frère de quelqu'un (variable *PERS), nous demanderons la résolution suivante en PROLOG :

$?: FRÈRE (CASTOR, *PERS).

— Comme tout à l'heure, la question correspond à la tête de l'unique règle FRÈRE (*X, *Y).

```
FRÈRE (*X, *Y) : HOMME (*X) &
                 PARENTS (*X, *PÈRE, *MÈRE).
                 PARENTS (*Y, *PÈRE, *MÈRE).
```

par définition.

```
FRÈRE (CASTOR, *PERS) : HOMME (CASTOR) &
                        PARENTS (CASTOR, *PÈRE,
                        *MÈRE) &
                        PARENTS (*PERS, *PÈRE, *MÈRE)
```

en calquant la question posée sur la règle de définition, la variable *PERS de la question est libre, tout comme celle nommée *Y dans la règle. Il n'est donc pas question ici d'instanciation. On dit que les variables *PERS de la question et *Y de la règle sont **unifiées** ; dès qu'une de ces variables sera instanciée par un objet, l'autre le sera également par le *même* objet. Si à la fin de la résolution une variable de la résolvante est "liée" à une autre variable libre, l'interprèteur affiche comme contenu une étoile (*) suivie d'un nombre qui peut être très grand.

— Le premier but à satisfaire est HOMME (CASTOR). Pas de problème, il va réussir comme précédemment, puisque ce fait est dans la base.

— Le deuxième but est PARENTS (CASTOR, *PÈRE, *MÈRE) et va correspondre à PARENTS (CASTOR, ZEUS, LEDA). A cette phrase, les variables *PÈRE et *MÈRE sont instanciées respectivement par les ZEUS et LEDA.

— Le troisième but consiste à trouver un objet qui corresponde à la variable *Y dans le but PARENTS (*Y, ZEUS, LEDA). Celui-ci correspond au fait PARENTS (POLLUX, ZEUS, LEDA). Donc *Y est instanciée par POLLUX. Par le procédé d'unification, la variable *PERS de la question est automatiquement instanciée par POLLUX.

— Tous les buts ayant été satisfaits, la règle toute entière est réussie. *X est représenté par CASTOR et *PERS par POLLUX. PROLOG a donc trouvé une solution : *PERS = POLLUX.

— PROLOG va essayer de resatisfaire ce but, en vue de trouver d'autres solutions. En effet peut-être que CASTOR est frère de quelqu'un d'autre ? En repartant du marqueur du troisième but dans la base de données, il va chercher si un autre fait peut correspondre, après avoir "oublié" la valeur précédente de *Y. Ici, il trouve PARENTS (CASTOR, ZEUS, LEDA). Les variables *Y de la règle puis *PERS de la question (par unification) sont instanciées par CASTOR. PROLOG en déduit une nouvelle solution : *X = CASTOR.

Cela peut paraître étrange (CASTOR est le frère de CASTOR !!!), mais cette solution et totalement en accord avec la règle que nous avons énoncée. Cela veut dire que nous ne l'avons peut-être pas énoncée assez précisément. Si nous voulons éviter ce genre de résultats, nous devrons rajouter et spécifier dans la règle, un prédicat précisant que les variables *X et *Y doivent être différentes.

— Ce troisième but ne pouvant plus être resatisfait, le système va effectuer un retour arrière, pour essayer de resatisfaire le deuxième but (sans effet ici), puis le premier (ce qui est également impossible). Donc la recherche s'arrête après nous avoir donné deux solutions :

```
*PERS = POLLUX
*PERS = CASTOR
--SUCCÈS--(2)--
```

En guise de conclusion, nous vous proposons le petit exercice classique suivant. Supposons que les relations suivantes soient déjà écrites :

PÈRE (*X, *Y)	(*X est le père de *Y)
MÈRE (*X, *Y)	(*X est la mère de *Y)
HOMME (*X)	(*X est un homme)
FEMME (*X)	(*X est une femme)
PARENT (*X, *Y)	(*X est un parent de *Y)
DIF (*X, *Y)	(*X est différent de *Y)

On vous demande d'écrire les relations suivantes, sous forme de règles PROLOG, à partir de ce dont vous disposez :

ÊTRE-MÈRE (*x)	(*X est une mère)
ÊTRE-PÈRE (*X)	(*X est un père)
ÊTRE-FILS (*X)	(*X est un fils)
SŒUR (*X, *Y)	(*X est la sœur de *Y)
GRAND-PÈRE (*X, *Y)	(*X est le grand-père de *Y)
FRÈRE-OU-SŒUR (*X, *Y)	(*X est frère ou sœur de *Y).

Exemple :

Si nous voulons écrire la règle ONCLE, sachant que les règles HOMME, FRÈRE-OU-SŒUR et PARENT sont déjà écrites :

```
ONCLE (*X, *Y) : HOMME (*X) &
                 FRÈRE-OU-SŒUR (*X, *Z) &
                 PARENT (*Z, *Y).
```

Comment pourrions-nous énoncer cette même règle, si nous disposions, en plus, de la règle FRÈRE ?

• CONCLUSION

Nous avons maintenant une vision d'ensemble sur la plupart des principes essentiels de PROLOG tels que :

- les faits, les questions posées par rapport à ces faits ;
- la nécessité d'introduire des variables et des conjonctions ;
- la notion de règles et de retour-arrière.

Nous avons vu, de plus, qu'il y avait plusieurs façons d'énoncer un prédicat, comme AIME par exemple :

— par des faits : AIME (ANNE, ANIMAUX).
AIME (JEAN, ANNE).

— par des règles : AIME (JEAN, *X) : AIME (*X, ANIMAUX).

En général, un prédicat sera défini par un mélange de faits et de règles. Les faits et les règles qui définissent un prédicat s'appellent les **clauses** du prédicat. Une clause de prédicat est donc soit un fait, soit une règle.

Bien sûr, nous ne sommes pas entrés dans les détails, et un certain nombre de points restent encore à préciser. C'est ce que nous allons faire dans le chapitre suivant.

3 - COMMENT PROGRAMMER EN PROLOG ?

3.1 - LES PRINCIPES DE BASE : LES ÉLÉMENTS DE LA SYNTAXE ET LEUR REPRÉSENTATION EN MÉMOIRE

Il est temps maintenant d'entrer un peu dans le détail. Nous n'avons pas encore, bien que nous les ayons utilisés souvent, défini explicitement comment représenter les principes que nous avons rencontrés et introduits. Comme pour tout langage, on utilise une syntaxe précise pour permettre à l'ordinateur d'interpréter correctement, et de la même façon que nous, ce que nous lui proposons lorsque nous tapons des caractères au clavier. La syntaxe PROLOG va permettre de structurer les mots du langage et l'ordre dans lequel on les énonce.

La syntaxe PROLOG est relativement simple, afin de vous permettre d'énoncer vos connaissances, simplement, au clavier. De son côté, le système les interprètera (d'où son nom d'interpréteur PROLOG) c'es-à-dire qu'il les analysera, et les rangera, s'il n'a constaté aucune erreur, dans sa mémoire (son cerveau), dans des "cases spécifiques en fonction de leur interprétation. Ces "cases" sont plus ou moins indépendantes, on le verra, de façon à pouvoir mémoriser des *relations* entre objets. Nous avons appelé ces "cases", de façon assez impropre, des "termes" mémoire. Par définition, un terme est l'unité minimum de représentation d'un objet PROLOG en mémoire (4 octets).

Une première remarque tout d'abord, quant à l'utilisation générale du clavier : certains types d'éléments de la syntaxe (comme les commandes : voir § 1.3.1, p. 46) doivent être *impérativement* écrits en MAJUSCULES. Sachant, de plus, que les accents ne sont pas gérés par l'interpréteur (et donc, que l'écriture en minuscules perd de sa valeur), il est recommandé d'utiliser en permanence le **mode MAJUSCULE** (obtenu en appuyant simultanément sur la touche MAJUSCULE et la barre d'espacement du clavier).

3.1.1 - Les termes

Un terme est l'élément de base du langage (ne pas confondre avec les "termes mémoire"). Il permet de représenter n'importe quel objet ou relation utilisés. Un terme est constitué d'une suite de caractères et/ou de symboles. Par analogie, il correspond aux mots ou groupes de mots de notre langue française. Selon la façon dont nous agencerons des caractères, nous créerons des termes de type différents.

C'est ce que nous allons voir maintenant. Il existe quatre types de termes différents : les constantes, les variables, les termes composés (ou structures) et les textes.

3.1.1.1. - Les constantes

Les constantes servent à nommer des objets ou des relations spécifiques, c'est-à-dire connus. Elles sont une signification unique et constante. On distingue deux types de constantes : les atomes et les nombres entiers.

LES ATOMES

Un atome est composé d'une séquence de caractères "nommant" un objet. Cet objet peut-être un objet élémentaire, un nom de prédicat ou de fonction (voir plus loin).

— Un atome ne doit pas avoir plus de huit caractères (sinon seuls les huit premiers sont pris en compte par l'interpréteur).

— Il doit commencer par une lettre et ne pas contenir de caractères "espace" ni de symboles séparateurs tels que la virgule, les parenthèses (ouvert et fermé), le point, le deux-points, le "&", le symbole "*". Ces symboles séparateurs sont en fait des atomes spéciaux et ont une signification précise pour l'interpréteur.

— Par contre, un atome peut contenir des chiffres.

Exemples :

TOTO
POSSÈDE
AIME
GRD-PÈRE
PIERRE
FAIT12
A
B

Par contre, les exemples suivants ne sont pas des atomes :

12EME
GRD-PÈRE
PÈRE, MÈRE
'BONJOUR'
A*B
1

Un atome occupe un terme de la mémoire. Tous les atomes sont regroupés dans une partie de la mémoire appelée "Dictionnaire" (DIC).

LES NOMBRES ENTIERS

Ils sont utilisés pour effectuer des opérations arithmétiques. PROLOG permet d'utiliser les nombres entiers compris entre — 32767 et + 32767.

Par définition, un nombre entier doit commencer soit par un chiffre, soit par le signe "-" et doit comporter au plus cinq chiffres, sans aucun blanc entre les chiffres. La valeur absolue maximale d'un entier est limitée à 32767. Un nombre entier occupe un terme mémoire.

3.1.1.2 - Les variables

Une variable permet de représenter un objet inconnu (que l'on ne peut pas nommer) ou un ensemble d'objets de même nature.

Sa syntaxe est : *XXXXXXXX où XXXXXXXX est un nom quelconque de un à huit caractères alphanumériques.

Tout nom commençant par le symbole "*" sera considéré comme une variable. C'est ce symbole particulier que le différencie de l'atome. Par définition, on peut donc dire qu'une variable est un atome précédé du symbole "*".

Exemples :

*PÈRE
*FILS
*COMPTEUR
*PERSONNE
*X
*Y

Une variable occupe un terme mémoire.

Remarques :

Lorsque vous saisissez une variable, vous pouvez lui donner soit un nom significatif ou "mnémonique" (ex. : *MÈRE), soit un nom plus abstrait (en utilisant les lettres de la fin de l'alphabet : *X, *Y...).

Sachez que, de toute façon, le résultat est le même pour l'interpréteur, puisqu'il va renommer les variables, en leur affectant un simple numéro (*X0, *X1, *X2, ...). En effet, ce qui l'intéresse n'est pas de conserver exactement ce que vous avez tapé, mais plutôt la cohérence de ce que vous avez tapé. Le nom mnémonique n'a donc d'intérêt qu'au niveau de la clarification du programme source (état du programme lorsque vous le saisissez et qu'il n'a pas encore été analysé). Une fois que l'interpréteur l'a analysé, les noms mnémoniques des variables ont disparu et ont été remplacés par ceux, plus manipulables, donnés par l'interpréteur. De la même façon, les noms mnémoniques des variables ne sont pas conservés, lorsqu'on enregistre ou sauvegarde un programme sur cassette ou sur disquette.

Exemple :

Si vous saisissez la clause :

GND-PÈRE (*GD-PÈRE, *PTT-FILS) : PÈRE (*GD-PÈRE, *PÈRE) & PÈRE (*PÈRE, *PTT-FILS).

définissant la règle "être grand-père de", et que vous demandez directement après à lister cette clause à l'écran (par la commande LIST (GND-PÈRE)), l'interpréteur vous affichera :

= 1 = < 9 > GND-PÈRE (*X0, *X1) :
PÈRE (*X0, *X2) &
PÈRE (*X2, *X1).

Il a donc "oublié" les noms que vous aviez donnés aux variables, en les remplaçant par des variables *X numérotées. Le principal est qu'il ait conservé la cohérence entre les variables énoncées. Habituez-vous donc, dès maintenant, à travailler avec des noms de variables plus ou moins abstraits.

- Lorsqu'on utilise des variables dans une résolvante, seuls les noms des cinq premières variables sont conservés par l'interpréteur (les autres sont renommées).

- Par définition, une variable n'a pas d'activité dans un fait élémentaire. Par contre, elle pourra apparaître dans une règle, comme précédemment, ou dans des questions, en vue de trouver les objets qu'elle représente.

- Une variable peut avoir deux états distincts ; elle est soit :

— **instanciée :** c'est-à-dire qu'on a trouvé un terme représenté par la variable. La variable est remplacée par ce terme désormais connu. Un cas particulier d'instanciation est l'*unification*, lorsque le terme est lui-même une variable : il n'est pas question de remplacer une variable par une autre, ce qui n'aurait pas beaucoup de sens ni d'effet, mais de dire que dès qu'une de ces deux variables sera instanciée par un objet réel, l'autre le sera automatiquement par le *même* objet.

— **libre ou non instanciée :** c'est-à-dire que ce que représente la variable n'est pas encore connu. Lors d'une résolution, si l'interpréteur trouve une variable libre qui soit solution, il l'affiche en la représentant par un nombre précédé du symbole "*" spécifique aux variables. Le nombre peut être très grand (ex. : 33765) nous ne détaillerons pas sa signification ici.

Les termes que nous venons de définir sont des plus élémentaires. Voyons comment nous pouvons les assembler pour obtenir des formulations plus complexes et plus intéressantes.

3.1.1.3 - Les termes composés ou structures

Un terme composé est formé d'un nom de *fonction* ou *prédicat* suivi d'une liste, entre parenthèses, de termes séparés par des virgules,

appelés *arguments* de la fonction. Il permet d'énoncer des relations entre objets.

La syntaxe d'une structure est :

< nom-de-fonction > (<t1>, <t2>, ..., <tn>).

La présence de caractères "espace" est autorisée entre les termes, et le nombre d'espaces tapés est quelconque (ils ne seront comptés que pour un espace). Ceci permet en particulier d'"aérer" le programme.

Exemples :

PÈRE(JEAN,MICHEL) est un terme composé, de nom de fonction PÈRE, et d'arguments JEAN et MICHEL (qui sont tous les trois des atomes). Il exprime le fait que JEAN est le père de MICHEL.

PÈRE (JEAN , MICHEL) est équivalent à la structure précédente (on y a seulement introduit des espaces, pour "aérer" la structure).

LIVRE (ZOLA, NANA, 1897) : le prédicat est l'atome LIVRE; les arguments sont au nombre de trois, les atomes ZOLA et NANA et le nombre entier 1879. Par cette structure, nous voulons signifier que ZOLA a écrit le livre NANA en 1879.

La caractéristique des termes composés est que leurs arguments sont des termes (donc soit des constantes, soit des variables, soit eux-mêmes des termes composés, soit du texte). Ainsi, on peut imbriquer plusieurs structures l'une dans l'autre.

Exemples :

POSSÈDE (MICHEL, NANA, AUTEUR (ZOLA)), où AUTEUR (ZOLA) est lui-même un terme composé.

FAMILLE (PÈRE (ALAIN, ROBERT), MÈRE (NICOLE, ROBERT)).

Les structures peuvent également utiliser des variables, dans le processus de questions-réponses, ou dans la définition des règles.

Ainsi, si nous voulons savoir quels livres de Zola possède Michel, nous poserons en PROLOG la question suivante :

$?: POSSÈDE (MICHEL, *X, AUTEUR (ZOLA)).

LA NOTION D'ARBRE

La notion de structure est souvent difficile à cerner, surtout lorsqu'elle est appliquée à des structures de forme plus ou moins complexe. Pour en faciliter la compréhension, on a l'habitude de la visualiser sous forme d'**arbre.**

Par définition et analogie, un arbre possède :

— une *racine*, correspondant ici au nom de la fonction ou prédicat.

— des *branches*, correspondant à chacun des arguments de la structure.

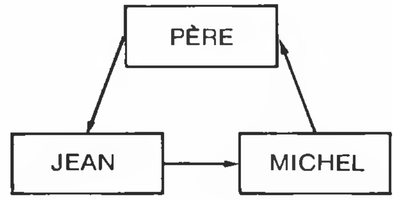
Si ces branches sont elles-mêmes des structures, on les appelle des *nœuds*, qui donneront également naissance à d'autres branches.

Une branche est donc, soit un terme élémentaire de la structure, soit une structure imbriquée. La première branche est reliée à la racine ou au nœud par une flèche descendante (↙), la dernière par une flèche ascendante (↗).

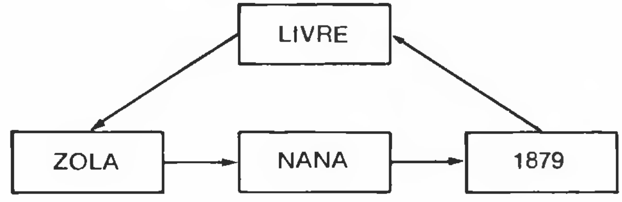
On convient de représenter les arguments d'une même structure au même niveau horizontal. On les relie par une flèche →.

Si nous prenons les exemples ci-dessous, nous pouvons les visualiser sous forme d'arbre, de la façon suivante :

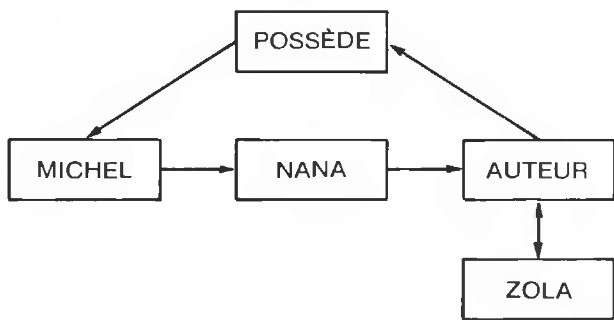
PÈRE (JEAN, MICHEL).



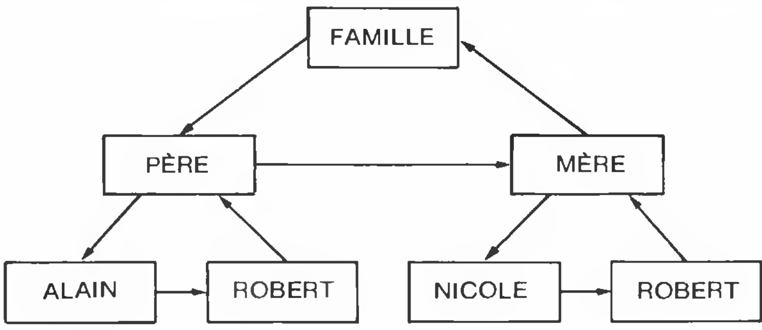
LIVRE (ZOLA, NANA, 1879).



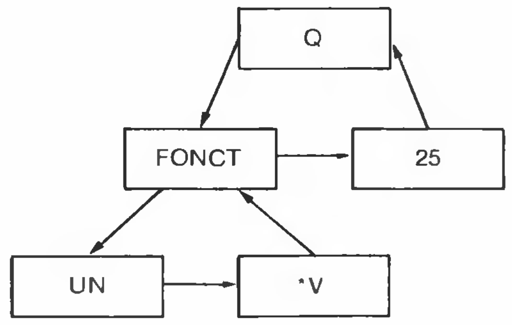
POSSÈDE (MICHEL, NANA, AUTEUR (ZOLA)).



FAMILLE (PÈRE (ALAIN, ROBERT), MÈRE (NICOLE, ROBERT)).



Q (FONCT (UN, *V), 25).



Remarques :

- Pour reconstituer une structure à partir d'un arbre, il faut effectuer un cheminement en *profondeur*, de gauche à droite, et en même temps de haut en bas. On s'intéresse d'abord à la branche de gauche tant qu'elle possède des nœuds ou des branches, puis seulement après on passe à la branche suivante du même niveau.
- Ces arbres sont dits "N-aires", c'est-à-dire que le nombre de branches qu'ils possèdent dépend du nombre et du type des arguments de la structure correspondante.

LA NOTION DE LISTE.

Une autre caractéristique intéressante est celle des termes composés particuliers, dont les arguments spécifiques sont appelés *listes de termes*.

Mais d'abord, qu'est-ce qu'une liste ? Une liste est une suite *ordonnée* d'éléments. C'est une structure très classique en programmation non numérique. Une liste peut contenir n'importe quel type d'éléments, c'est-à-dire tout type de termes (constantes, variables, structures, ou même d'autres listes), ce qui permet de travailler sur des listes de longueur ou de contenu inconnu à l'avance.

Une liste permet de représenter pratiquement toutes les structures que l'on peut avoir à manipuler en calcul symbolique.

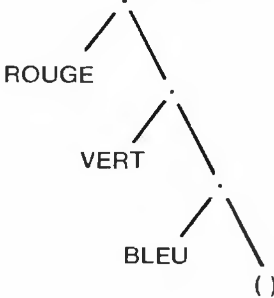
Sa puissance est telle qu'un langage de programmation (appelé LISP) a été conçu pour ne travaille qu'à partir de constantes et de listes. Il était normal que PROLOG utilise ce type de structure.

Par définition, une liste est soit la liste vide (elle n'a pas d'éléments), notée ( ) ou NIL, soit une structure à deux composants ou arguments : la tête et la queue, reliés par le symbole fonctionnel ou *constructeur de liste* ".".

Exemples de listes :

La liste composée du seul caractère 'P' s'écrit (P ( )) et son arbre est :

La liste composée des atomes (ROUGE, VERT, BLEU) peut s'écrire (ROUGE, (VERT, (BLEU, ( )))) et se représente par :



Remarques :

- On voit que l'ordre dans lequel on énonce les composants d'une liste est important.
- Une liste est un cas particulier d'arbre, appelé arbre *binaire* : la racine et chaque nœud de l'arbre possèdent deux branches et exactement deux. Les éléments qui ne possèdent pas de branches sont appelés *feuilles* de l'arbre : ils correspondent aux composants de la liste et à la liste vide (fin de la liste).

— Les listes de termes. Par définition, une liste de termes est soit :

- la liste vide : c'est-à-dire qu'elle ne contient aucun terme;
- une liste contenant au moins un terme.

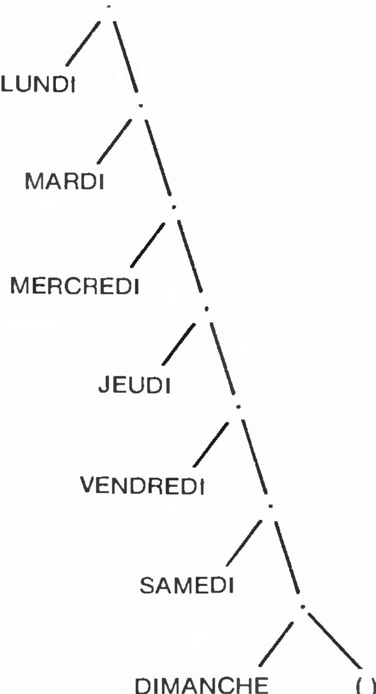
Syntaxe :

- la liste vide sera notée ( ) ou NIL.
- une liste de n termes t1, t2, ..., tn sera représentée par la liste (t1, t2, ..., tn). Cette représentation est plus aisée et claire que la précédente, mais sa signification reste la même.

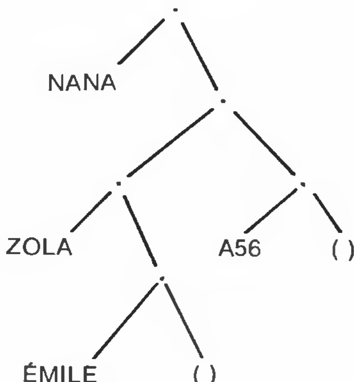
Les listes ci-dessus s'écrivent désormais (P), (ROUGE, VERT, BLEU).

Exemples :

(LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE) est la liste des jours de la semaine :



(NANA, (ZOLA, ÉMILE), A56) est la liste, composée elle-même d'une liste, donnant pour un livre, les nom et prénom de son auteur, et sa référence en bibliothèque.



Remarquons que si l'on inverse certains termes, ce n'est plus la même liste. (On rappelle qu'une liste est une suite *ordonnée* d'éléments).

— **Le séparateur ";"**. Souvent, on ignore la longueur réelle d'une liste, ou on a besoin de s'intéresser uniquement à certains éléments de celle-ci. Donc l'énoncé explicite de tous les éléments de la liste devient peu performant et pertinent. On voudrait alors pouvoir représenter le ou les premiers éléments de la liste, et le "reste" (qui est une liste).

On utilise pour cela, le séparateur ";", à la place de la virgule, pour différencier la partie *tête* de liste de la partie *queue* de liste.

Syntaxe :

(t1, t2, ..., tn; tr) où t1 ... tn représentent les n premiers termes de la liste (ou tête), et tr le reste de la liste (ou queue).

Évidemment, cette syntaxe ne permet d'utiliser qu'un point-virgule dans une liste ; sinon il y aurait contradiction dans notre énoncé.

Cette représentation est particulièrement intéressante, lorsque tr est une variable : cette variable représente alors la liste des éléments constituant la queue de la liste initiale, cette liste pouvant être de longueur inconnue.

Exemples :

(LUNDI, MARDI, *JOUR, JEUDI, VENDREDI ; *WEEK-END) :

*JOUR représente le MERCREDI et *WEEK-END représente la liste (SAMEDI, DIMANCHE).

(LUNDI, MARDI ; *JOUR ; DIMANCHE) :

cette écriture ne correspond qu'à une contradiction, puisqu'il y a deux points-virgules. Il y a erreur dans la syntaxe de la liste : en effet, où sont ici la tête et la queue de la liste ?

(JANVIER ; FÉVRIER) :

la queue de la liste est FÉVRIER).

Nous pourrions représenter ici la structure SEMAINE par :

SEMAINE ((LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE)).

Le prédicat SEMAINE a un seul argument, qui correspond à la liste des jours de la semaine.

Si nous posons la question :

```
$?: SEMAINE ((*UN, *DEUX, *TROIS ; *RESTE)).
*UN    = LUNDI
*DEUX  à MARDI
*TROIS = MERCREDI
*RESTE = (JEUDI, VENDREDI, SAMEDI, DIMANCHE).
--SUCCÈS--(1)--
```

La demande de résolution montre ici comment sélectionner la queue d'une liste. Ainsi, nous pouvons traiter les éléments d'une liste, alors que sa définition montrait qu'elle correspondait à un argument unique. On remarque, une fois de plus, que l'ordre dans lequel sont énoncés les termes d'une liste est primordial et significatif.

— **Les chaînes de caractères.** Un cas particulier de liste est la **chaîne de caractères.** Mais par tradition et par commodité, on préfère utiliser une notation différente pour représenter une chaîne. En effet, la chaîne de caractères "bonjour" (énoncée entre guillemets) représente la liste ('b', 'o', 'n', 'j', 'o', 'u', 'r'). On comprend aisément l'intérêt de représenter ce mot par une chaîne de caractères et non par une liste ! Mais nous vous conseillons de les utiliser avec parcimonie, car elles consomment beaucoup de place mémoire : "bonjour" est une liste qui occupe 15 termes en mémoire (un terme par caractère / un terme pour la fin de chaîne).

Cependant, l'utilisation des chaînes de caractères s'avère nécessaire lorsqu'on veut comparer des caractères entre eux.

Cas particulier : une chaîne réduite à un seul caractère s'écrit entre apostrophes.

3.1.1.4 - Les textes

Il est fréquent d'avoir à afficher un texte à l'écran par programme. Par exemple, lorsque vous voulez que le système vous annonce : "Une solution au problème est :". On ne veut plus ici traiter une chaîne de caractères, mais un groupe de mots dans son ensemble (aucun traitement ne sera effectué dessus). Pour cela, on écrit le

texte entre apostrophes. Ce texte sera considéré comme un groupe à part entière, non divisible et non analysé par l'interpréteur. On peut donc considérer qu'un texte est un type particulier de constante.

Par exemple, si l'on déclare le fait P ('UN'), et que l'on demande ensuite à l'interpréteur :

$?: P('UN')
--ÉCHEC--

Ainsi, l'interpréteur ne sera pas capable de reconnaître que deux textes — apparaissant à deux endroits du programme — sont identiques. Il convient donc de réserver le type *texte* à l'édition de messages à l'écran ou à l'écriture de longues chaînes de caractères.

Un texte occupe un octet mémoire par caractère et un terme pour le texte lui-même.

En pratique, l'édition de messages par programme se fait à l'aide du prédicat prédéfini PUT.

Exemples :

MESSAGE 1 : PUT ('BONJOUR A TOUS').
$?: MESSAGE1 BONJOUR A TOUS :

la résolution entraîne l'écriture du message directement à la suite de la question.

Dans un texte, on peut inclure des *caractères de contrôle*, sous forme hexadécimale, en les faisant précéder du symbole "$" (voir liste des caractères de contrôle en annexe). Ces caractères permettent de formater l'édition. Mais ils peuvent entraîner des problèmes à la résolution, si on utilise la commande ou primitive d'édition sur imprimante PRINTON.

Exemples :

$0D : retour au début de la ligne
$0A : aller à la ligne.

Ces deux caractères de contrôle combinés correspondent au prédicat prédéfini LINE.

Si on doit insérer, dans le texte, des caractères qui peuvent conduire à une certaine ambiguïté (comme l'apostrophe qui indique normalement la fin du texte, ou le $, la présence d'un caractère de contrôle), on les duplique.

MESSAGE 2 : PUT ('$0A$0DL "AN DERNIER LE $$ ÉTAIT PLUS HAUT !")
$?: MESSAGE 2
L'AN DERNIER LE $ ÉTAIT PLUS HAUT !

MESSAGE 3 : PUT ('$0A$0DA$0A$0DB')
$?: MESSAGE 3
A
B

```
MESSAGE 4 : PUT ('$0A$0DBONJOUR$0AA$0ATOUS!')
$?: MESSAGE 4
BONJOUR
A
TOUS !
```

RAPPEL : DIFFÉRENCES ENTRE TEXTES ET CHAINES DE CARACTÈRES

– UN TEXTE EST ÉNONCÉ ENTRE APOSTROPHES. IL N'EST PAS ANALYSÉ PAR L'INTERPRÉTEUR. IL OCCUPE PEU DE PLACE MÉMOIRE : UN OCTET PAR CARACTÈRE ET UN TERME POUR LE TEXTE.

– UNE CHAINE DE CARACTÈRES EST ÉNONCÉE ENTRE GUILLEMETS (SAUF SI ELLE NE CONTIENT QU'UN CARACTÈRE, QUE L'ON ÉCRIT ENTRE APOSTROPHES). CHAQUE CARACTÈRE EST ANALYSÉ PAR L'INTERPRÉTEUR. ELLE PEUT DONNER LIEU A DES COMPARAISONS AVEC D'AUTRES CHAINES. MAIS SON INCONVÉNIENT EST QU'ELLE OCCUPE BEAUCOUP DE PLACE MÉMOIRE : DEUX TERMES PAR CARACTÈRE ET UN POUR LA FIN DE CHAINE.

Nous connaissons désormais tous les types de "mots" ou termes du langage. Reste à savoir désormais comment on les assemble pour former des phrases du langage, et concevoir ainsi des programmes.

3.1.2 - Les programmes

Un programme est composé d'un ensemble de *faits* (relations établies entre des termes et constituant une base de données) et d'un ensemble de *règles*, qui ne sont plus des faits élémentaires, comme nous l'avons vu, mais des faits "conditionnels" (une relation entre termes est vraie si...). Les faits et les règles sont représentés par des *clauses*. L'ensemble des clauses de même nom constitue un *sous-programme*. Par déduction, un programme est constitué d'un ensemble de sous-programmes. Nous allons détailler maintenant ces différents concepts.

3.1.2.1 - Les prédicats

Un prédicat sert à définir une relation entre des objets du domaine sur lequel on travaille. Il ressemble aux structures qui ont été définies plus haut, à ceci près qu'il possède en plus une valeur "logique", c'est-à-dire la propriété de pouvoir être évalué à *vrai* ou à *faux*, selon les objets auxquels on l'applique. Ainsi un prédicat prendra une valeur différente selon le cas.

En fait, PROLOG n'évalue pas directement le sens d'un prédicat. Pour ce faire, il le compare dans sa syntaxe et dans sa forme à ce qu'il

connaît du domaine. Ce n'est donc pas une analyse sémantique qu'il effectue, mais une *comparaison dans la syntaxe et la similitude des "formes"*. Un prédicat ne sera évalué que lorsqu'on aura essayé de le faire correspondre à un fait de la base.

On peut donc considérer qu'un prédicat est un "terme composé". Par analogie, un prédicat pourra donc posséder des arguments de tout type (constantes, variables, prédicats ou listes de prédicats, comme nous le verrons pour certaines primitives de manipulation de clauses).

Par définition, il existe deux types de prédicats :
- les prédicats définis par l'utilisateur,
- les prédicats *prédéfinis* ou *"primitives"*, destinés à faciliter la programmation par l'utilisateur. Ce sont des prédicats connus à priori du système. Ils permettent d'effectuer certaines "actions" couramment utilisées. Nous les examinerons au § 3.1.2.4, p. 108.

La syntaxe d'un prédicat est :

<nom-de-prédicat> (t1, t2, t3, ..., tn).

Le nom-de-prédicat est un atome, les termes t1, t2, ..., tn sont ses arguments (il peut y en avoir de 0 à 15).

Certains noms de prédicats prédéfinis ne comportent qu'un seul caractère, qui est un symbole spécial (par exemple, les prédicats arithmétiques et de relations d'ordre).

Exemples :

PERE (JEAN, MICHEL) : affirme que JEAN est le père de MICHEL.

(*X, 34) : son évaluation dépend de l'objet que représente la variable *X au moment de l'appel (">" est le prédicat prédéfini de comparaison "supérieur" entre les nombres entiers).

Si le contenu de *X est un nombre entier supérieur à 34, alors le prédicat est évalué à vrai, sinon il est faux.

AIME (MARC, *X) : ne sera vrai que si on réussit à trouver un objet de la base qui puisse correspondre à *X (c'est-à-dire que MARC aime effectivement quelque chose), sinon le prédicat est faux.

3.1.2.2 - Les clauses

Une clause est une *affirmation* établissant une relation entre plusieurs termes, en utilisant les prédicats vus précédemment. Elle se décompose en un prédicat de tête, et zéro ou plusieurs prédicats de queue.

● S'il n'y a pas de prédicat de queue, la clause est une affirmation du domaine, donc un fait de la base : elle est vraie de façon inconditionnelle.

Exemple :

AIME (ANNE, ANIMAUX) :
affirme qu'ANNE aime les ANIMAUX.

- S'il y a un ou des prédicats de queue, la clause est une affirmation conditionnelle du domaine, ou règle du domaine (le prédicat de tête est vrai si...).

La syntaxe d'une clause est :

<Prédicat de tête> : <P1> & <P2> & ... & <Pn>.

- Le prédicat de tête est aussi appelé *tête de la clause*, les prédicats de queue, *corps de la clause*.
- La tête et le corps de la clause sont séparés l'un de l'autre par le symbole ":".
- La clause se termine par un point.

L'interprétation que l'on peut donner d'une clause est :

<Prédicat de tête> est VRAI si tous les prédicats P1, P2 ... à Pn sont vrais simultanément, pour les termes auxquels ils se rapportent au moment de l'évaluation. C'est pour cela que les prédicats de queue sont séparés par des "&", qui indiquent des conjonctions de condition.

—Si le corps de la clause est vide, alors le prédicat de tête (ou plus simplement le prédicat) est vrai systématiquement : c'est un fait.

— Un prédicat de tête d'une règle comporte en général des variables dans ses arguments. Celle-ci se retrouvent aussi dans un ou plusieurs prédicats du corps de la clause. Ces variables servent à "passer" des valeurs et à les propager entre les prédicats de tête et de corps d'une clause. Une clause ne peut comporter dans son énoncé plus de 16 variables différentes.

ATTENTION :
DES VARIABLES DE MÊME NOM NE SONT IDENTIQUES QUE DANS LA LIMITE DE LA PORTÉE DE LA CLAUSE. C'EST-A-DIRE QUE SI ELLES APPARAISSENT SOUS LA MÊME FORME DANS UNE AUTRE CLAUSE, ELLES N'EN AURONT PAS POUR AUTANT TOUJOURS LA MÊME SIGNIFICATION.

Exemples :

PERE (JEAN, MICHEL) : affirme que JEAN est le père de MICHEL
PERE (MICHEL, PIERRE)
PERE (JEAN, ALBERT)

GD-PERE (*GP, *PF) : PRE (*GP, *P) & PERE (*P, *PF) : le prédicat GD-PERE (*GP, *PF) représente la tête de la règle et les prédicats PERE, le corps de la règle.

Cette règle permet de définir la relation "être grand-père paternel de". Elle se lit : un individu *GP est grand-père paternel d'un autre individu *PF, si il existe un troisième individu *P tel que *GP soit le père de *P et *P soit le père de *PF.

On voit ici que les variables présentes dans le prédicat de tête réapparaissent dans le corps de la clause, c'est-à-dire qu'à un moment ou à un autre, elles représenteront le même objet. C'est le principe de *cohérence* d'une variable dans une règle.

REPRÉSENTATION D'UNE CLAUSE

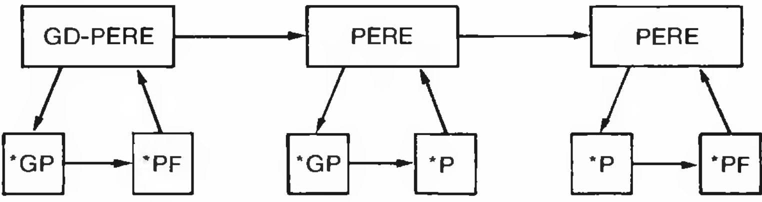
De même que nous avons visualisé et représenté une structure ou une liste sous forme d'arbre, nous pouvons représenter une clause de cette façon, (pour voir comment sont établies les relations entre les objets), sachant que :

—les prédicats de tête et du corps de la clause sont placés au même niveau horizontal ;
—le premier prédicat correspond au prédicat de tête, les suivants aux prédicats du corps de la clause (reliés par "&").

Exemples :

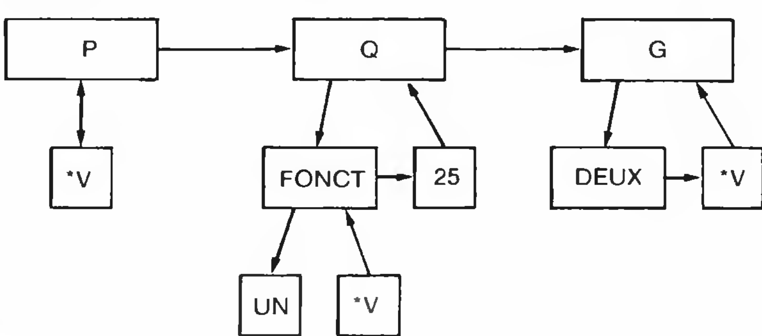
GD-PERE(*GP, *PF) : PERE (*GP, *P) & PERE (*P, *PF)

s'écrit :



P (*V) : Q (FONCT (UN, *V), 25) & G (DEUX, *V)

s'écrit :



Chaque terme initial de la tête de la clause est, comme nous le verrons, "pointé" par un élément de la liste des clauses CLAUSES (un élément pour chaque clause).

Ainsi cette liste permet de retrouver toutes les clauses énoncées dans un programme (voir § 3.1.2.3, p. 100).

Insistons ici sur la **différence entre une clause et un prédicat.**

- Une clause est une affirmation par rapport au domaine, donc elle n'est pas à évaluer. Elle correspond à une définition générale (comme par exemple : que signifie être grand-père paternel), ou à une connaissance établie du domaine (JEAN est le père de MICHEL).

- Le rôle d'un prédicat, lui, est par contre d'être évalué (sauf s'il correspond à un fait qui est vrai par définition) : c'est ce que nous faisons lorsque nous utilisons une règle pour vérifier si des relations entre objets sont vrais. Le prédicat GD-PERE (JEAN, MICHEL) est vrai si les deux prédicats PERE, énoncés dans le corps de la clause, sont vrais simultanément. Quand nous demandons une résolution, nous énonçons en fait un ou des prédicats et demandons leur évaluation. On va chercher, pour chacun, un fait ou une règle qui puisse lui être appliqué. Selon le résultat de la recherche, le prédicat sera évalué à vrai (SUCCES) ou à faux (ECHEC).

UNE CLAUSE PARTICULIÈRE : LA DEMANDE DE RÉSOLUTION

Cela explique la syntaxe utilisée pour énoncer une question ou demander une résolution en PROLOG :

```
$?: PERE (JEAN, MICHEL)
$?: BIBI : GD-PERE (JEAN, *X).
```

Cette syntaxe est celle d'une clause, dont le nom de prédicat de tête commence par le symbole spécial "?" (en général, par souci de simplicité et de clarté, on limite ce nom à l'unique symbole "?") et n'a pas d'arguments.

Le corps de la clause est un prédicat ou une conjonction de prédicats correspondant à la question, donc à une demande d'évaluation sur ce ou ces prédicats.

A chaque fois que PROLOG rencontre "?", il reconnaît ces symboles comme signifiant une demande de résolution (c'est pour cela en particulier, qu'ils ne doivent pas apparaître dans le nom d'un atome), et sait que la question est énoncée dans le corps de la clause. Il sait donc que son travail va consister à évaluer les prédicats contenus dans le corps de la clause. Ainsi, l'interpréteur sait que la clause particulière dont la tête est réduite au symbole "?", définit une question, un but à atteindre.

Exemple :
Sachant que l'exemple précédent est connu du système, si PROLOG rencontre :

```
$?: GD-PERE (JEAN, PIERRE)
```

il reconnaît immédiatement qu'il a affaire à une demande de résolution, donc qu'il doit évaluer le corps de la clause, qui se résume ici au prédicat GD-PERE (JEAN, PIERRE). Pour ce faire, il va rechercher un fait ou une règle qui lui corresponde dans la base.

Il se trouve l'unique règle :

GD-PERE (*GP, *PF) : PERE (*GP, *P) & PERE (*P, *PF)

dont le prédicat de tête correspond à la question. Par définition, ce prédicat est vrai si tous les prédicats du corps de la règle sont vrais simultanément. Donc son objectif va être désormais d'évaluer PERE (JEAN, *P) et PERE (*P, PIERRE). Il va donc commencer par évaluer PERE (JEAN, *P) et PERE (*P, PIERRE). Il va donc commencer par évaluer PERE (JEAN, *P), c'est-à-dire essayer de trouver un fait ou une règle qui lui corresponde. Ici il trouve le fait PERE (JEAN, MICHEL). *P est désormais instancié par MICHEL. Son dernier objectif va être de satisfaire le prédicat PERE (MICHEL, PIERRE). C'est un fait de la base, donc ce prédicat est vrai. Comme les prédicats du corps de la clause sont vrais tous les deux en même temps, le prédicat de tête GD-PERE (JEAN-MICHEL) est vrai, donc le système a réussi à confirmer la question initiale.

Résoudre un problème en PROLOG revient donc à évaluer toute une série de prédicats, en espérant pouvoir les rendre vrais ou les satisfaire simultanément. Les prédicats présents dans la question sont appelés des buts.

3.1.2.3 - Les sous-programmes

En général, on ne peut définir une relation par l'énoncé d'un seul fait ou d'une seule règle. On aura besoin d'énoncer plusieurs clauses de même nom de prédicat de tête, plusieurs alternatives, regroupant les affirmations et les définitions concernant cette relation dans le domaine.

Quel que soit l'ordre dans lequel on saisit les clauses d'un programme, l'interpréteur "stocke" ensemble les clauses de même nom. Un ensemble de clauses de même nom groupées de la sorte s'appelle un **sous-programme** ou **paquet de clauses.** Chaque nouvelle clause saisie est placée à la fin du sous-programme correspondant.

L'évaluation d'un prédicat revient à évaluer successivement, dans l'ordre où elles apparaissent, les différentes clauses composant le sous-programme correspondant. L'ordre dans lequel sont énoncées les clauses est donc important. On prendra soin en particulier d'énoncer les faits avant les règles, et de faire attention aux problèmes de récursion (c'est-à-dire lorsqu'une clause se rappelle elle-même).

Par contre, l'ordre dans lequel on saisit les différents sous-programmes n'a pas d'importance ici.

Prenons deux exemples de récursivité :

1 - Soient les clauses énoncées dans l'ordre suivant :
HUMAIN (*X) : HUMAIN (*Y) & MERE (*X, *Y)
HUMAIN (EVE).

Ces clauses donnent les deux alternatives pour définir la relation "être un humain". Un objet est un être humain si, soit il existe un autre être humain qui est sa mère, soit cet objet est EVE (qui par définition n'a pas de mère, et est donc considérée comme une exception). Nous allons voir que l'ordre dans lequel ont été énoncées les clauses est impropre à la résolution.

Si nous demandons la résolution :
$?: HUMAIN (*X)
afin de connaître tous les êtres humains répertoriés, que va-t-il se passer ?

PROLOG utilisera d'abord la première clause qu'il va rencontrer, et qui est la règle HUMAIN (*X) : HUMAIN (*Y) & MERE (*X, *Y). Il essayera de la satisfaire, en satisfaisant d'abord le premier but du corps de la règle, qui est HUMAIN (*Y).

Ce but est, on le voit, identique au précédent, donc en essayant de satisfaire ce but, c'est une fois de plus, la règle générale qu'il va essayer de resatisfaire, etc. Ce rappel de la règle générale est appelé appel *récursif*.

Ici, PROLOG n'arrivera jamais à atteindre le fait HUMAIN (EVE) la tâche à résoudre est infiniment longue et ne pourra ni réussir, ni échouer. On dit que le programme *boucle sur lui-même,* ce que l'interpréteur nous annonce par le message ERR5 : MEMOIRE (à un moment ou à un autre, la pile de résolution est pleine). Cela aurait pu être évité, si l'on avait pris soin d'énoncer la clause HUMAIN (EVE), qui constitue une condition d'arrêt, avant la règle. L'ordre dans lequel on énonce les clauses est donc important.

2 - Supposons que nous voulions définir la relation APPARTIENT d'appartenance d'un terme à une liste. Le système nous répondra -SUCCES- si le terme appartient effectivement à la liste, -ECHEC- sinon. Pour cela, nous allons devoir comparer successivement le terme recherché à chaque terme de la liste. Cette recherche est basée sur la propriété des listes de termes, qui possèdent une tête et une queue (voir § 3.1.1.3, p. 86).

- Si le terme est identique à la tête de la liste, alors on a trouvé ce terme dans la liste.

- Sinon, on va rechercher ce terme dans la queue de la liste, en la considérant comme la nouvelle liste (donc on rappelle simplement la relation APPARTIENT sur la queue de la liste). Pour saisir ce type de relation récursive, il faut se poser la question de savoir quelle est la condition d'arrêt de recherche (sinon le programme risque de boucler indéfiniment). Ici, il y a deux conditions d'arrêt pour le

prédicat APPARTIENT. Soit l'objet que nous cherchons est dans la liste, soit il n'y est pas (c'est-à-dire qu'on a parcouru toute la liste sans le trouver).

— La première condition d'arrêt va être définie dans une première clause : nous avons trouvé une solution si le premier argument d'APPARTIENT (qui correspond au terme recherché) coïncide avec la tête du deuxième argument (correspondant à la tête de la liste), ce que nous énonçons par le fait :

APPARTIENT (*X, (*X, *Y)).

— La deuxième condition d'arrêt va être intégrée dans la deuxième clause APPARTIENT, qui définit en même temps l'appel récursif :

APPARTIENT (*X, (*T, *Q)) : APPARTIENT (*X, *Q).

Ce qui signifie :

- Si le terme *X ne correspond pas à la tête de la liste (donc à son premier élément), comme nous avons pu le vérifier en utilisant la première clause, alors on rappelle le prédicat APPARTIENT avec la liste correspondant à la queue de la liste initiale. Il est à remarquer que chaque fois que APPARTIENT tente de se resatisfaire lui-même, le but reçoit une liste plus petite (la queue d'une liste est toujours une liste plus courte que la liste originale).

- Si la liste d'appel est une liste qui n'a plus qu'un seul élément (donc dont la queue est la liste vide, par définition), le rappel du prédicat APPARTIENT aura pour liste la liste vide, donc échouera puisque, par définition, la liste vide n'a ni queue ni tête. On ne trouvera aucun fait, aucune tête de règle qui corresponde à APPARTIENT(*X, NIL). Donc on aura atteint la condition d'arrêt finale.

Les deux clauses précédentes devront être saisies dans ce sens, afin que l'interpréteur teste d'abord si le caractère coïncide avec la tête de la liste, sinon il utilisera la deuxième clause. Le sous-programme APPARTIENT s'écrit donc :

APPARTIENT (*X, (*X, *Q)).
APPARTIENT (*X, (*T, *Q)) : APPARTIENT (*X, *Q).

REPRÉSENTATION D'UN SOUS-PROGRAMME EN MÉMOIRE

Voyons maintenant comment cela se passe en pratique. Supposons que nous voulions saisir le sous-programme suivant, composé des clauses de même nom :

LIVRE (HUGO, 'LES MISERABLES', 1862)
LIVRE (VERLAINE, 'SAGESSE', 1881)
LIVRE (ZOLA, 'NANA', 1879).

Ici, nous n'avons que des faits, qui sont ordonnés par ordre alphabétique sur les noms d'auteurs. En supposant que cela soit voulu, nous allons regarder comment ces clauses sont mémorisées par l'interpréteur, et pourquoi l'ordre influe sur les évaluations qui pourront être faites.

A chaque fois que nous saisissons une clause et que nous la validons, l'interpréteur affiche le nombre de *termes* (au sens d'unité minimum de représentation d'un objet en mémoire (4 octets), et non pas dans le sens de l'objet lui-même) qu'elle occupe en mémoire. Ce nombre correspond donc au nombre de "cases" mémoire utilisées pour représenter une clause en mémoire.

L'interpréteur reconnait une clause comme étant une définition de relation, une affirmation. Qui dit relation, dit liaison entre des objets. Ces objets sont les arguments de la clause.

Reprenons l'exemple précédent pour voir comment l'interpréteur établit ces relations en mémoire :

● Pour mieux comprendre ce qui se passe, avant toute saisie tapez d'abord la commande CLEAR pour réinitialiser l'interpréteur, c'est-à-dire lui faire "oublier" tout ce qu'il a pu stocker précédemment (en actionnant les commandes STAT et DIC, on voit que l'occupation de la mémoire a été remise à zéro).

● Rappelons ici brièvement ce qui a été dit sur l'effet de la commande STAT (voir § 1.3.1, p. 46). Elle donne d'une part la capacité mémoire (-1-MÉMOIRE) de l'interpréteur, et d'autre par l'état de la mémoire (-2-OCCUPATION) qui évolue en fonction du programme que vous saisissez. Nous utiliserons et examinerons ici plus précisément les structures "statiques", c'est-à-dire celles qui contiennent le programme en mémoire. Rappelons la définition de chacune :

— **DICO :** c'est le dictionnaire de la mémoire, qui contient tous les noms de prédicats ou d'atomes que vous aurez créés. Ici on donne le nombre de noms mémorisés par l'interpréteur (sachant qu'un nom occupe 13 octets).

La commande DIC affiche plus précisément le contenu du dictionnaire, par ordre de saisie du programme.

— **CLAUSES :** correspond au nombre de clauses entrées par l'utilisateur. A chaque clause est associé un élément (occupant 5 octets) qui représente l'ensemble de la clause.

— **TERMES :** correspond au nombre de termes (occupant 4 octets) contenus en mémoire. Ce nombre est initialisé à 4, pour les besoins de l'interpréteur.

— **TEXTE :** correspond à l'occupation maximale des textes saisis, en mémoire.

● Ces structures sont appelées, *piles*, car certaines fonctionnent par empilements successifs chaque fois qu'on saisit une nouvelle clause.

● Voyons maintenant comment les choses se passent dans le cas qui nous concerne. Notre démarche sera, pour chaque clause de l'exemple à saisir, de voir ce que cela entraîne au niveau de l'occupation en mémoire (par action des commandes STAT et DIC).

Chaque clause saisie occupe ici quatre "termes" mémoire : un pour le nom du prédicat, un pour l'atome correspondant au premier

argument, un pour le deuxième argument qui est un texte, un pour le troisième argument qui est un nombre entier.

● Saisissons la première clause LIVRE (HUGO, 'LES MISERABLES', 1862) et regardons ce qui a été modifié dans les structures statiques de la commande STAT :

```
$ STAT
-2-OCCUPATION
DICO 2 SUR 315
CLAUSES 1 SUR 409
TERMES 8 SUR 4352
TEXTE 15 SUR 1024.
```

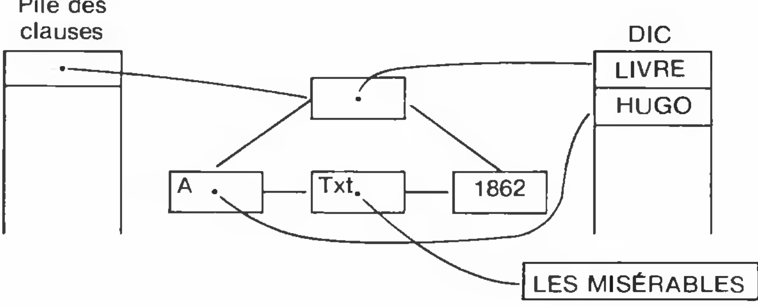
Le dictionnaire comporte désormais deux noms, on nous annonce qu'une clause a été saisie et que huit termes sont mémorisés (en se rappelant que le nombre de termes mémoire est initialisé à quatre, cela veut dire que nous avons saisi nous-mêmes quatre termes mémoire), de plus le texte occupe 15 octets (en se rappelant qu'on compte un octet par caractère, plus un pour le message).

● La commande DIC nous précise plus exactement ce que contient le dictionnaire :

```
1 LIVRE 1 C
2 HUGO 1 A.
```

Chaque nom est précédé d'un numéro correspondant à son ordre d'apparition dans la saisie du programme. Le nombre qui suit le nom correspond au nombre de fois où celui-ci a été rencontré dans le programme (ici, une fois). La lettre C ou A indique plus précisément le type de chaque nom cité. Un C indique que le nom est celui d'une tête de clause, un A indique que le nom est celui d'un atome d'un prédicat ou d'une structure.

Nous avons vu que ces structures se comportent comme des piles. Visualisons ce qui se passe en mémoire, une fois que la clause a été saisie et analysée par l'interpréteur :



Le premier élément de la liste des clauses pointe sur le terme initial de la tête de la clause. Cette clause conserve les liaisons qui relient ces termes (comme nous l'avons vu précédemment), mais le contenu de ces termes, mis à part les nombres entiers qui sont stockés directement, est véritablement stocké soit dans le dictionnaire, soit dans un texte. Le contenu de chaque terme est accessible et connu, par l'utilisation de pointeurs.

Vous vous demandez certainement le pourquoi d'une telle organisation. Allons plus loin pour le comprendre, en saisissant la clause suivante :

LIVRE (VERLAINE, 'SAGESSE', 1881).

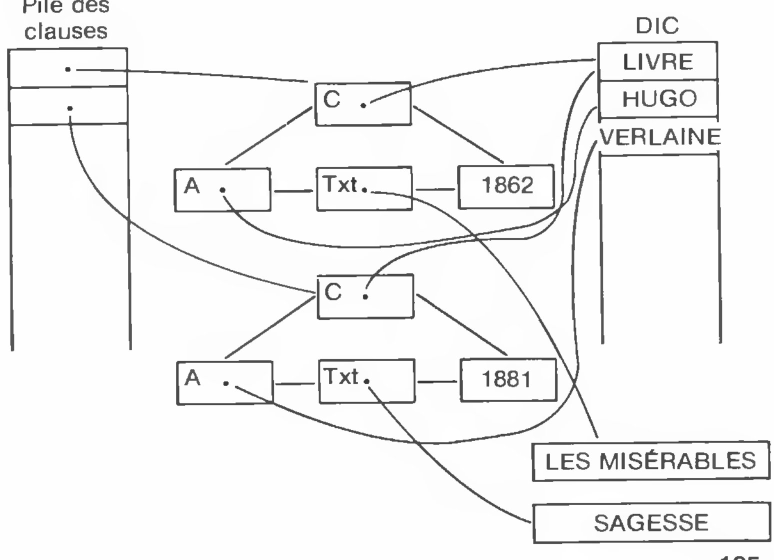
Regardons ce qui change au niveau occupation mémoire (par les commandes STAT et DIC). Les modifications sont représentées en caractères gras :

$ STAT

-2-OCCUPATION
DICO **3** SUR 315
CLAUSES **2** SUR 409
TERMES **12** SUR 4352
TEXTE **23** SUR 1024

$ DIC
1 LIVRE **2** C
2 HUGO 1 A
3 **VERLAINE 1 A**.

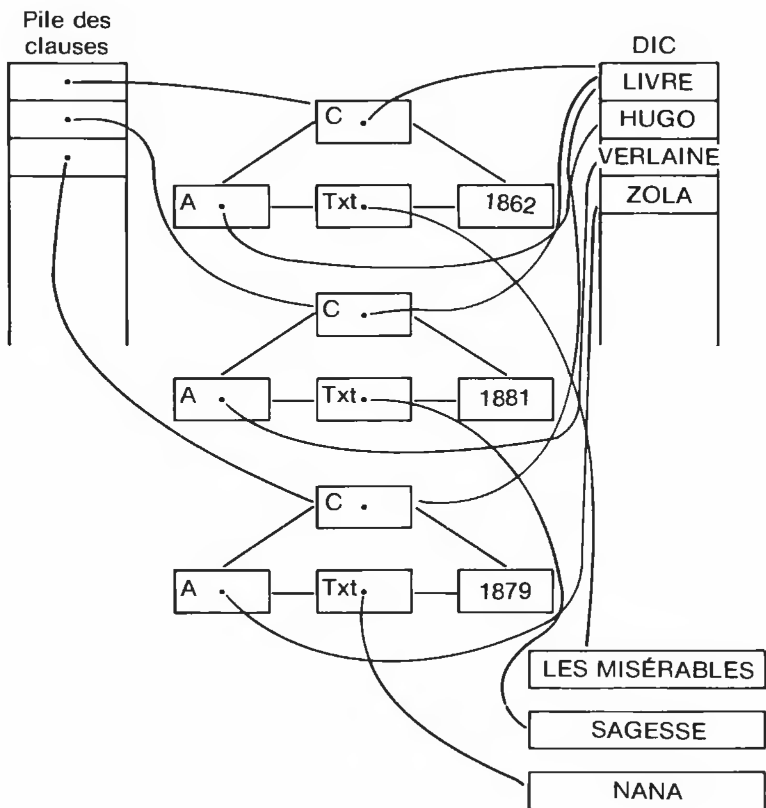
En effet, nous avons saisi une clause de plus, avec ce qu'elle comporte comme nouveaux termes.



On commence à voir l'intérêt d'une telle représentation en mémoire. Le fait d'utiliser des pointeurs permet de clarifier et d'alléger la configuration de notre programme. L'interpréteur ne mémorise réellement qu'une fois les termes utilisés, qu'ils soient ou non différents, et les retrouve par le biais des liaisons établies par les pointeurs (sauf pour les textes qui sont considérés comme des entités à part entière : leur représentation en mémoire ne permet donc pas à l'interpréteur de vérifier qu'ils sont identiques ou non. Voir § 3.1.1.4, p. 93). Ainsi il n'y a pas de redondance dans l'occupation mémoire. mémoire.

Remarquons que l'ordre dans lequel on saisit les clauses est important, comme nous l'avons déjà dit, car la pile des clauses se remplit au fur et à mesure de la saisie. La première est liée à la deuxième et vice-versa, dans l'ordre de saisie.

De la même façon, en saisissant la troisième clause LIVRE (ZOLA, 'NANA', 1879), on obtiendrait le schéma interne suivant (regardez l'évolution de l'occupation de la mémoire) :



Supposons par exemple que nous voulions maintenant supprimer la deuxième clause LIVRE (VERLAINE, 'SAGESSE', 1881) de la mémoire par la commande DEL(LIVRE, 2). Cette opération correspond, pour l'interpréteur, à supprimer tous les pointeurs relatifs à cette clause, donc toutes les relations qu'elle supposait.

Dans le cas présent :

- le deuxième élément de la pile des clauses et les pointeurs qui s'y référaient sont supprimés. Ainsi le premier élément de la pile des clauses pointe désormais sur celui qui était le troisième et qui devient désormais le second ;
- tous les pointeurs qui se référaient aux termes de la clause elle-même sont supprimés ;
- de plus, le dictionnaire prend en considération ces changements, en supprimant les noms qui ont disparu (s'ils n'existaient et n'apparaissaient que dans la clause supprimée) ou en modifiant le nombre de fois où ils apparaissaient dans le programme. On peut donc considérer que l'existence d'un nom dans le dictionnaire est liée indubitablement à l'existence d'au moins un pointeur le désignant ;
- les textes et les nombres entiers ne sont plus considérés en tant que tels, puisqu'aucun pointeur ne les désigne. Ne vous inquiétez pas de la validité de l'occupation mémoire des textes, après suppression de la clause : il se peut que le nombre de "TEXTE" n'ait pas été modifié. Cela ne veut pas dire que le texte n'a pas été supprimé de la mémoire, mais simplement que le texte actuel occupe au plus X termes.

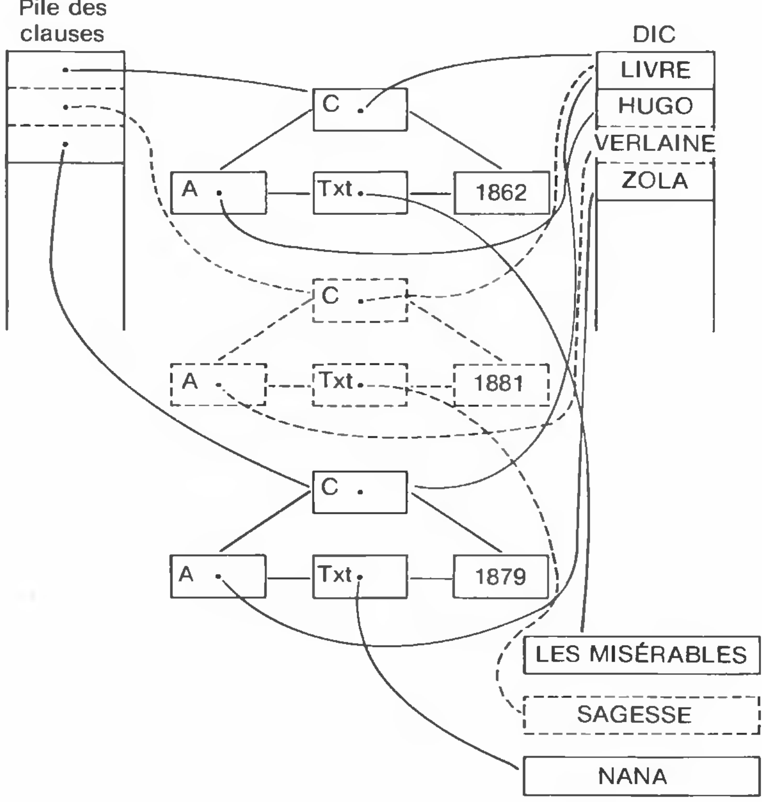
L'occupation mémoire annoncée est :

```
$ STAT

-2-OCCUPATION
DICO 3 SUR 315
CLAUSES 2 SUR 409
TERMES 12 SUR 4352
TEXTE 28 SUR 1024

$ DIC
1 LIVRE 2 C
2 HUGO 1 A
3 ZOLA 1 A.
```

Dans le cas présent notre schéma devient (les pointillés symbolisent ce qui a disparu).



3.1.2.4 - Les prédicats prédéfinis ou primitives

Heureusement pour vous, utilisateur et programmeur en PROLOG, il existe un certain nombre de clauses connues a priori de l'interpréteur. On les appelle **prédicats prédifinis** ou **primitives.**

● Ces clauses sont indispensables à la réalisation de programmes. Elles ont été "prédéfinies", parce qu'elles font partie des fonctions élémentaires du langage, dont vous aurez constamment besoin (comme par exemple, les calculs arithmétiques, les Entrées/Sorties, les manipulations de programmes, etc.).

● Elles ont la même syntaxe et les mêmes propriétés logiques que les prédicats que vous écrivez, c'est-à-dire qu'elles prennent également, après évaluation, la valeur "vrai" ou "faux" et peuvent donc être utilisées dans d'autres clauses.

● Par contre, elles possèdent en plus une particularité, qu'aucune clause que vous écrirez ne pourra avoir, qu'on appelle parfois *"l'effet de bord"* : c'est-à-dire qu'elles ont une action *irréversible*, comme par exemple écrire sur l'écran, supprimer une clause du programme en mémoire ou en ajouter une, sélectionner une clause, contrôler le travail de l'interpréteur. Cet effet est particulièrement intéressant pour toutes les opérations concernant les Entrées/Sorties et les manipulations de programmes.

On distingue deux types de primitives :

— celles qui ne peuvent être exécutées que dans un programme (c'est-à-dire qu'elles doivent apparaître dans le corps d'une clause) ;

— celles qui peuvent se comporter soit comme des commandes (vous pouvez les appeler directement), soit comme des éléments de clauses contenus dans un programme.

Nous verrons successivement :
- les primitives arithmétiques,
- les primitives de comparaison,
- les primitives de type de termes,
- la primitive "VALUE",
- les primitives de contrôle de l'interpréteur,
- les primitives de manipulation des clauses,
- les primitives d'Entrée/Sortie de texte,
- les primitives d'édition.

A - LES PRIMITIVES ARITHMÉTIQUES SUR LES NOMBRES ENTIERS

Toutes les primitives arithmétiques sont de la forme : <symbole-de-la-fonction> (<terme 1>, <terme 2>, <terme 3>.

Elles vous permettent d'effectuer les opérations élémentaires sur des nombres entiers. Chaque fonction opère sur <terme 1> et <terme 2> qui sont des nombres entiers signés, et donne le résultat dans <terme 3> (si c'est variable).

Remarque :

Les termes qui sont les arguments de l'opération (<terme 1> et <terme 2>) peuvent être des variables. Mais dans ce cas, celles-ci doivent être *instanciées* au moment de l'appel du prédicat (c'est-à-dire qu'elles doivent représenter un nombre entier signé à ce moment-là) sinon l'opération ne pourra se faire.

Les différents symboles de fonction sont :

+ pour l'addition
— pour la soustraction
* pour la multiplication
DIV pour la division entière
MOD pour le modulo.

Précisons, dans le détail, l'utilisation de ces différents prédicats :

+ : Addition : <terme 1> , <terme 2> ---> <terme 3>
Le résultat de l'addition est dans <terme 3>.

Exemples :

$?: + (4, 5, 9)
--SUCCES--(1)--
$?: + (89, 6, *L)
*L = 95
--SUCCES--(1)--
$?: + (*X, 4, *Y)
ERR 1 : EXECUTION -> erreur car l'un des opérandes *X est inconnu au moment de l'évaluation : il y a une infinité de valeurs pour *X qui satisfont ce prédicat !

— : Soustraction : <terme 1> - <terme 2> ---> <terme 3>
Le résultat de la soustraction est dans <terme 3>.

Exemples :

$?: — (6, 9, *X)
*X = — 3
--SUCCES--(1)--
$?: — (6, 3, 3)
--SUCCES--(1)--

*** : Multiplication :** <terme 1> * <terme 2> ---> <terme 3>
Le résultat de la multiplication est dans <terme 3>.

Exemples :

$?: *(9, 9,81)
--SUCCES--(1)--
$?: *(3, 9, *X)
*X = 27
--SUCCES--(1)--

DIV : Division : <terme 1> / <terme 2> ---> <terme 3>
Le résultat de la division entière est dans <terme 3> (le reste de la division est perdu). La division par zéro est détectée : il y a envoi d'un message d'erreur et abandon de l'exécution.

Exemples :

$?: DIV (10, 3, *X)
*X = 3
--SUCCES--(1)--
$?: DIV (56, O, *J)
ERR 13 : EXECUTION -> erreur division par 0.

MOD : Modulo : <terme 1> MOD <terme 2> --> <terme 3>.
La définition du modulo est la même qu'en BASIC : elle correspond au reste de la division de <terme 1> par <terme 2>.

Exemples :

```
$?: MOD (78, 7, *RESULTAT)
*RESULTAT = 1
--SUCCES--(1)--
$?: MOD (79, 79, *MOD)
*MOD = 0
--SUCCES--(1)--
```

B - LES PRIMITIVES DE RELATION D'ORDRE SUR LES CONSTANTES

Toutes ces primitives sont de la forme :
<prédicat de comparaison> (<argument 1>, <argument 2>).

Les arguments à comparer sont obligatoirement des constantes de même type, au moment de l'appel de la primitive, c'est-à-dire soit des entiers, soit des atomes, soit des caractères.

Les entiers sont comparés par ordre de grandeur, les atomes et les caractères par ordre alphabétique.

Un argument peut être une variable au moment de l'écriture du programme, mais celle-ci doit être instanciée au moment de l'appel par une constante de même type que l'autre argument.

Les prédicats de comparaison sont :

```
 <  évalué à vrai si <argument 1>  <  <argument 2>
 =  évalué à vrai si <argument 1>  =  <argument 2>
 >  évalué à vrai si <argument 1>  >  <argument 2>
<=  évalué à vrai si <argument 1> <=  <argument 2>
>=  évalué à vrai si <argument 1> >=  <argument 2>
< > évalué à vrai si <argument 1> < > <argument 2>
```

Exemples :

```
$?: > (5, 6) -> teste si 5 > 6
--ECHEC--
$?: = ('E', 'E') -> teste si 'E' = 'E'
--SUCCES--(1)--
$?: >(TRUC, TROC) -> teste si TRUC est après TROC dans
--SUCCES--(1)          l'ordre alphabétique
$?: < >(UN, DEUX) ->   vérifie si les deux atomes UN et
--SUCCES--(1)          DEUX sont différents.
```

C - LES PRIMITIVES DE CONTROLE DU TYPE D'UN TERME ET COMPARAISON

Ces prédicats permettent, en cours d'exécution, de faire des tests sur le type d'un terme, ou de comparer deux termes, pour déclencher des actions appropriées. Nous allons les définir successivement.

a. Les prédicats de contrôle du type d'un terme

Ils permettent de tester les termes de tout type. Ils fonctionnent tous selon le même principe.

VAR (<terme>)
Ce prédicat est évalué à vrai si <terme> est une variable libre (non encore instanciée par un terme autre qu'une variable) au moment de l'appel.

CONSTANT (<terme>)
Ce prédicat est évalué à vrai si <terme> est une constante ou une variable instanciée au moment de l'appel par une constante. On rappelle qu'un terme de type "texte" est une constante particulière.

INTEGER (<terme>)
Ce prédicat est évalué à vrai si <terme> est un entier signé, ou une variable instanciée par un nombre entier signé au moment de l'appel.

CHAR (<terme>)
Ce prédicat réussit si <terme> est un caractère, ou une variable instanciée au moment de l'appel par un caractère.

ATOM (<terme>)
Ce prédicat réussit si <terme> est un atome, ou une variable instanciée au moment de l'appel par un atome.

STRUCT (<terme>)
Ce prédicat réussit si <terme> est un terme composé, ou une variable instanciée par un terme composé au moment de l'appel.

Exemple :

Soient les huit clauses suivantes, du paquet TERM, qui ont chacune un argument de type différent :

```
TERM (*X0)                  - une variable libre *X0
TERM ('A')                  - un caractère 'A'
TERM ('TEXTE')              - un texte
TERM ("CHAINE")             - une liste de caractères
TERM (NB (123, CINQ))       - une structure NB (123, CINQ)
TERM ((UN, DEUX, TROIS))    - une liste de trois termes
TERM (145)                  - un nombre entier
TERM (ATOME)                - un atome
```

Le programme OBJET ci-dessous permet de lister (par les prédicats d'écriture à l'écran PUT, et de passage en début de ligne suivante LINE) tous les arguments de la base de faits TERM :

```
OBJET (*X0) : TERM (*X0) &
              LINE &
              PUT (*X0).
```

Une fois ces clauses saisies, on peut demander les résolutions suivantes, qui donnent des exemples d'utilisation des primitives de

test sur la nature des termes. La résolution donne une solution si le terme généré par OBJET est du type qui satisfait le prédicat qui suit OBJET dans la résolvante.

Les termes sont tous affichés par OBJET. Les solutions seront affichées derrière le nom de la variable de résolution *RESULTAT (en caractère gras, dans la résolution).

$?: OBJET (*RESULTAT) & VAR (*RESULTAT)
sélectionne parmi tous les arguments de TERM, ceux qui sont des variables.

```
*X0
*RESULTAT = *X0              - solution 1 : variable libre
'A'
'TEXTE'
"CHAINE'
NB (123, CINQ)
(UN, DEUX, TROIS)
145
ATOME
--SUCCES--(1)--
```

$?: OBJET (*RESULTAT) & CONSTANT (*RESULTAT)
sélectionne parmi tous les arguments de TERM, ceux qui sont des constantes.

```
*X0
'A'
*RESULTAT = 'A'              - solution 1
'TEXTE'
*RESULTAT = 'TEXTE'          - solution 2
"CHAINE"
NB (123, CINQ)
(UN, DEUX, TROIS)
145
*RESULTAT = 145              - solution 3
ATOME
*RESULTAT = ATOME            - solution 4
--SUCCES--(4)--
```

$?: OBJET (*RESULTAT) & INTEGER (*RESULTAT)
sélectionne les nombres entiers parmi les arguments de TERM.

```
*X0
'A'
'TEXTE'
"CHAINE"
NB (123, CINQ)
(UN, DEUX, TROIS)
145
*RESULTAT = 145              - solution 1
ATOME
--SUCCES--(1)--
```

```
$?: OBJET (*RESULTAT) & CHAR (*RESULTAT)
```

sélectionne les caractères de TERM.

```
*X0
'A'
*RESULTAT = 'A'                - solution 1
'TEXTE'
"CHAINE"
NB (123, CINQ)
(UN, DEUX, TROIS)
145
ATOME
--SUCCES--(1)--
```

```
$?: OBJET (*RESULTAT) & ATOM (*RESULTAT)
```

sélectionne les atomes de TERM.

```
*X0
'A'
'TEXTE'
"CHAINE"
NB (123, CINQ)
(UN, DEUX, TROIS)
145
ATOME
*RESULTAT = ATOME              - solution 1
--SUCCES--(1)--
```

```
$?: OBJET (*RESULTAT) & STRUCT(*RESULTAT)
```

sélectionne les structures de TERM.

```
*X0
'A'
'TEXTE'
"CHAINE"
*RESULTAT = "CHAINE"           - solution 1 : une chaîne de
NB (123, CINQ)                                caractères est
                                              une structure
*RESULTAT = NB (123, CINQ) - solution 2
(UN, DEUX, TROIS)
*RESULTAT =
(UN, DEUX, TROIS)              - solution 3 : une liste
145                                           d'atomes est
ATOME                                         une structure
--SUCCES--(3)--
```

b. Les prédicats de comparaison entre termes

EQ (<terme 1>, <terme 2>)

Ce prédicat teste l'égalité stricte de deux termes. Il réussit si ces deux termes sont égaux. On peut ainsi tester que deux variables ont été unifiées (qu'elles correspondent donc au même objet).

DIFF (<terme 1>, <terme 2>)

C'est l'opposé de EQ. Ce prédicat réussit si les deux termes sont différents.

Exemples :

Rappelons qu'il existe deux sens de "égalité" :

— l'égalité au sens *large* qui signifie "qui est unifiable". Nous la définirons au moyen du prédicat utilisateur EGAL (*A, *A), qui lance l'unification de ses arguments lorsqu'il est appelé,

— l'égalité *stricte*. Elle est testée par la primitive EQ. Le prédicat est vrai si les deux termes (simples ou composés) sont totalement identiques. C'est-à-dire que s'ils contiennent des variables, celles-ci doivent avoir été instanciées par des termes identiques, ou liées entre elles par unification.

```
$?: EQ (*A, *B)    - deux variables libres non liées sont
                     strictement différentes
--ECHEC--
$?: EGAL (*A, *B) & EQ (*A, *B)
*A                 = *1
*B                 = *1
--SUCCES--(1)--
```

Les variables libres *A et *B sont unifiées donc liées par le prédicat EGAL (ce que montre les solutions données pour *A et *B), donc elles sont égales au sens strict.

```
$?: EGAL (PRO (UN, 23), PRO (UN, 23))
--SUCCES--(1)--
```

Ces deux structures sont égales au sens large.

```
$?: EQ (PRO (UN, *X1), PRO (UN, *X2))
--ECHEC--
```

Il y a échec car les variables *X1 et *X2 sont libres et non liées entre elles.

```
$?: EGAL (*X1, *X2) & EQ (PRO (*X1), PRO (*X2))
*X1                = *1
*X2                = *1
--SUCCES--(1)--
```

Il y a égalité stricte entre les structures PRO (*X1) et PRO (*X2), car les variables libres *X1 et *X2 ont été auparavant liées entre elles.

```
$?: EGAL (PRO (*X1), PRO (*X2))
*X1                = *1
*X2                = *1
--SUCCES--(1)--
```

Il y a égalité au sens large.

```
$?: DIFF (PRO (*X1), PRO (*X2))
*X1                = *0
*X2                = *1
--SUCCES--(1)--
```

Les structures sont en effet différentes, car les variables qu'elles comportent sont libres, et non liées entre elles.

D. LA PRIMITIVE VALUE

La primitive VALUE correspond à l'affectation d'une variable globale classique. Elle permet de mémoriser un nombre ou un caractère dans un *atome*, qui ne doit pas être une tête de clause, et de le récupérer ultérieurement dans une variable. Cette affectation subsiste entre deux résolutions, sauf si l'atome est devenu une tête de clause.

Sa syntaxe est : VALUE (<source>, <destination>)

— *<source>*. La source correspond à la "valeur" qu'on veut utiliser et mémoriser. Elle peut être un nombre, un caractère, un atome ou une variable (qui doit automatiquement être instanciée par l'un des trois termes précédents, au moment de l'exécution).

Si <source> est un atome, c'est le contenu antérieur de cet atome qui sera utilisé ;

— *<destination>*. Destination désigne ce qui doit recevoir l'objet issu de <source> (qui est donc soit un nombre, soit un caractère). Elle doit donc être un atome ou une variable libre. Si c'est une variable instanciée, elle doit l'être par un atome.

Exemples :

Soit le programme en mémoire :

```
PERE (JEAN, LOUIS)
PERE (LOUIS, ANDRE)
PERE (ANDRE, PIERRE).
```

Examinons l'état du dictionnaire (par la commande DIC) :

```
$ DIC
1  PERE       3  C
2  JEAN       1  A
3  LOUIS      2  A
4  ANDRE      2  A
5  PIERRE     1  A
```

La résolution suivante va demander l'assignation du nombre entier 23 à l'atome JEAN :

```
$?: VALUE (23, JEAN)
--SUCCES--(1)--
```

Regardons ce que cela entraîne au niveau du contenu du dictionnaire :

```
$ DIC
1  PERE       3  C
2  JEAN       1  V
3  LOUIS      2  A
4  ANDRE      2  A
5  PIERRE     1  A
```

La lettre V indique que l'atome JEAN a "reçu" une valeur.

```
$?: VALUE (45,PERE)
ERR 4 EXECUTION
```

Ici, il y a erreur, car on ne peut affecter une valeur à un atome qui est une tête de clause. Donc aucune modification sur le dictionnaire.

```
$?: VALUE (*A, LOUIS)
ERR 4 : EXECUTION
```

L'assignation n'est pas possible non plus, car la variable *A n'est pas instanciée au moment de l'appel. D'où une erreur à l'exécution.

```
$?: PERE (*A, LOUIS) & VALUE (*A, *B)
*A                  = JEAN
*B                  = 23
--SUCCES--(1)--
```

La variable *A est d'abord instanciée par JEAN, puis la valeur mémorisée par JEAN est transférée dans *B et affichée.

```
$?: VALUE (JEAN, LOUIS) & VALUE (LOUIS, *A)
*A                  = 23
--SUCCES--(1)--
```

Cette résolution permet de transférer la "valeur" de JEAN dans l'atome LOUIS, puis de LOUIS dans la variable *A.

E. LES PRIMITIVES DE CONTROLE DU FONCTIONNEMENT DE L'INTERPRÊTEUR

On utilise deux prédicats de contrôle du fonctionnement de l'interpréteur.

Il existe d'autres prédicats PROLOG qui permettent de contrôler l'évaluation de l'interpréteur (réalisation de co-routines), mais ceux-ci ne sont nécessaires que pour une "programmation avancée" en PROLOG, et n'ont donc pas été implémentés ici (mais on remarquera pour mémoire que les prédicats définis ci-dessous sont les seuls prédicats de contrôle implémentés dans micro-PROLOG et dans le PROLOG d'EDINBOURG). Voyons maintenant ce qu'ils signifient.

NOT (<prédicat>)
Ce prédicat réussit si le prédicat énoncé en argument échoue. C'est-à-dire qu'il permet de vérifier qu'il n'existe pas de solution pour le but donné en argument. S'il existe une solution, il échoue. NOT ne peut pas servir à retourner des valeurs dans des variables.

Exemples :

```
$?: NOT (=(1,2))
--SUCCÈS--(1)--
```

Reprenons la base de termes utilisée dans le paragraphe c) :

```
TERM (*X0).
TERM ('A').
TERM ('TEXTE').
TERM ("CHAINE").
TERM (NB (123, CINQ)).
TERM ((UN, DEUX, TROIS)).
TERM (145).
TERM (ATOM).
$?: NOT (TERM ((UN, DEUX)))
--SUCCÈS--(1)--
```

Ce prédicat réussit car TERM ((UN, DEUX)) n'existe pas.

```
$?: NOT (TERM (NB (124, CINQ)))
--SUCCÈS--(1)--
```

Ce prédicat réussit car TERM (NB (124, CINQ)) n'existe pas dans le paquet TERM.

```
$?: NOT (TERM ('A'))
--ÉCHEC--
```

Ce prédicat échoue car TERM ('A') existe dans le paquet TERM.

CUT.

Ce prédicat est un *coupe-choix*. Il fige les choix faits depuis l'appel de la clause courante (dans laquelle se trouve le coupe-choix). C'est-à-dire que dès que l'interpréteur rencontre ce coupe-choix, il en déduit que les choix déjà effectués depuis l'appel de la clause courante ne seront pas remis en question lors du retour arrière.

Exemple :

Soit le programme contenu en mémoire :

```
HOMME (MARC).
HOMME (JULES).
HOMME (ANTOINE).
FEMME (MURIEL).
FEMME (CLAIRE).
```

Si nous voulons connaître tous les couples possibles (HOMME, FEMME), nous demanderons la résolution :

```
*X1 = MARC          - solution 1
*X2 = MURIEL
*X1 = MARC          - solution 2
*X2 = CLAIRE
*X1 = JULES         - solution 3
*X2 = MURIEL
*X1 = JULES         - solution 4
*X2 = CLAIRE
```

```
*X1 = ANTOINE          - solution 5
*X2 = MURIEL
*X1 = ANTOINE          - solution 6
*X2 = CLAIRE
--SUCCÈS--(6)--
```

Mais si nous voulons connaître maintenant tous les couples possibles dont MARC est l'homme, nous allons utiliser le prédicat CUT, pour signifier à l'interpréteur de ne pas remettre en question le choix HOMME (*X1), une fois qu'il l'aura satisfait avec le premier fait de la base HOMME (MARC).

```
$?: HOMME (*X1) & CUT & FEMME (*X2)
*X1 = MARC             - solution 1
*X2 = MURIEL
*X1 = MARC             - solution 2
*X2 = CLAIRE
--SUCCÈS--(2)--
```

F - LES PRIMITIVES DE MANIPULATION DE CLAUSES

Nous avons vu que l'ordre dans lequel on énonce nos clauses dans un sous-programme est important, et que l'interpréteur les traite dans cet ordre-là. Il se peut qu'à un moment ou à un autre, on ait besoin de travailler ou de se référer à une clause particulière. C'est le rôle des primitives de manipulation de clauses.

Ces primitives permettent de travailler au niveau d'une ou plusieurs clauses, donc de manipuler un programme "par programme" :

— certaines permettent de désigner (sélectionner) une clause unique, qui devient alors la *clause courante*.

— d'autres primitives travaillent implicitement au niveau de cette clause courante. Cela veut dire que celle-ci doit avoir été définie, sinon une erreur est détectée et la résolution s'arrête.

— la clause courante n'existe que durant une résolution, donc à chaque nouvelle résolution, il faut donner une nouvelle clause courante.

TOP

Ce prédicat, sans argument, sélectionne la première clause du premier paquet de clauses du programme. Cette clause devient la clause courante.

BOTTOM

Ce prédicat sélectionne la première clause du dernier paquet de clauses du programme. Cette clause devient la clause courante.

FORWARD

Cette primitive permet de sélectionner la clause qui suit la clause courante, dans un sous-programme. Cette clause devient la nouvelle clause courante, et le prédicat prend la valeur vrai.

Si la clause courante initiale était la dernière clause du paquet, le but échoue donc le prédicat prend la valeur faux, et la clause courante est inchangée.

BACKWARD

Cette primitive est identique à FORWARD, mais sélectionne la clause précédant la clause courante. Si la clause courante est la première du paquet, le prédicat échoue, et la clause courante est inchangée.

PACK (<nom-d'une-tête-de-clause>)

Ce prédicat permet de sélectionner, par programme, la première clause du paquet de clauses dont le nom (qui correspond au nom du prédicat de tête de toutes les clauses constituant le sous-programme) est donné en argument. Il ressemble donc au prédicat TOP, sauf qu'il se réfère cette fois-ci à un paquet spécifique et non au premier paquet du programme.

Si le nom de cette clause existe, elle devient la clause courante et le but réussit. Sinon le but échoue.

L'argument de PACK peut être une variable libre. Dans ce cas :

— si aucune clause courante n'est définie, PACK sélectionne la première clause du premier paquet de clauses, qui devient la clause courante (effet identique au prédicat TOP).

— sinon, PACK sélectionne la première clause du premier paquet suivant la clause courante.

— dans tous les cas, la variable libre est instanciée par le prédicat de tête de la clause sélectionnée.

ATTENTION

ON PEUT ÉCRIRE PACK (PÈRE(JEAN, MICHEL)), MAIS L'EFFET EST LE MÊME QUE SI L'ON AVANT ÉCRIT : PACK (PÈRE). EN EFFET, SEUL LE NOM PÈRE, QUI CORRESPOND A LA TÊTE D'UNE CLAUSE, EST CONSIDÉRÉ : AUCUNE UNIFICATION N'EST RÉALISÉE SUR LES ARGUMENTS DU PRÉDICAT PÈRE, ET LA CLAUSE SÉLECTIONNÉE SERA LA PREMIÈRE CLAUSE DU PAQUET PÈRE, MÊME SI SES ARGUMENTS NE SONT PAS JEAN ET MICHEL.

CLAUSE (<terme-tête>, <variable-libre-de-queue-de-liste>)

Cette primitive permet d'examiner et de manipuler, par programme, la clause courante en mémoire. Celle-ci doit être définie, sinon une erreur est signalée.

Il suffit que la tête de cette clause s'unifie avec <terme-tête>, auquel cas <variable-libre-de-queue-de-liste>, qui est libre au moment de l'appel, sera instanciée par la liste composée des différents prédicats du corps de la clause.

Utilisée en conjonction avec PACK, elle permet de faire un "filtrage" sur les clauses (en autorisant l'examen des clauses d'un paquet précis).

Exemple :

Si la clause courante est :

GD-PÈRE (*X0, *X1) : PÈRE (*X0, *X2) & PÈRE (*X2, *X1)

et que l'on demande la résolution de CLAUSE (*T, *Q), celle-ci réussit avec les solutions :

*T = GD-PÈRE (*X0, *X1)
*Q = (PÈRE (*X0, *X2), PÈRE (*X2, *X1)).

Les variables *T et *Q sont instanciées respectivement par la tête de la clause courante et par la liste des prédicats du corps de la clause. Ce prédicat permet donc de sélectionner et de traiter le corps d'une clause comme une liste, ce qui est fort intéressant. Si la clause courante est un fait de la base, l'argument <variable-libre-de-queue-de-liste> sera la liste vide ou NIL.

INSERT (<terme-tête>)
INSERT (<terme-tête>, <liste-queue>)

Nous avons vu que lorsque nous saisissons une clause, celle-ci était placée à la fin du sous-programme correspondant. Ici, le prédicat INSERT, sous ses deux formes, permet d'insérer une clause en tête du paquet de clause de même nom.

La clause courante doit donc normalement avoir le même nom. Si ce n'est pas le cas, la clause sera ajoutée au début du sous-programme correspondant.

La clause à insérer est donnée par son <terme-tête> (si c'est une variable, elle doit être instanciée au moment de l'appel), et éventuellement par la liste des prédicats du corps de la clause, dans <liste-queue>. Elle apparaîtra dans le programme sous la forme syntaxique normale d'une clause.

APPEND (<terme-queue>)
APPEND (<terme-tête>, <liste-queue>)

Ce prédicat est identique au précédent, mais il ajoute, cette fois-ci par programme, la clause désignée à la fin du sous-programme correspondant (de même nom).

ERASE

Ce prédicat a pour effet de détruire la clause courante, qui doit donc être définie. La clause suivante devient la clause courante. S'il n'y a pas de clause suivante, la clause courante est indéterminée.

Ce prédicat est à manier avec prudence ; en effet, la clause à détruire peut être un "ancêtre" de la clause en cours de résolution (donc être nécessaire, à un moment ou à un autre, pour la résolution). Or, aucun contrôle n'est effectué sur le rôle de la clause à détruire. Dans ce cas, l'interpréteur a un comportement aléatoire et imprévisible, qui peut entraîner la destruction du programme en mémoire...

PRINT

Ce prédicat, sans argument, permet d'affiche à l'écran, par programme, la clause courante (la tête et la queue).

Exemples :

Soit la base de données "familiale" et les règles suivantes :

```
PÈRE (JEAN, LUCIE).
PÈRE (JEAN, PIERRE).
PÈRE (ROBERT, JEANNE).
PÈRE (ROBERT, ALINE).
PÈRE (PIERRE, BOB).
PÈRE (PIERRE, OLIVIA).

MÈRE (ANNIE, LUCIE).
MÈRE (ANNIE, PIERRE).
MÈRE (CLAUDIE, JEANNE).
MÈRE (CLAUDIE, ALINE).
MÈRE (JEANNE, BOB).
MÈRE (JEANNE, OLIVIA).

PARENTS (*X0, *X1) : PÈRE (*X0, *X1).
PARENTS (*X0, *X1) : MÈRE (*X0, *X1).
```

Une fois cette base saisie, nous pouvons demander les résolutions suivantes :

```
$?: TOP & CLAUSE (*TÊTE, *QUEUE) & APPEND (*TÊTE).
*TÊTE = PÈRE (JEAN, LUCIE).
*QUEUE = NIL
--SUCCÈS--(1)--
```

Cette résolution a pour effet d'ajouter par programme, le prédicat PÈRE (JEAN, LUCIE) en queue du paquet PÈRE (7e clause), sans pour autant supprimer la clause initiale.

Nous pouvons constater cette action par la commande LIST (PÈRE). La clause rajoutée est visualisée en caractères gras.

```
$LIST (PÈRE)
= 1 =<3> PÈRE (JEAN. LUCIE).
= 2 =<3> PÈRE (JEAN, PIERRE).
= 3 =<3> PÈRE (ROBERT, JEANNE).
= 4 =<3> PÈRE (ROBERT, ALINE).
= 5 =<3> PÈRE (PIERRE, BOB).
= 6 =<3> PÈRE (PIERRE, OLIVIA).
= 7 =<3> PÈRE (JEAN, LUCIE).
```

Soit la clause LAST (*X0) qui permet de sélectionner et d'imprimer la dernière clause d'un paquet (dont le nom est passé en paramètre). Elle utilise les clauses AVANCE qui permettent de progresser dans le sous-programme, jusqu'à ce que la dernière clause soit atteinte.

```
LAST (*X0) : PACK (*X0) &
             AVANCE.
AVANCE : FORWARD &
         CUT &
         AVANCE.
AVANCE : PRINT &
         CUT.
```

```
$?: LAST (PÈRE)
PÈRE (JEAN, LUCIE)
--SUCCÈS--(1)--
```

```
$?: LAST (PÈRE) & ERASE
PÈRE (JEAN, LUCIE),
--SUCCÈS--(1)--
```

- sélectionne la dernière clause du paquet PÈRE et la détruit.

```
$LIST (PÈRE)
= 1 =<3> PÈRE (JEAN, LUCIE).
= 2 =<3> PÈRE (JEAN, PIERRE).
= 3 =<3> PÈRE (ROBERT, JEANNE).
= 4 =<3> PÈRE (ROBERT, ALINE).
= 5 =<3> PÈRE (PIERRE, BOB).
= 6 =<3> PÈRE (PIERRE, OLIVIA).
```

La demande de listage du paquet PÈRE montre que la dernière clause (la septième) a été détruite. Cet effet correspond à celui de la commande DEL (PÈRE,7).

Pour sélectionner la deuxième clause du paquet PARENTS, on demandera la résolution suivante :

```
$?: PACK (PARENTS) & FORWARD & CLAUSE (*A, *B)
*A = PARENTS (*X0, *X1)
*B = (MÈRE (*X0. *X1))
--SUCCÈS--(1)--
```

*A est instanciée par la tête de la clause et *B par la liste des prédicats de queue (il n'y a qu'un prédicat ici).

```
$?: TOP & CLAUSE (*TÊTE, *QUEUE)
*TÊTE = PÈRE (JEAN, LUCIE)
*QUEUE = NIL
--SUCCÈS--(1)--
$?: TOP & CLAUSE (PÈRE (JEAN, *F), *QUEUE)
*F = LUCIE
*QUEUE = NIL
--SUCCÈS--(1)--
```

Ici, on utilise le procédé de "filtrage" d'une clause : la tête de la clause est unifiée avec le premier argument de CLAUSE.

G - LES PRIMITIVES D'ENTRÉE/SORTIE DE TEXTE

On peut être amené à devoir "inter-agir" avec l'utilisateur, en cours d'exécution d'un programme : il se peut qu'on ait à lui afficher des informations ou, au contraire, qu'on doive en recevoir de sa part, par l'intermédiaire du clavier de saisie.

Il existe pour cela un ensemble de prédicats d'affichage à l'écran et de saisie au clavier.

a. Primitives d'affichage

PRINTON

A priori, toutes les sorties se font vers l'écran. Le prédicat PRINTON, (sans argument), permet de sortir "simultanément" sur écran et sur imprimante. Ce prédicat peut aussi être considéré comme une commande. Il réussit, bien sûr, à partir du moment où l'imprimante est bien connectée.

Exemple :

```
$?: PRINTON
$LIST
```

permet de sortir simultanément sur l'écran et sur l'imprimante un listing du programme en mémoire.

PRINTOFF

Ce prédicat supprime l'effet de PRINTON. Il est aussi considéré comme une commande.

PUT (<terme>, <terme>, ..., <terme>)

Le prédicat PUT permet de programmer l'affichage à l'écran de toute terme PROLOG, y compris un prédicat ou un terme composé. Nous avons vu son utilisation avec les termes "textes" (voir § 3.1.1.4, p. 93). On peut lui donner jusqu'à 15 paramètres.

LINE

Le prédicat LINE provoque un saut de ligne sur l'écran. Il permet donc, lors d'une résolution, de "formater" par ligne l'affichage de termes.

b. Primitives de saisie clavier

GET (<terme>)

Le prédicat GET permet de lire tout objet PROLOG (terme quelconque ou même prédicat) entré au clavier par l'utilisateur, et même de créer interactivement des clauses par programme.

A l'exécution, le GET engendre un retour à la ligne suivante sur l'écran, et affiche ">" pour indiquer qu'il est en attente d'entrée clavier. L'utilisateur a 38 caractères pour faire sa saisie, en s'aidant des touches d'édition du mode saisie de commande (voir § 3.1.3.1).

Une fois la saisie terminée, elle est validée par la touche **ENTRÉE**. Le texte est alors passé à l'analyseur syntaxique.

Si une erreur est détectée (c'est-à-dire que le texte saisi n'est pas compatible avec l'argument de GET), un message est affiché à l'écran, et la résolution est terminée.

Exemple :

GET (PÈRE (*A, *B)).

Si l'opérateur tape JEAN, il y a échec à l'unification. On a donc tout intérêt à envoyer un message à l'utilisateur (par le prédicat PUT), pour lui spécifier le type de saisie attendu.

Si aucune erreur n'est détectée, alors l'interpréteur essaye d'unifier le terme saisi et le terme "argument" de GET. Si l'unification réussit, les variables libres contenues éventuellement dans <terme> seront instanciées.

Exemple :

Si on reprend l'exemple précédent mais que, cette fois-ci, l'opérateur tape PÈRE (JEAN, PAUL), l'unification réussit : *A est instancié par JEAN, *B est instancié par PAUL.

A chaque fois qu'on lit une structure ou un atome par la primitive GET, il y a stockage en mémoire du texte lu, pour la durée de la résolution. Il peut arriver que la mémoire sature, auquel cas la résolution est abandonnée. En fin de résolution, tous les éléments stockés sont détruits.

Exemples :

Les textes rentrés au clavier par l'utilisateur sont écrits en italique. Rappelons également que le signe ">" indique que GET attend une saisie au clavier.

```
$?: GET (*A)
> 123
*A = 123
--SUCCÈS--(1)--
```

```
$?: GET (PÈRE (*P, *FILS))
>JEAN
--ÉCHEC--
```

Ici, il y a échec car JEAN ne s'unifie pas avec l'argument de GET, qui est PÈRE (*P, *FILS).

```
$?: GET (PÈRE (*P, *FILS))
> PÈRE (JEAN, MICHEL)
*P = JEAN
*FILS = MICHEL
--SUCCÈS--(1)--
```

Ici, la réponse PÈRE (JEAN, MICHEL) s'unifie avec l'argument du GET. *P est instancié par JEAN, *FILS est instancié par MICHEL.

Le prédicat GET permet, on l'a dit, de créer interactivement des clauses par programme.

Soit le programme AJCL (AJouter des CLauses) qui permet de lire un fait (c'est-à-dire un prédicat sans queue) et l'ajoute aux autres clauses en mémoire (à la fin du paquet de clauses correspondant).

```
AJCL : LINE &
       PUT (ENTREZ UN FAIT :') &
       GET (*X0) &
       APPEND (*X0).
```

```
$?: AJCL
ENTREZ UN FAIT :
> PÈRE (JEAN, MICHEL)
--SUCCÈS--(1)--
```

Le prédicat PÈRE (JEAN, MICHEL) est tapé par l'utilisateur puis ajouté en mémoire, ce que nous montre l'action de la commande LIST :

```
$LIST (PÈRE)
= 1 =<3> PÈRE (JEAN, MICHEL).
```

Si on demande une autre résolution :

```
$?: AJCL
ENTREZ UN FAIT :
> PÈRE (MICHEL, ALBERT)
--SUCCÈS--(1)--
```

Le prédicat PÈRE (MICHEL, ALBERT) est rajouté en mémoire, à la fin du paquet de clauses correspondant :

```
$LIST (PÈRE)

= 1 =<3> PÈRE (JEAN, MICHEL).
= 2 =<3> PÈRE (MICHEL, ALBERT).
```

GETC (<suite-de-caractères>)

Le prédicat GETC permet de lire caractère par caractère ce qui vient d'être entré au clavier par l'utilisateur (correspondant à la primitive READ en langage PASCAL).

A l'exécution, GETC entraîne un retour à la ligne et l'affichage du symbole "=", pour indiquer qu'il est en attente d'entrée clavier. L'utilisateur entre simplement une suite de caractères (sans utiliser les termes de type texte ou chaîne) et valide sa saisie par **ENTRÉE**. Ceci permet de tester des réponses par programme (exemple : si l'on attend de l'utilisateur la réponse OUI ou NON).

H - LES PRIMITIVES D'ÉDITION A LA RÉSOLUTION

Lorsqu'on demande une résolution, on peut souhaiter pouvoir suivre le déroulement de celle-ci à l'écran. C'est le rôle des différentes primitives d'édition à la résolution.

PAUSE

Cette primitive permet d'arrêter momentanément l'affichage des solutions lors d'une résolution, en particulier lorsque cet affichage est formaté par programme (puisque lors de l'affichage des solutions par l'interpréteur, cet effet s'obtient par appui sur n'importe quelle touche du clavier. Si l'on désir pouvoir le faire, il suffit de rajouter la primitive PAUSE à la fin de la demande de résolution. A chaque fois que ce prédicat est "évalué" (après tout affichage d'une nouvelle solution), il effectue une scrutation du clavier. Si une touche est enfoncé (c'est-à-dire que vous voulez arrêter momentanément l'affichage des solutions), il suspend la résolution. Pour la reprendre, il attend que vous appuyiez une nouvelle fois sur une touches quelconque.

La touche **RAZ** provoque l'abondon de la résolution en cours.

DOFF

Ce prédicat supprime l'affichage des solutions trouvées lors d'une résolution. Il peut aussi être utilisé comme une commande. L'effet persiste sur toutes les résolutions ultérieures, tant que l'on ne l'a pas supprimé par le prédicat DON.

Ainsi, l'utilisateur pourra formater la présentation des solutions, à sa convenance.

DON

Ce prédicat rétablit l'affichage systématique des solutions lors d'une résolution. Il peut aussi être considéré comme une commande.

TRON

Ce prédicat permet de suivre à l'écran une résolution, en affichant les différentes évaluations successives de la clause à évaluer. On dit que ce prédicat arme le mode trace.

Le fait que ce soit un prédicat permet de déclencher la trace lors de l'évaluation d'un but précis. Mais il peut aussi être considéré comme une commande. L'effet persiste sur toutes les résolutions ultérieures, tant qu'il n'a pas été annulé par TROFF.

- Pour arrêter le défilement de l'écran, il suffit d'appuyer sur une touche quelconque du clavier.
- Pour reprendre l'exécution du programme et l'affichage de la trace, on réappuie sur une touche.
- L'appuie sur la touche RAZ fait sortir du mode trace et la résolution continue en mode normal.
- Un nouvel appui sur RAZ provoque l'abandon de la résolution.

Remarque :

Ce mode de résolution ralentit considérablement l'exécution du programme.

TROFF

Ce prédicat annule l'effet de TRON. Il peut être utilisé aussi comme une commande.

Exemple :

Reprenons un programme déjà utilisé et suivons sa résolution en mode trace :

```
HOMME (MARC).
HOMME (JULES).
HOMME (ANTOINE).

FEMME (MURIEL).
FEMME (CLAIRE).
```

Si nous actionnons la commande LIST :

```
$LIST
= 1 =<2> HOMME (MARC).
= 2 =<2> HOMME (JULES).
= 3 =<2> HOMME (ANTOINE).

= 1 =<2> FEMME (MURIEL).
= 2 =<2> FEMME (CLAIRE).
```

On rappelle, cela nous servira pour comprendre la résolution en mode trace, que le nombre placé entre les deux signes "=", devant les clauses, indique le numéro d'ordre de la clause dans le paquet correspondant.

Armons le mode trace :

```
$?: TRON
```

Pour suivre et comprendre la résolution en mode trace, nous indiquons d'abord les conventions qui ont été prises :

— les solutions sont affichées normalement par la résolution.

— le reste est affiché par le mode trace (en italique, ici) :
- le signe "-->" indique le prédicat courant (qui est donc en train d'être évalué) : c'est le *point d'appel.*
- les prédicats précédés par un nombre entre deux signes "=", indiquent les têtes de prédicat qui s'unifient avec l'appel courant (le nombre entre deux "=" rappelle le rang de cette clause dans le paquet).
- à chaque fois que l'interpréteur essaye de resatisfaire un but, il y a retour arrière ou backtrack, visualisé par le signe "<--".
- l'interpréteur renomme les variables dans le mode trace, tout en assurant toujours leur cohérence.

Suivons maintenant la résolution :

```
$?: HOMME (*X1) & FEMME (*X2)
--> HOMME (*X0)
= 1 = HOMME (MARC)
--> FEMME (*X1)
= 1 = FEMME (MURIEL)
*X1 = MARC                  - solution 1
*X2 = MURIEL
<-- FEMME (*X1)             - backtrack sur FEMME
= 2 = FEMME (CLAIRE)
*X1 = MARC                  - solution 2
*X2 = CLAIRE
<-- HOMME (*X0)             - backtrack sur HOMME
= 2 = HOMME (JULES)
--> FEMME (*X1)             - nouvel appel à FEMME
= 1 = FEMME (MURIEL)
*X1 = JULES                 - solution 3
*X2 = MURIEL
<-- FEMME (*X1)             - backtrack sur FEMME
= 2 = FEMME (CLAIRE)
*X1 × JULES                 - solution 4
*X2 = CLAIRE
<-- HOMME (*X0)             - backtrack sur HOMME
= 3 = HOMME (ANTOINE)
--> FEMME (*X1)             - nouvel appel à FEMME
= 1 = FEMME (MURIEL)
*X1 = ANTOINE               - solution 5
*X2 = MURIEL
<-- FEMME (*X1)             - backtrack sur FEMME
= 2 = FEMME (CLAIRE)
*X1 = ANTOINE               - solution 6
*X2 = CLAIRE
```

```
--SUCCÈS--(6)--          - toutes les combinaisons de
                           HOMME et FEMME ont été
                           affichées : la résolution s'ar-
                           rête.
```

Si nous reprenons le même exemple, avec cette fois-ci, l'utilisation du prédicat CUT, qui annule toutes les alternatives sur HOMME (on n'a pas besoin de réarmer le mode trace, qui est toujours actif ici, puisqu'il a été appelé sous forme de commande) :

```
$?: HOMME (*X1) & CUT & FEMME (*X2)
--> HOMME (*X0)          - appelle une fois HOMME
= 1 = HOMME (MARC)
--> CUT
--> FEMME (*X1)
= 1 = FEMME (MURIEL)
*X1 = MARC               - solution 1
*X2 = MURIEL
<-- FEMME (*X1)          - backtrack sur FEMME
= 2 = FEMME (CLAIRE)
*X1 = MARC               - solution 2
*X2 = CLAIRE
--SUCCÈS--(2)--
```

*X1 ne sera instancié que par un seul objet MARC, car le prédicat CUT interdit le retour arrière sur HOMME. *X2 prendra les deux valeurs successives : MURIEL, CLAIRE.

EXEMPLE RÉCAPITULATIF SUR L'UTILISATION DES PRIMITIVES

Le programme ci-dessous, composé des clauses LPROG et LCL, permet de lister à l'écran (par exécution d'un programme) toutes les clauses contenues en mémoire. Le listing obtenu par LPROG n'est pas formaté comme dans le cas d'une commande LIST. Il montre des exemples d'utilisation des commandes LIST (<nom-de-clause>) et des prédicats prédéfinis TOP, LINE, PACK (<nom-d'une-tête-de-clause>) et CUT.

```
$LIST(LPROG)
= 1 =<3> LPROG:
         TOP &
         LCL.
$LIST(LCL)
= 1 =<4> LCL:
         PRINT &
         FORWARD &
         LCL.
= 2 =<5> LCL:
         PACK(*X0) &
         LINE &
         LCL.
= 3 =<2> LCL:
         CUT.
```

On voit que :

— la clause LPROG permet de sélectionner la première clause du premier paquet de clauses du programme, et d'appeler LCL.

— la clause LCL possède trois alternatives, dont deux récursives.

On suppose que le programme de la base de données précédente était déjà en mémoire :

```
= 1 =<2> HOMME (MARC).
= 2 =<2> HOMME (JULES).
= 3 =<2> HOMME (ANTOINE).

= 1 =<2> FEMME (MURIEL).
= 2 =<2> FEMME (CLAIRE).

= 1 =<7> COUPLE (*X0, *X1) :
         HOMME (*X0) &
         FEMME (*X1).
```

Demandons la résolution :

```
$?: LPROG
HOMME (MARC).
HOMME (JULES).
HOMME (ANTOINE).

FEMME (MURIEL).
FEMME (CLAIRE).

LPROG : TOP & LCL.

COUPLE (*X0, *X1) : HOMME (*X0) & FEMME (*X1).

LCL : PRINT & FORWARD & LCL.
LCL : PACK (*X0) & LINE & LCL.
LCL : CUT.
--SUCCÈS--(10)--
```

3.2 - RÉSOLVANTE DE PROGRAMME : LES ÉTAPES DE RÉSOLUTION

Un programme PROLOG consiste, on l'a vu, à saisir, donc à énoncer des clauses (faits et règles) de façon formalisée. Ces clauses regroupent notre représentation des "connaissances" (les faits) et un mécanisme général de résolution de problèmes (les règles).

Demander une résolution correspond à exécuter un programme PROLOG, en donnant à l'interpréteur un but (une question) à résoudre. Nous avons vu que la syntaxe d'une demande de résolution est :

```
?: (nom-but)
ou ?(nom) : (nom-but)
```

Le "nom-but" est soit un prédicat, soit une conjonction de prédicats définis par l'utilisateur et/ou prédéfinis.

La deuxième formulation a la syntaxe d'une clause, mais la première lettre du nom de tête est un "?". Cette clause est ajoutée momentanément au programme durant l'exécution, et détruite à la fin.

● Pour comprendre ce qui va suivre, assurez-vous d'avoir bien assimilé les principes vus précédemment :

— l'ordre dans lequel on énonce les clauses d'un même sous-programme est important et détermine le cheminement de l'interpréteur lors d'une résolution (voir § 3.3.1.2.3). En général, les faits seront énoncés avant les règles, et on gérera avec prudence l'ordre des clauses comprenant des appels récursifs (on prendra soin de saisir les conditions d'arrêt avant les clauses récursives, afin d'éviter que le programme ne boucle).

— les clauses de même nom, reconnues par l'interpréteur, sont mémorisées dans le sous-programme correspondant, selon une représentation bien particulière utilisant des piles, des pointeurs et des termes mémoire.

● Jusqu'à maintenant, nous avons vu plus particulièrement l'aspect "statique" des programmes, c'est-à-dire la façon de les énoncer. Demander une résolution, c'est **activer** un programme, c'est-à-dire le "dynamiser". L'interpréteur va alors utiliser le programme pour essayer de satisfaire le but à atteindre. Satisfaire le but, ce n'est pas trouver *directement* une solution, mais essayer par *tous* les moyens, en utilisant toutes les clauses pertinentes, de trouver *toutes les solutions possibles*. Évidemment cette recherche ne se fait pas de façon anarchique, mais suit certains principes que nous allons rappeler.

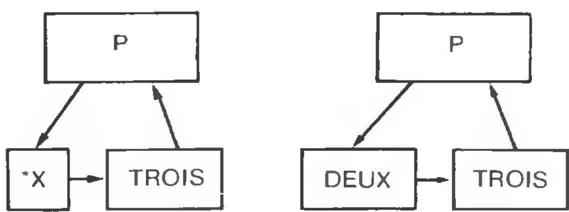
La résolution est basée sur la correspondance des formes selon le processus d'unification.

L'interpréteur est incapable d'évaluer directement la question, le but que vous lui proposez de résoudre ; et d'en déduire la ou les solutions. Il travaille en fait sur les principes de similitude de syntaxe et de forme. Il vérifie d'abord que la syntaxe du but est valide dans le langage PROLOG. Puis, pour chaque but qu'il aura à traiter, il essayera de trouver un fait ou une tête de règle qui lui corresponde, pour cela, il compare les constantes et instancie les variables libres, si cela est possible.

L'unification réussit si les constantes de même rang sont identiques et si on a pu instancier les variables.

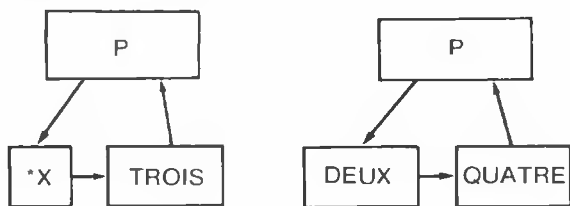
Exemples :

P (*X, TROIS) et P (DEUX, TROIS)



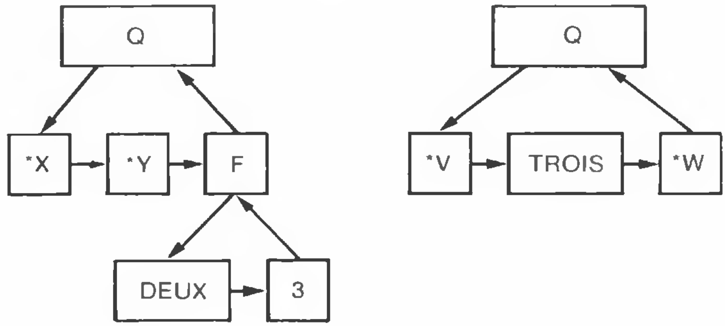
En parcourant parallèlement les deux arbres de termes, l'unification réussit, après instanciation de *X par DEUX.

P (*X, TROIS) et P (DEUX, QUATRE)



L'unification échoue ici : la variable libre *X a été instanciée par DEUX, mais la comparaison TROIS, QUATRE montre que ces deux constantes sont différentes.

Q (*X, *Y, F (DEUX, 3)) et Q (*V, TROIS, *W)



Ici l'unification réussit :

— les variables libres *X et *V sont unifiées. Dès que l'une d'entre elles sera instanciée, l'autre le sera automatiquement par le même objet.

— la variable libre *Y est instanciée par TROIS.

— la variable libre *W est instanciée par la structure F (DEUX, 3).

De façon générale, le processus d'unification de deux clauses s'effectue de la manière suivante :

— on part du sommet de chaque clause.

— on compare les termes du sommet, qui sont des atomes :
- s'ils sont identiques, on va essayer d'unifier les termes suivants de chaque arbre, en suivant les flèches (de haut en bas puis de gauche à droite).
- sinon l'unification est un échec.

— pour chaque terme à comparer :
- si ces deux termes sont des constantes, on regarde si elles sont identiques (succès) ou non (échec).
- si l'un des termes est une variable libre, on l'instancie par l'autre terme.
- si les deux termes sont des variables libres, on les lie. C'est-à-dire que, dès que l'une de ces variables sera instanciée, l'autre le sera automatiquement par le même objet.

• LES SOUS-PROGRAMMES OU APPEL DE PROCÉDURES

Bien sûr, l'interpréteur ne vas pas "perdre son temps" à essayer toutes les unifications possibles du programme lors d'une résolution. Quel que soit le but énoncé, son nom de prédicat se réfère à une ou plusieurs clauses de même nom, énoncées dans le programme. Ces clauses de même nom, on l'a vu, sont mémorisées et groupées, après leur saisie et l'analyse syntaxique, dans ce que l'on appelle un sous-programme.

Ainsi, toute tentative d'unification commence par l'appel de la procédure ou sous-programme correspondant. Le processus d'unification n'est donc engagé que sur le sous-programme correspondant. Nous verrons que ce principe est valable tout au long de la résolution.

• LE PRINCIPE DU BACKTRACK OU RETOUR ARRIÈRE

Demander une résolution en PROLOG ne se réduit pas, pour l'interpréteur, à s'arrêter à la première solution trouvée, mais à envisager et à rechercher toutes les solutions possibles. Dans un premier temps, il cherche une première solution : on dit qu'il essaye de *satisfaire* le but. Puis il reprend sa recherche, en vue de nouvelles solutions. On dit qu'il essaye de *resatisfaire* le but. Rappelons la définition de ces deux termes :

— **Satisfaire un but.** Quand on veut satisfaire un but on essaye de trouver le paquet de clauses correspondant. Deux cas peuvent se présenter :

a. On trouve un fait ou une tête de règle qui correspond. On marque l'endroit de la base de données où a été trouvé la correspondance et on instancie toute les variables par le processus d'unification.

   Si le but correspond à un fait, il réussit.

   Si le but correspond à une tête de règle, on doit — par définition — essayer de satisfaire tous les buts du corps de la règle (prédicats reliés par des "&", en partant du premier. Si ce but réussit, on essaye alors de satisfaire celui directement à sa droite, etc. Le but initial réussit si tous les buts du corps de la règle réussissent simultanément.

b. On ne trouve aucun fait ou tête de règle correspondant : le but *échoue*. On essaye alors de resatisfaire le but situé directement à la gauche de celui-ci (en remontant au dernier choix effectué). Si le but qui a échoué est le premier (ou le seul), alors on ne peut plus satisfaire le but.

— **Resatisfaire un but.** Resatisfaire un but, c'est chercher à utiliser d'autres alternatives aboutissant à une solution, en "oubliant" pour cela les solutions antérieures : pour cela, on utilise le processus du **retour arrière.**

En règle générale, on ne peut resatisfaire un but que si celui-ci possède plusieurs alternatives.

Pour resatisfaire une règle, on essaye de resatisfaire chacun des buts du corps de la règle l'un après l'autre, en remontant cette fois-ci de droite à gauche.

Si on ne peut resatisfaire aucun des buts, on essaye alors de trouver une autre clause, une autre alternative, pour le but initial.

Essayer de resatisfaire un but commence d'abord par l'action de lui faire "oublier" la solution antérieure, c'est-à-dire de libérer toutes les variables qui avaient été instanciées auparavant pour ce but. Puis la recherche s'effectue sur la base de données, en partant cette fois-ci de la position du marqueur du but (puisque par définition, tout ce qui se trouve au-dessus à été testé).

Comme dans le cas de la satisfaction d'un but, ce nouveau but peut réussir ou échouer. On se retrouve alors dans les cas a) ou b) décrits ci-dessus.

Demander une résolution consiste donc, pour l'interpréteur à essayer de satisfaire et de resatisfaire successivement une série de buts en vue de donner toutes les solutions possibles. Ces solutions sont énoncées dans un ordre qui dépend de l'ordre de saisie des clauses dans un sous-programme, comme nous l'avons déjà dit précédemment.

Pour vous en assurer, prenons le programme simple suivant :

```
$LIST
= 1 =<2> ENTRÉE (RADIS).
= 2 =<2> ENTRÉE (PATE).

= 1 =<2> PLAT (VEAU).
= 2 =<2> PLAT (TRUITE).

= 1 =<2> DESSERT (FROMAGE).
= 2 =<2> DESSERT (FRUIT).

= 1 =<10> REPAS (*X0, *X1, *X2) :
          ENTRÉE (*X0) &
          PLAT (*X1) &
          DESSERT (*X2).
```

La règle REPAS permet de définir la composition d'un repas comme proposant successivement une entrée, un plat et un dessert.

Si nous voulons savoir quel menu nous pouvons composer, en proposant des radis en entrée, nous demanderons la résolution :

```
$?: REPAS (RADIS, *P, *D)
*P = VEAU                -> solution 1
*D = FROMAGE
*P = VEAU                -> solution 2
*D = FRUIT
*P = TRUITE              -> solution 3
*D = FROMAGE
*P = TRUITE              -> solution 4
*D = FRUIT
--SUCCÉS--(4)--
```

Pourquoi les solutions sont-elles énoncées dans cet ordre ? Parce que cet ordre correspond à celui utilisé dans la démarche de résolution :

— PROLOG doit essayer ici de satisfaire l'unique règle REPAS, qu'il trouve dans le programme. Pour cela, il va essayer de satisfaire les buts proposés dans le corps de la règle.

— le premier but à satisfaire est ENTRÉE (RADIS), c'est-à-dire ici que l'interpréteur va vérifier que ce fait existe dans la base. PROLOG satisfait ensuite le deuxième but PLAT (*P), en trouvant VEAU (troisième fait de la base). Le but suivant à satisfaire est DESSERT (*D) : PROLOG trouve FROMAGE. Vu que tous les buts ont réussi, le but initial réussit et PROLOG a trouvé une première solution (VEAU, FROMAGE).

— PROLOG essaye de resatisfaire le dernier but traité DESSERT (*D), afin d'aller jusqu'au bout de sa recherche. Il trouve un autre dessert FRUIT, donc une deuxième solution au problème : (VEAU, FRUIT). Cette fois-ci il a "épuisé" les faits DESSERT, et va remonter au but précédent PLAT, pour essayer de le resatisfaire.

— resatisfaire le but PLAT (*P), c'est d'abord "oublier" les résultats antérieurs portant sur PLAT et DESSERT. PROLOG se rappelle que le plat VEAU a déjà été testé, et essaye donc d'en trouver un autre dans la base : il trouve TRUITE. Il va donc tenter de satisfaire le but suivant DESSERT (*D). Cette fois-ci, il a complètement oublié les valeurs antérieures de *D (qu'il avait épuisées) et sa recherche repart du début. Il trouve d'abord FROMAGE, donc une troisième solution (TRUITE, FROMAGE), puis en essayant de resatisfaire ce but, il trouve le fait suivant FRUIT : une quatrième solution est découverte (TRUITE, FRUIT).

— PROLOG essaye une fois de plus de resatisfaire le but DESSERT (*D), mais il n'y a plus de faits correspondants dans la base. Donc il "remonte" au dernier choix effectué et essaye de resatisfaire PLAT (*P) : la situation étant la même, il "remonte" encore au premier but ENTRÉE (RADIS), qui ne peut lui non plus être resatisfait.

Toutes les solutions ont donc été envisagées, puisqu'il n'y a plus de but à satisfaire (ENTRÉE (RADIS) étant le premier), et la résolution s'arrête.

Cet exemple est très simple, car il contient seulement la définition de quelques faits et d'une règle pour accéder à ces faits. En pratique, les programmes sont souvent plus complexes, laissant place à plus d'alternatives et utilisant plus de variables. Mais la résolution fonctionne toujours sur les mêmes principes de base.

Prenons le programme suivant un peu plus compliqué (nous avons affecté à chaque clause un numéro entre parenthèses, que nous allons utiliser pour notre démonstration) :

```
$ 4 sT
= 1 =<3> PÈRE (JEAN, MICHEL).           (1)
= 2 =<3> PÈRE (MICHEL, MARC).           (2)

= 1 =<6> ANCÈTRE (*X0, *X1) :           (3)
         PÈRE (*X0, *X1).
= 2 =<9> ANCÊTRE (*X0, *X1) :           (4)
         PÈRE (*X0, *X2) &
         ANCÊTRE (*X2, *X1).
```

Nous avons volontairement réduit la base de données à deux faits, pour clarifier l'explication (numérotés (1) et (2)).

Le programme a besoin de deux règles, (numérotées (3) et (4)), pour définir la relation "être ancêtre paternel de", qui s'énonce de la plus simple à la plus compliquée, et dans un ordre défini par la donnée d'une condition d'arrêt sur les clauses récursives :

— la plus simple d'abord. On est ancêtre paternel de quelqu'un si on est son ancêtre direct, donc ici son père (numéro (3)). Cette règle définit, de plus, la condition d'arrêt et évite donc que le programme "boucle" à l'exécution (voir § 3.1.2.3).

— mais on peut être ancêtre paternel de quelqu'un de façon plus lointaine. C'est ce qu'énonce la deuxième clause (numéro (4)). Un individu est l'ancêtre paternel d'un autre, si on réussit à établir un "chaînage père-fils" entre le premier individu et le deuxième. Cette règle utilise le principe de récursivité, puisqu'elle se rappelle elle-même.

Supposons que nous demandions la résolution suivante :

$?: ANCÊTRE (*X0, MARC)

c'est-à-dire que nous demandions de nous donner les objets (ici les individus) qui sont les ancêtres paternels de MARC.

Les solutions annoncées, qui vous paraissent sûrement évidentes sont :

```
*X0 = MICHEL
*X0 = JEAN
--SUCCÈS--(2)--
```

Les étapes de résolution

Voyons maintenant comment l'interpréteur en est arrivé là. Le principe de résolution est toujours le même (que vous pouvez suivre à l'écran, avant de demander la résolution, en actionnant la commande TRON) :

```
$TRON
$?: ANCÊTRE (*X0, MARC)
--> ANCÊTRE (*X0, MARC)
= 1 = ANCÊTRE (*X0, MARC)
--> PÈRE (*X0, MARC)
= 2 = PÈRE (MICHEL, MARC)
*X0 = MICHEL
<-- ANCÊTRE (*X0, MARC)
= 2 = ANCÊTRE (*X0, MARC)
--> PÈRE (*X0, *X2)
= 1 = PÈRE (JEAN, MICHEL)
--> ANCÊTRE (MICHEL, MARC)
= 1 = ANCÊTRE (MICHEL, MARC)
--> PÈRE (MICHEL, MARC)
= 2 = PÈRE (MICHEL, MARC)
*X0 = JEAN
<-- ANCÊTRE (MICHEL, MARC)
= 2 = ANCÊTRE (MICHEL, MARC)
--> PÈRE (MICHEL, *X2)
= 2 = PÈRE (MICHEL, MARC)
--> ANCÊTRE (MARC, MARC)
= 1 = ANCÊTRE (MARC, MARC)
--> PÈRE (MARC, MARC)
>-- ANCÊTRE (MARC, MARC)
= 2 = ANCÊTRE (MARC, MARC)
--> PÈRE (MARC, *X2)
<-- PÈRE (*X0, *X2)
```

```
= 2 = PÈRE (MICHEL, MARC)
--> ANCÊTRE (MARC, MARC)
= 1 = ANCÊTRE (MARC, MARC)
--> PÈRE (MARC, MARC)
<-- ANCÊTRE (MARC, MARC)
= 2 = ANCÊTRE (MARC, MARC)
--> PÈRE (MARC, *X2)
--SUCCÈS--(2)--
```

Au lieu d'expliquer cette résolution par des mots, nous avons choisi de représenter celle-ci par un schéma, simulant la démarche de l'interpréteur lors d'une résolution, en utilisant les conventions suivantes :

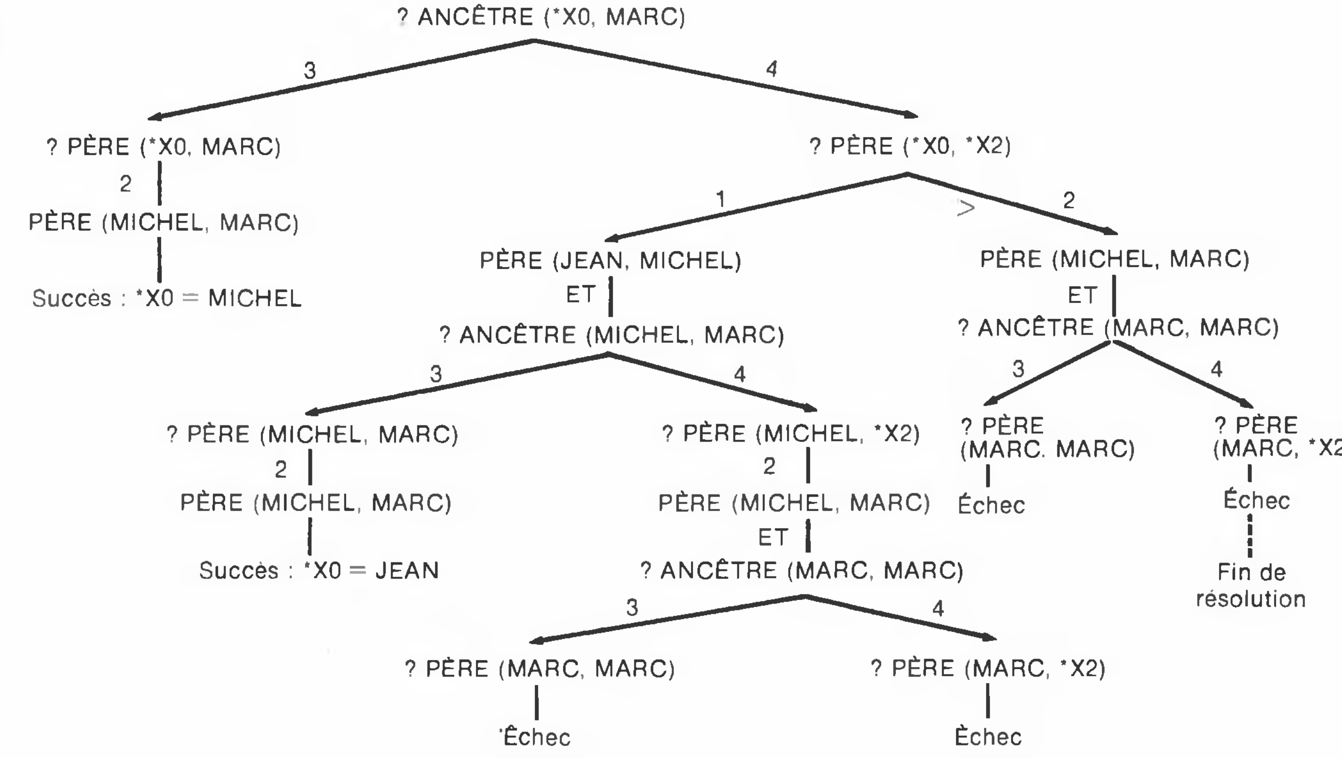
— les prédicats précédés du "?" sont des buts à satisfaire. Si le "?" n'apparaît pas, l'énoncé correspond à une affirmation, donc à un fait de la base (c'est-à-dire que l'on a pu satisfaire le but).

— les flèches latérales se rapportent aux différents alternatives possibles d'une même règle (entraînant des retours arrière). Elles sont affectées d'un numéro correspondant à celui de la clause testée.

Rappelez-vous que l'on teste une règle en **profondeur**, c'est-à-dire qu'on ne l'abandonne pas tant que l'on n'a pas abouti à "succès" ou "échec". C'est seulement à ce moment que l'on teste les autres alternatives.

Donc le schéma se lira en "zig-zag", en partant de la gauche, de haut en bas, puis en remontant à l'alternative suivante, qu'on lit également en même temps de haut en bas et de gauche à droite. La liaison "haut-bas" se termine à la rencontre de "succès" ou "échec". On remonte alors aux dernières flèches latérales rencontrées, en suivant par la flèche ↺, la lecture "gauche-droite". La résolution s'arrête lorsqu'on ne rencontre plus de flèche ↺.

— les flèches verticales sont relatives au travail effectué sur un but. Les flèches affectées d'un "ET" visualisent le passage au but suivant d'une conjonction (à partir du moment où le but courant a été satisfait). Les autres comportent un numéro relatif à la clause à satisfaire.



? ANCÊTRE (*X0, MARC)
3
4
? PÈRE (*X0, MARC)
2
PÈRE (MICHEL, MARC)
Succès : *X0 = MICHEL
? PÈRE (*X0, *X2)
1
2
PÈRE (JEAN, MICHEL)
ET
? ANCÊTRE (MICHEL, MARC)
PÈRE (MICHEL, MARC)
ET
? ANCÊTRE (MARC, MARC)
3
4
? PÈRE (MICHEL, MARC)
2
PÈRE (MICHEL, MARC)
Succès : *X0 = JEAN
? PÈRE (MICHEL, *X2)
2
PÈRE (MICHEL, MARC)
ET
? ANCÊTRE (MARC, MARC)
3
4
? PÈRE (MARC, MARC)
Échec
? PÈRE (MARC, *X2)
Échec
3
4
? PÈRE (MARC. MARC)
Échec
? PÈRE (MARC, *X2)
Échec
Fin de résolution

- **UTILISATION DE LA COUPURE**

La plupart des règles PROLOG utilisent beaucoup de variables libres dans leurs énoncés, donc vont engendrer, par définition, beaucoup de solutions lors d'une résolution.

Or, souvent, la connaissance de toutes ces solutions n'est pas nécessaire. On ne doit satisfaire le but qu'une fois (et donc on ne veut pas s'intéresser aux autres solutions envisageables).

Exemple :

Reprenons les règles (déjà définies au § 3.3.1.2.3) de la relation d'appartenance d'un terme à une liste :

```
APPARTIENT (*X, (*X ; *Y)).
APPARTIENT (*X, (*T ; *Q)) : APPARTIENT (*X, *Q).
```

Si nous demandons la résolution :

```
$?: APPARTIENT ('E', "ESSENTIELLEMENT")
```

nous voulons ici savoir si le caractère 'E' appartient à la chaîne de caractères "ESSENTIELLEMENT". Notre objectif n'est pas de savoir combien de fois ce caractère apparaît, mais seulement **si** il apparaît.

Or l'interpréteur nous répond : --SUCCÈS--(5)--

Pour signifier à l'interpréteur que nous voulons qu'il ne prenne en compte que la première solution trouvée, et qu'il détruise les choix ultérieurs, nous utiliserons le prédicat prédifini **CUT** (voir § 3.1.2.4-E, p. 117).

Le programme sera modifié de la façon suivante :

```
APPARTIENT (*X, (*X ; *Y)) : CUT.
APPARTIENT (*X, (*T ; *Q)) : APPARTIENT (*X, *Q).
```

La définition de l'effet de coupure s'énonce ainsi : quand le système rencontre une coupure, il s'en tient aux choix effectués depuis le début de la règle contenant la coupure. Toutes les autres possibilités sont exclues. Ainsi on ne pourra plus resatisfaire un but compris entre le début de la règle et la coupure. On ne tiendra pas compte des différentes alternatives obtenues par retour arrière.

Les conséquences sont importantes et sont les suivantes :

- les programmes s'exécutent plus rapidement, car ils ne perdront pas de temps à essayer de satisfaire des buts qui ne vous intéressent plus.
- les programmes utiliseront moins de place mémoire : il devient inutile de mémoriser certains points de retour arrière.

• ABANDON D'UNE RÉSOLUTION

Il se peut que vous vouliez abandonner une résolution en cours.

Pour cela deux solutions :

a. appuyer en continu sur la touche **RAZ**, la résolution s'arrêtera lorsque l'interpréteur affichera une solution. Cette méthode est "propre".

b. appuyer sur le bouton d'initialisation à chaud (INIT) de la machine. Dans la majorité des cas, cela se passera bien, mais il y a un risque (selon l'instant où vous appuyez) de détruire irrémédiablement des structures interne de votre programme, auquel cas celui-ci sera perdu.

En règle générale, la première solution est préférable, mais elle nécessite qu'il y ait, de temps en temps, affichage de solution.

4 - MANUEL DE RÉFÉRENCES

Vous trouverez dans ce chapitre la liste alphabétique de toutes les commandes et de toutes les primitives utilisables en PROLOG, ainsi que leur définition et leur syntaxe.

+ (<terme 1>, <terme 2>, <terme 3>)

Primitive arithmétique d'addition de deux termes <terme 1> et <terme 2>. Le résultat de l'addition est dans <terme 3>.

<terme 1> et <terme 2> sont des nombres entiers signés, ou des variables instanciées avant l'appel par des nombres entiers signés.

APPEND (<terme-tête>)

Primitive de manipulation de clauses par programme. Elle permet d'ajouter la clause désignée par son <terme-tête> (si celui-ci est une variable, elle doit être instanciée au moment de l'appel), à la fin du paquet de clauses de même nom.

APPEND (<terme-tête>, <liste-queue>)

Primitive identique à la précédente, sauf qu'ici un deuxième argument donne la liste des prédicats du corps de la clause. La clause apparaîtra à la fin du sous-programme correspondant, sous sa forme syntaxique normale.

ATOM (<terme>)

Primitive de contrôle du type d'un terme. Elle est évaluée à vrai si <terme> est un atome, ou une variable instanciée par un atome au moment de l'appel.

BACKWARD

Primitive de manipulation de clauses par programme. Elle permet de sélectionner la clause qui précède la clause courante (qui doit donc être définie au moment de l'appel). Si la clause courante est la première du paquet, le prédicat BACKWARD échoue et la cause courante est inchangée.

BOTTOM

Primitive de manipulation de clauses par programme. Elle sélectionne la première clause du dernier paquet de clauses du programme. Celle-ci devient la clause courante.

CH (<nom-d'un-paquet-de-clauses>)

Commande permettant d'utiliser l'une des fonctions de l'éditeur. Elle permet de modifier un paquet de clauses (dont le nom est donné en paramètre) de façon définitive : l'ancienne version est détruite. Vérifiez avant tout choix que les modifications que vous allez apporter ne vont pas faire "déborder" l'éditeur (si elles sont trop conséquentes, seul ce que vous venez de taper sera conservé).

Commande utilisée pour la mise au point des programmes donc des clauses déjà saisies. Les modifications sont prises en compte une fois que l'on a demandé à sortir de l'éditeur par la touche **STOP**.

CHAR (<terme>)

Primitive de contrôle du type d'un terme. Elle est évaluée à vrai si <terme> est un caractère, ou une variable instanciée par un caractère au moment de l'appel.

CLAUSE (<terme-tête>, <variable-libre-de-queue-de-liste>)

Primitive de manipulation de clauses par programme. Elle permet d'examiner et de manipuler en mémoire la clause courante (qui doit être définie au moment de l'appel).

La tête de la clause courante est unifiée à <terme-tête>. L'argument <variable-libre-de-liste-queue>, qui est libre au moment de l'appel, est instanciée par la liste composée des différents prédicats du corps de la clause. Ce prédicat permet donc de traiter le corps d'une clause comme une liste de prédicats. Utilisé en conjonction avec PACK (<nom-de-tête-de-clause>), elle permet de "filtrer" l'examen de clauses, en fonction d'un paquet précis.

CLEAR

Commande de réinitialisation de l'interpréteur : le programme contenu en mémoire est effacé, l'occupation mémoire est remise à zéro (visualisable par la commande STAT).

CLS

Commande d'effacement de l'écran du micro-ordinateur. Cette commande n'a aucun effet sur le programme contenu en mémoire (celui-ci n'est pas effacé).

CLS (<couleur de caractères>, <couleur de fond>)

Commande d'effacement de l'écran (sans pourtant effacer le programme contenu en mémoire), et de modification des couleurs de caractères et de fond de l'écran, selon les conventions habituelles TO7/MO5 (codes de 0 à 7).

<prédicat-de-comparaison> (arg 1>, <arg 2>)

Ces primitives permettent d'effectuer des comparaisons sur des constantes. Au moment de l'appel, les arguements <arg 1> et <ar- <arg 2> à comparer doivent être de même type : c'est-à-dire soit des nombres entiers signés, soit des atomes, soit des caractères. Si l'un des arguments est une variable, celle-ci doit être au moment de l'appel instanciée par une constante de même type que l'autre argument.

Les nombres entiers sont comparés par ordre de grandeur, les atomes et les caractères par ordre alphabétique.

Les prédicats de comparaison sont :

< évalué à vrai si <arg 1> < <arg 2<.
= évalué à vrai si <arg 1> > <arg 2>.
> évalué à vrai si <arg 1> > <arg 2>.
<= évalué à vrai si <arg 1> <= <arg 2>.
>= évalué à vrai si <arg 1> >= <arg 2>.
<> évalué à vrai si <arg 1> <> <arg 2>.

CONSTANT (<terme>)

Primitive de contrôle du type d'un terme. Elle est évaluée à vrai si <terme> est une constante, ou une variable instanciée par une constante au moment de l'appel.

CUT

Primitive de contrôle du fonctionnement de l'interpréteur. Ce prédicat est un "coupe-choix", c'est-à-dire qu'il "fige" les choix effectués depuis l'appel de la clause courante (dans lequel se trouve le CUT). Ainsi, les choix déjà effectués ne seront pas remis en question lors du retour arrière.

DEL (<nom-d'un-paquet-de-clauses>)

Commande permettant de détruire un paquet de clauses ou sous-programme (dont le nom est donné en paramètre) présent en mémoire. La commande sera plus ou moins longue à s'effectuer selon la taille du paquet.

DEL (<nom-d'un-paquet-de-clauses>, <rang-de-la-clause>)

Commande permettant de détruire une clause d'un paquet de nom donné en paramètre. Cette clause est spécifiée plus précisément par son rang dans le paquet (deuxième paramètre de la commande), correspondant au numéro placé entre les deux signes "=" devant la clause, lors de l'action de la commande LIST (<nom-d'un-paquet-de-clauses>).

DIC

Commande permettant de lister à l'écran l'état et le contenu exact du dictionnaire. L'appui sur n'importe quelle touche du clavier provoque l'arrêt momentané du défilement (pour le reprendre, répéter cette opération). L'appui sur la touche **RAZ** provoque l'abandon de la commande.

Le dictionnaire répertorie, par ordre d'apparition à la saisie, les noms d'atomes et de prédicats définis par l'utilisateur.

Ces noms sont suivis du nombre de fois où ils apparaissent dans le programme, et d'un caractère spécifiant plus précisément leur type ("A" pour atome constant; "C" pour nom de tête de clause, de prédicat ou de structure, "V" pour affectation d'une valeur à un atome, après utilisation de la primitive VALUE (<source>, <destination>)).

DIFF (<terme 1>, <terme 2>)

Primitive de comparaison de deux termes. Elle est évaluée à vrai si les deux termes sont différents. Si les termes sont des variables, on vérifie donc qu'elles ne sont pas instanciées par le même objet, ou qu'elles ne sont pas liées entre elles.

DIR (X)

Commande permettant de visualiser à l'écran les fichiers sauvegardés sur une disquette. X est le numéro du lecteur utilisé (de 0 à 4). Par défaut, c'est le lecteur 0 qui est utilisé.

Le catalogue de fichiers se présente sur deux colonnes, selon le format suivant ("A A" indique que les fichiers sont de type texte) : <nom du fichier> <extension> A A <taille en Ko>.

DIV (<terme 1>, <terme 2>, <terme 3>)

Primitive arithmétique de division entière de deux termes, <terme 1> et <terme 2> (le reste de la division est perdu). Le résultat de la division est dans <terme 3>.

<terme 1> et <terme 2> sont des nombres entiers signés, ou des variables instanciées, avant l'appel, par des nombres entiers signés. Si <terme 2> est nul, la division par zéro est détectée : il y a envoi d'un message d'erreur et abandon de l'exécution.

DOFF

Primitive et commande d'édition à la résolution. Elle supprime l'affichage des solutions trouvées, lors d'une résolution, permettant ainsi à l'utilisateur de "formater" lui-même, et à sa convenance, la présentation des solutions. L'effet persiste sur toutes les résolutions ultérieures, tant que l'on ne l'a pas supprimé par DON.

DON

Cette primitive ou commande rétablit l'affichage systématique des solutions lors d'une résolution.

ED

Commande permettant d'utiliser l'une des fonctions de l'éditeur. L'éditeur utilise les zones mémoire NŒUDS et PILE pour stocker momentanément le texte à saisir. La commande ED permet de rentrer simplement dans l'éditeur, afin de saisir une ou plusieurs lignes de texte. Elle entraîne l'affichage d'une page vierge bleue de vingt lignes. Le curseur est positionné dans le coin supérieur gauche de l'écran et attend la saisie d'une ou plusieurs clauses. Commande indispensable pour saisir un programme (puisqu'elle permet de travailler en mode page). Une fois la saisie terminée, elle est validée par appui sur la touche **STOP**, qui entraîne la sortie de l'éditeur et l'analyse du texte saisi. Il est conseillé de faire la saisie par petits paquets de clauses (pour éviter que l'éditeur ne déborde).

ED (<nom-d'un-paquet-de-clauses>)

Commande permettant d'utiliser une autre fonction de l'éditeur. Elle se rapporte à un paquet de clauses déjà en mémoire (donné en paramètre). Elle a pour effet d'afficher une fenêtre bleue remplie des clauses demandées. Cette commande permet de créer un nouveau paquet de clauses à partir de celui proposé, sans pour autant détruire les clauses initiales : les nouvelles clauses sont rajoutées au programme déjà en mémoire. Commande très utile pour saisir des paquets de clauses qui se ressemblent.

EQ (<terme 1>, <terme 2>)

Cette primitive teste l'égalité stricte de deux termes, <terme 1> et <terme 2>. Elle est évaluée à vrai si les deux termes (simples ou composés) sont strictement égaux. C'est-à-dire que s'ils contiennent des variables, celles-ci doivent avoir été instanciées par des termes identiques, ou liées par unification (donc elles correspondent au même objet).

ERASE

Primitive de manipulation de clauses par programme. Elle a pour effet de détruire la clause courante (qui doit donc être définie). La clause suivante (si elle existe) devient la clause courante, sinon celle-ci est indéterminée.

Aucun contrôle n'est effectué sur la clause à détruire : le comportement de l'interpréteur est ici aléatoire et imprévisible, ce qui peut entraîner la destruction du programme en mémoire. Il faut donc manier cette primitive avec prudence, et vérifier en particulier son incidence sur la résolution en cours.

FORMAT (X)

Commande d'initialisation d'une disquette. X représente le numéro du lecteur utilisé (de 0 à 4).

FORWARD

Primitive de manipulation de clauses par programme. Elle permet de sélectionner la clause qui suit la clause courante dans un sous-programme.

Si la clause courante initiale était la dernière clause du paquet, le but échoue et la primitive est évaluée à faux. Sinon, la clause suivante devient la nouvelle clause courante.

GET (<terme>)

Primitive d'Entrée/Sortie de texte. Elle permet de lire tout objet PROLOG, entré au clavier par l'utilisateur, ou de créer interactivement des clauses par programme.

Le texte saisi par l'utilisateur devra être du même type que celui donné en argument de la primitive GET (terme quelconque ou même prédicat).

L'exécution d'un GET entraîne le passage à la ligne suivante à l'écran et l'affichage en début de ligne du symbole "> : l'interpréteur attend une saisie au clavier, l'utilisateur a 38 caractères pour taper son texte (en mode commande). Une fois le texte saisi, celui-ci est validé par appui sur la touche **ENTRÉE** et passé à l'analyseur syntaxique.

Si une erreur de syntaxe est détectée, un message s'affiche et la résolution est arrêtée.

GETC (<suite-de-caractères>)

Primitive d'Entrée/Sortie de texte. Elle permet de lire caractère par caractère une suite entrée au clavier par l'utilisateur (correspondant à l'instruction READ en PASCAL).

L'exécution d'un GETC entraine le passage à la ligne suivante et l'affichage d'un "= : l'utilisateur entre une suite de caractères et valide sa saisie par l'appui sur la touche **ENTRÉE**. L'interpréteur lit les caractères un par un jusqu'à ce qu'il rencontre ENTRÉE. Ceci permet de lire un mot caractère par caractère.

INSERT (<terme-tête>)

Primitive de manipulation de clauses par programme. Cette primitive permet d'insérer avant la clause courante une clause dans un sous-programme. La clause à insérer est donnée par son <terme-tête> (si c'est une variable, elle doit être instanciée au moment de l'appel). Elle apparaîtra dans le programme sous la forme syntaxique normale d'une clause.

La clause courante doit normalement avoir le même nom de prédicat de tête. Si ce n'est pas le cas, la clause à insérer sera ajoutée au début du sous-programme correspondant.

INSERT (<terme-tête>, <liste-queue>)

Primitive identique à la précédente, sauf qu'ici un deuxième argument donne la liste des prédicats du corps de la clause. La clause apparaîtra dans le programme sous sa forme syntaxique normale.

INTEGER (<terme>)

Primitive de contrôle du type d'un terme. Elle est évaluée à vrai si <terme> est un nombre entier signé, ou une variable instanciée par un nombre entier signé au moment de l'appel.

KILL ('<X> : <nom-de-fichier>)

Commande de destruction du fichier <nom-de-fichier> sauvegardé sur une disquette placée sur le lecteur X. Ceci entraîne l'affichage du catalogue des fichiers mis à jour.

LINE

Primitive d'Entrée/Sortie de texte permettant de "formater", par ligne, l'affichage à l'écran. Elle provoque lors d'une résolution, un saut de ligne sur l'écran à partir de la ligne courante.

LIST

Commande permettant de visualiser à l'écran la totalité du programme en mémoire. Pour arrêter le défilement, appuyez sur une touche du clavier (répétez cette opération pour continuer). L'appui sur la touche **RAZ** provoque l'abandon de la commande.

Le listing est découpé par paquet de clauses. Chaque clause mémorisée est précédée d'un numéro entre deux signes "=" indiquant son rang dans le paquet, et d'un nombre entre "<>" donnant le nombre de termes mémoire qu'elle occupe.

LIST (<nom-d'un-paquet-de-clauses>)

Commande permettant de ne lister à l'écran que les clauses du paquet dont le nom est donné en paramètre. Le formatage du listing est le même que pour la commande LIST.

LOAD

Commande de lecture d'un fichier préalablement sauvegardé sur cassette. La lecture d'un fichier nécessite une taille mémoire (allouée par SIZE) au moins aussi grande que celle utilisée lors de la sauvegarde du programme.

Le nom du fichier n'étant pas précisé, le programme lit le premier fichier source rencontré sur la cassette et essaye de le compiler.

Dès que le programme a trouvé le premier fichier, son nom est affiché entre parenthèses (et non entre apostrophes) sur la ligne suivante. Il essaye alors de le dire, tout en le compilant : dès qu'un paquet de clauses lu est complet, il est compilé, et le nombre de termes mémoire qu'il occupe est affiché (X TERMES). Dès que la lecture du fichier est terminé, le symbole "$" réapparaît à l'écran.

LOAD ('C : <nom-de-fichier>')

Commande de lecture d'un fichier préalablement enregistré sur cassette. Le programme lit séquentiellement la bande cassette jusqu'à ce qu'il tombe sur le fichier dont le nom est donné en paramètre. Cette recherche sera d'autant plus rapide que l'on aura pris soin de "mémoriser" au compteur du magnétophone (lors de la sauvegarde du fichier) sa position relative sur la bande.

Le principe de lecture est le même que précédemment.

LOAD ('<X> : <nom-du-fichier>')

Commande de lecture d'un fichier préalablement enregistré sur disquette. <X> représente le numéro du lecteur de disque utilisé (de 0 à 4). Le lecteur de disquettes se positionne automatiquement et directement à l'emplacement du fichier dont le nom est donné en paramètre. Le principe de lecture du fichier est toujours le même.

MOD (<terme 1>, <terme 3>, <terme 3>)

Primitive arithmétique du modulo de deux termes, <terme 1> et <terme 2>. La définition du modulo est la même qu'en BASIC : elle correspond au reste de la division entière de <terme 1> par <terme 2>. Le résultat du modulo est dans <terme 3>.

<terme 1> et <terme 2> sont des nombres entiers signés, ou des variables instanciées au moment de l'appel par des nombres entiers signés.

*(<terme 1>, <terme 2>, <terme 3>)

Primitive arithmétique de multiplication de deux termes, <terme 1> et <terme 2>. Le résultat de la multiplication est dans <terme 3>.

<terme 1> et <terme 2> sont des nombres entiers signés, ou des variables instanciées au moment de l'appel par des nombres entiers signés.

NOT (<prédicat>)

Primitive de contrôle du fonctionnement de l'interpréteur. Ce prédicat est évalué à vrai, si le prédicat énoncé en argument échoue. C'est-à-dire qu'il permet de vérifier qu'il n'existe pas de solutions pour le but donné en argument.

PACK (<nom-d'une-tête-de-clause>)

Primitive de manipulation de clauses par programme. Elle permet de sélectionner la première clause du paquet de clauses, dont le nom (qui correspond à celui du prédicat de tête de toutes les clauses constituant le sous-programme) est donné en argument.

Si le nom de cette clause existe (donc si le sous-programme existe), celle-ci devient la clause courante, sinon la primitive est évaluée à faux.

L'argument de PACK peut être une variable libre. Dans ce cas :

— si aucune clause courante n'est définie au moment de l'appel, PACK sélectionne la première clause du premier paquet de clauses, qui devient la clause courante (effet identique à celui de la primitive de manipulation de clauses TOP).

— sinon, PACK sélectionne la première clause du premier paquet de clauses qui suit la clause courante.

— dans tous les cas, la variable libre est instanciée par le prédicat de tête de la clause sélectionnée.

PAUSE

Primitive d'édition à la résolution. Elle permet à l'utilisateur d'arrêter momentanément l'affichage des solutions qu'il a lui-même programmées.

A chaque fois que cette primitive est évaluée, l'interpréteur effectue une scrutation du clavier :

— si une touche est enfoncée (c'est-à-dire que vous voulez arrêter momentanément l'affichage des solutions), il suspend la résolution.

— pour la reprendre, l'interpréteur attend que vous appuyiez une nouvelle fois sur une touche quelconque du clavier.

— la touche **RAZ** provoque l'abandon définitif de la résolution.

PRINT

Primitive de manipulation de clauses par programme. Ce prédicat, sans argument, permet d'afficher à l'écran la clause courante (la tête et la queue), sous sa forme normale.

PRINTON

Primitive et commande (sans argument) d'affichage d'Entrée/Sortie de texte. PRINTON demande l'affichage simultané sur écran et sur imprimante, de tout ce qui sera affiché ultérieurement à l'écran (sauf les caractères de contrôle édités par PUT). Son effet persiste tant qu'il n'a pas été supprimé par la primitive ou commande PRINTOFF.

Cette primitive réussit, bien sûr, à partir du moment où l'imprimante est bien connectée.

PRINTOFF

Primitive et commande d'affichage d'Entrée/Sortie de texte. Elle supprime l'effet de PRINTON.

PUT (terme 1, terme 2, ..., terme N)

Primitive d'affichage d'Entrée/Sortie de texte par programme. Elle permet l'affichage à l'écran de tout terme PROLOG, y compris les prédicats et les termes composés. Elle peut comporter jusqu'à 15 termes en arguments. Si un argument est une variable, celle-ci doit être instanciée au moment de l'appel. Elle est particulièrement utilisée pour envoyer à l'utilisateur des messages de type "texte".

?: (Résolution de programme)

Commande particulière qui permet de demander une résolution portant sur le programme en mémoire. Sa syntaxe est :

?: (<nom-but>)
?(<nom-question>) : (<nom-but>)

Les deux formulations sont équivalentes. Cependant, par souci de clarté et de simplicité, on préférera utiliser la première.

— Le <nom-but> correspond à un prédicat, ou à une conjonction de prédicats à évaluer (prédicats définis par l'utilisateur et/ou prédéfinis, reliés par des "&").

— Le <nom-question> est un atome. Dans ce cas, la demande de résolution est stockée en fin de programme, pendant la durée de la résolution, et détruite à la fin.

Résoudre un problème en PROLOG consiste donc à donner à l'interpréteur des prédicats ou *buts* à évaluer, en fonction d'une problématique précise.

Si l'on veut abandonner une résolution en cours, il suffit d'appuyer sur la touche **RAZ** ou sur le bouton INIT (attention, il y a un risque, faible, de destruction du programme).

SAVE ('C : <nom-de-fichier>')

Commande de sauvegarde d'un fichier sur cassette.

Il est conseillé de sauvegarder tout programme avant de l'exécuter.

Le <nom-de-fichier> correspond au nom que vous voulez donner au programme à sauvegarder. Le "C :" est facultatif (cassette par défaut).

Il doit comporter au plus huit caractères. Si vous avez besoin d'une extension pour le nom (permettant d'identifier plus précisément le type du fichier), celle-ci comportera au plus trois caractères et sera séparée du nom du fichier par un point. Par défaut, l'extension du nom de fichier est PRG (pour programme).

Avant de valider la commande par la touche **ENTRÉE**, positionnez votre cassette à l'endroit où vous voulez que soit effectué l'enregistrement (en notant le numéro donné au compteur). Appuyez simultanément sur les touches **"Play"** et **"Enreg. Rec"**, puis lancez la commande de sauvegarde. Une fois celle-ci effectuée, stoppez votre magnétophone.

Le programme est sauvé sous forme de programme source décompilé, c'est-à-dire comme lorsqu'on demande un listing à l'écran. La sauvegarde s'effectue paquet de clauses par paquet de clauses. Un paquet de clauses trop important nécessitera des sauvegardes successives. Pendant la sauvegarde, un ou des caractères "#" s'affichent à l'écran, visualisant la progression du travail. Une fois la sauvegarde terminée, le symbole "$" réapparaît à l'écran.

SAVE ('<X> : <nom-de-fichier>')

Commande de sauvegarde ou enregistrement sur disquette du programme contenu en mémoire.

Le <nom-de-fichier> et le déroulement de la sauvegarde sont identiques à ceux vus ci-dessus.

<X> représente le numéro du lecteur de disquettes utilisé (de 0 à 4).

Une fois la commande validée, le lecteur de disquettes gère lui-même l'emplacement de sauvegarde sur la disquette, en fonction de la place disponible.

SIZE (N1, N2, N3, N4, N5)

Commande d'extension par structures de la taille initiale de la mémoire. Celle-ci est nécessaire lorsque la taille du programme à saisir dépasse les limites mémoire proposées par l'interpréteur (voir commande STAT).

Elle a également pour effet de remettre à zéro toute la mémoire (donc par définition, d'effacer le programme qui pouvait s'y trouver). Il est recommandé de l'utiliser avant toute saisie de programme, et en fonction de la taille supposée de celui-ci.

La commande SIZE comprend cinq paramètres obligatoires, qui sont des entiers, donnant le nombre de blocs de 256 octets alloués à chacune des structures présentes en mémoire :

— N1 : taille du dictionnaire (un élément occupe 13 octets).

— N2 : taille de la pile des clauses (un élément occupe 5 octets).

— N3 : taille de la pile des termes (un élément occupe 4 octets).

— N4 : taille de la zone de stockage des textes (un texte occupe un octet par caractère + un octet pour le texte).

— N5 : taille de la zone des nœuds de résolution (un élément occupe 12 octets).

Le reste de la mémoire est alloué à la pile.

La taille mémoire occupée par la zone des nœuds et la zone de la pile doit être au maximum de 3 Ko, sinon une erreur est signalée.

— (<terme 1>, <terme 2>, <terme 3>)

Primitive arithmétique de soustraction sur des nombres entiers signés.

<terme 1> et < terme 2>, qui sont les arguments de l'opération, doivent être des nombres entiers signés, ou des variables instanciées au moment de l'appel par des nombres entiers signés. Le résultat de la soustraction est dans <terme 3>.

STAT

Commande d'examen des structures mémoire et de leur occupation. Elle permet de visualiser, d'une part la taille initiale de chacune des structures mémoire (-1- MÉMOIRE), et d'autre part la place mémoire occupée par votre programme dans les structures précitées (-2- OCCUPATION), c'est-à-dire le nombre d'éléments occupés sur le nombre total de libres.

Son action entraîne et lance le "garbage" (récupérateur de mémoire) sur les structures TERMES et TEXTE (si vous venez de supprimer des clauses de votre programme).

La rubrique -1- MÉMOIRE donne l'adresse, en hexadécimal, du premier et de dernier octet de chaque structure, donc l'occupation maximale de chaque zone (modifiable par la commande SIZE (N1, N2, N3, N4, N5)).

Par définition, lorsqu'il n'y a aucun programme présent en mémoire, l'occupation des différentes structures est de 0 (sauf pour les "termes" où il y a quatre termes stockés initialement, pour les besoins de l'interpréteur).

On distingue les structures statiques (permettant de représenter le programme en mémoire sous forme interne) et les structures dynamiques (utilisés par l'interpréteur pour effectuer une résolution). Ces structures se comportent comme des piles. Elles sont considérées comme pleines lorsqu'elles ne possèdent plus qu'un élément libre.

— Les structures statiques

- Le dictionnaire contenant tous les noms de prédicats ou d'atomes définis par l'utilisateur mémorisés sur 13 octets (DICO).
- La "pile" des clauses : chaque clause est associée à un élément de cette pile (sur 5 octets), qui représente l'ensemble de la clause (CLAUSES).
- La "pile" des termes : chaque terme composant d'une clause y est mémorisé sur 4 octets (TERMES).
- La "pile" des textes : chaque caractère d'un texte y est mémorisé (TEXTE).

— **Les structures dynamiques** occupant au moins 3 Ko

- La "pile" des nœuds de résolution, dont chaque élément occupe 12 octets (NŒUDS).
- La "pile" des substitutions, dont chaque élément occupe en moyenne 7 octets (PILE).
- Le compteur d'unification (sur 16 bits) donnant pour la dernière résolution, le nombre d'unifications réussies sur le nombre total essayé.

STRUCT (<terme>)

Primitive de contrôle du type d'un terme. Elle est évaluée à vrai si <terme> est un terme composé, ou une variable instanciée par un terme composé au moment de l'appel.

TOP

Primitive de manipulation de clauses. Ce prédicat sans argument permet de sélectionner la première clause du premier paquet de clauses du programme. Cette clause devient la clause courante.

TROFF

Primitive et commande d'édition à la résolution. Elle permet d'annuler l'effet du prédicat ou de la commande TRON.

TRON

Primitive et commande d'édition à la résolution. Elle permet de suivre à l'écran une demande de résolution, en affichant les différentes évaluations successives du but initial à évaluer (ce qui ralentit considérablement la durée de l'exécution). On dit que TRON arme le mode trace.

Utilisé en tant que prédicat, il permet de déclencher le mode trace pour l'évaluation d'un but précis. Son effet persiste tant qu'il n'a pas été annulé par TROFF.

Pour arrêter le défilement de l'écran, il suffit d'appuyer sur une touche quelconque du clavier. Pour reprendre l'exécution du programme et l'affichage de la trace, on réappuie sur une touche du clavier.

L'appui sur la touche **RAZ** fait sortir du mode trace et la résolution continue en mode normal.

Un deuxième appui sur la touche **RAZ** provoque l'abandon de la résolution.

VALUE (<source>, <destination>)

Primitive correspondant à l'affectation d'une variable globale classique. Elle permet de mémoriser un nombre ou un caractère dans

un atome, qui ne doit pas être une tête de clause, et de le récupérer ultérieurement dans une variable. Cette affectation subsiste entre deux résolutions, sauf si l'atome est devenu une tête de clause. Ainsi, on peut mémoriser momentanément une valeur.

<source> : la source correspond à la valeur qu'on veut utiliser et mémoriser. Elle est soit un nombre, soit un caractère, soit un atome, soit une variable (qui doit être obligatoirement instanciée par l'un des trois termes précédents au moment de l'appel).

Si <source> est un atome, c'est le contenu antérieur de cet atome qui sera utilisé.

<destination> : <destination> désigne ce qui doit recevoir l'objet issu de <source> (cet objet est donc soit un nombre, soit un caractère), <destination> est un atome ou une variable. Si la variable est instanciée, elle doit l'être par un atome.

VAR (<terme>)

Primitive de contrôle du type d'un terme. Ce prédicat est évalué à vrai si <terme> est une variable libre ou unifiée (non encore instanciée par un terme autre qu'une variable).

IV - ANNEXES

1 - EXEMPLES DE PROGRAMMES 159
1.1 - Une base de données "familiale" : un arbre généalogique 159
1.2 - Une base de données : les employés d'une entreprise 162
1.3 - Le programme MUTANT 164
1.4 - Les Tours de Hanoï 166
1.5 - Problème de traversée de rivière 169
1.6 - Problème : SEND + MORE = MONEY? ... 171
1.7 - Définition d'un "micro-monde" de robots .. 173
1.8 - Analyse du langage 177

2 - LEXIQUE 181

3 - INDEX 187

4 - LISTE DES MESSAGES D'ERREURS 191

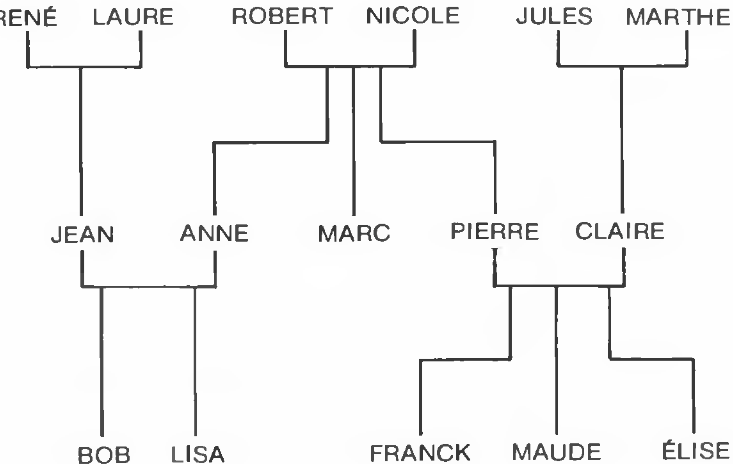
5 - LISTE DES CARACTÈRES DE CONTROLE .. 199

BIBLIOGRAPHIE 200

1 - EXEMPLES DE PROGRAMMES

1.1 - UNE BASE DE DONNÉES "FAMILIALE" : UN ARBRE GÉNÉALOGIQUE.

Supposons que des recherches aient été faites sur l'ascendance d'une famille, aboutissant à la création de l'arbre généalogique suivant :



En utilisant les prédicats PÈRE et MÈRE, nous créons la base de faits représentant l'arbre généalogique ci-dessus (correspondant aux relations élémentaires symbolisées par les flèches de l'arbre).

Les règles nous permettent de définir des relations de parenté.

```
$LIST
= 1 =<3> PÈRE (RENÉ, JEAN).
= 2 =<3> PÈRE (ROBERT, ANNE).
= 3 =<3> PÈRE (ROBERT, MARC).
= 4 =<3> PÈRE (ROBERT, PIERRE).
= 5 =<3> PÈRE (JULES, CLAIRE).
= 6 =<3> PÈRE (JEAN, BOB).
= 7 =<3> PÈRE (JEAN, LISA).
= 8 =<3> PÈRE (PIERRE, FRANCK).
= 9 =<3> PÈRE (PIERRE, MAUDE).
= 10 =<3> PÈRE (PIERRE, ÈLISE).
```

```
= 1 =<3> MÈRE (LAURE, JEAN).
= 2 =<3> MÈRE (NICOLE, ANNE).
= 3 =<3> MÈRE (NICOLE, MARC).
= 4 =<3> MÈRE (NICOLE, PIERRE).
= 5 =<3> MÈRE (MARTHE, CLAIRE).
= 6 =<3> MÈRE (ANNE, BOB).
= 7 =<3> MÈRE (ANNE, LISA).
= 8 =<3> MÈRE (CLAIRE, FRANCK).
= 9 =<3> MÈRE (CLAIRE, MAUDE).
= 10 =<3> MÈRE (CLAIRE, ÉLISE).

= 1 =<6> PARENT (*X0, *X1) :
         PÈRE (*X0, *X1).
= 2 = <6> PARENT (*X0, *X1) :
          MÈRE (*X0, *X1).

= 1 = <6> ENFANT (*X0, *X1) :
          PARENT (*X1, *X0).

= 1 =<6> ANCÊTRE (*X0, *X1) :
         PARENT (*X0, *X1).
= 2 =<9> ANCÊTRE (*X0, *X1) :
         PARENT (*X0, *X2) &
         ANCÊTRE (*X2, *X1).

= 1 =<18> FR-SO (*X0, *X1) :   -> relation "être frère ou
                                  sœur".
          PÈRE (*X2, *X0) &
          PÈRE (*X2, *X1) &
          MÈRE (*X3, *X0) &
          MÈRE (*X3, *X1) &
          <> (*X0, *X1).

= 1 =<9> G-PARENT (*X0, *X1) : -> relation "être grand-père".
         PARENT (*X0, *X2) &
         PARENT (*X2, *X1).

= 1 =<9> ONC-TANT (*X0, *X1) : -> relation "être oncle ou tante
         PARENT (*X2, *X1) &      direct".
         FR-SO (*X0, *X2).

= 1 =<9> COUSINS (*X0, *X1) :  -> relation "être cousin ou
         ONC-TANT (*X2, *X0) &    cousine".
         ENFANT (*X1, *X2).
```

EXEMPLES DE RÉSOLUTIONS SUR LA BASE DE DONNÉES FAMILIALE

● Pour demander à l'interpréteur de donner tous les descendants de ROBERT :

```
$?: ANCÊTRE (ROBERT, *DESCEND)
*DESCEND = ANNE
*DESCEND = MARC
*DESCEND = PIERRE
*DESCEND = BOB
*DESCEND = LISA
*DESCEND = FRANCK
*DESCEND = MAUDE
*DESCEND = ÉLISE
--SUCCÈS--(8)--
```

Remarquez que l'ordre dans lequel sont données les solutions correspond aux différentes étapes de résolution par l'interpréteur, du but ANCÊTRE (donnant lieu à des appels récursifs et des retours arrière).

● Pour demander à l'interpréteur de donner le nom des enfants de PIERRE :

```
$?: ENFANT (*ENFANT, PIERRE)
*ENFANT = FRANCK
*ENFANT = MAUDE
*ENFANT = ÈLISE
--SUCCÈS--(3)--
```

● Pour demander le nom des parents de MAUDE :

```
$?: PARENT (*PP, MAUDE)
*PP = PIERRE
*PP = CLAIRE
--SUCCÈS--(2)--
```

● Pour demander le nom des oncles et tantes directs de MAUDE :

```
$?: ONC-TANT (*X, MAUDE)
*X = ANNE
*X = MARC
--SUCCÈS--(2)--
```

● Pour demander le nom des neveux et nièces de MARC :

```
$?: ONC-TANT (MARC, *N)
*N = FRANCK
*N = MAUDE
*N = ÉLISE
*N = BOB
*N = LISA
--SUCCÉS--(5)--
```

- Pour demander le nom des cousins et cousines de BOB :

```
$?: COUSINS (*COUSIN, BOB)
*COUSIN = FRANCK
*COUSIN = MAUDE
*COUSIN = ÈLISE
--SUCCÈS--(3)--
```

- Pour demander le nom des frères et sœurs de MARC :

```
$?: FR-SO (*FR-SO, MARC)
*FR-SO = ANNE
*FR-SO = PIERRE
--SUCCÈS--(2)--
```

- Pour demander le nom des grands-parents de LISA :

```
$?: G-PARENT (*G-PARENT, LISA)
*G-PARENT = RENÈ
*G-PARENT = ROBERT
*G-PARENT = LAURE
*G-PARENT = NICOLE
--SUCCÈS--(4)--
```

- Pour demander tous les ancêtres de FRANCK :

```
*ANC = PIERRE
*ANC = CLAIRE
*ANC = ROBERT
*ANC = JULES
*ANC = NICOLE
*ANC = MARTHE
--SUCCÈS--(6)--
```

1.2 - UNE BASE DE DONNÉES : LES EMPLOYÉS D'UNE ENTREPRISE

Le programme ci-dessous constitue une "petite" base de données des employés d'une entreprise. Plusieurs relations ont été définies :

— une relation d'identification :
ID (<n°-de-code>, <prénom>, <nom>)
association pour chaque individu de l'entreprise, son numéro d'identification interne, son nom et son prénom. Les noms et prénoms sont donnés sous forme de texte (non analysables mais occupant moins de place mémoire).

— une relation SAL (<n°-de-code>, <montant-du-salaire>) donnant, pour un individu reconnu par son <n°-de-code> (dans le but de conserver un certain anonymat, et d'éviter de reprendre tous les arguments cités dans la relation précédente), le montant de son salaire.

— une relation d'information sur la fonction de l'individu dans l'entreprise :
INFO (<n°-de-code>, <sexe>, <département>)
donnant, pour chaque individu reconnu par son <n°-de-code>, son sexe et son lieu de travail.

```
$LIST
= 1 =<1> ID (108, 'JEAN-PIERRE', 'DURAND').
= 2 =<4> ID (123, 'ANDRÉ', 'DE LA MARRE').
= 3 =<4> ID (234, 'LUCIEN', 'DUBOIS').
= 4 =<4> ID (112, 'ANNE', 'FAGER').
= 5 =<4> ID (105, 'MARC', 'DUCLOS').
= 6 =<4> ID (222, 'CLAIRE', 'HAUTIN').
= 7 =<4> ID (101, 'ANTOINE', 'MARTIN').

= 1 =<3> SAL (108, 7600).
= 2 =<3> SAL (123, 5600).
= 3 =<3> SAL (234, 9800).
= 4 =<3> SAL (112, 8000).
= 5 =<3> SAL (105, 6900).
= 6 =<3> SAL (222, 6500).
= 7 =<3> SAL (101, 10000).

= 1 =<4> INFO (108, HOMME, VENTE).
= 2 =<4> INFO (123, HOMME, VENTE).
= 3 =<4> INFO (234, HOMME, FABRIC).
= 4 =<4> INFO (112, FEMME, GESTION).
= 5 =<4> INFO (105, HOMME, FABRIC).
= 6 =<4> INFO (222, FEMME, VENTE).
= 7 =<4> INFO (101, HOMME, DIRECT).
```

La clause (ou sous-programme) SUPER (*X0) permet de lister tous les employés dont le salaire est supérieur à un plancher donné en argument.

```
= 1 =<24> SUPER (*X0) :
          LINE &
          PUT ('EMPLOYÉS AYANT UN SALAIRE > A:', *X0,
          FRS') &
          ID (*X1, *X2, *X3) &
          SAL (*X1, *X4) &
          < (*X0, *X4) &
          LINE &
          PUT (*X2, ' ', *X3, 'NRO:', *X1).
```

La clause SUPER formate l'affichage des solutions (par utilisation des prédicats prédéfinis LINE et PUT).

● Pour connaître les employés dont le salaire est supérieur à 5000 francs :

```
$?: SUPER (5000)
EMPLOYÉS AYANT UN SALAIRE > A: 5000 FRS
JEAN-PIERRE DURAND NRO : 108
ANDRÉ DE LA MARRE NRO : 123
LUCIENT DUBOIS NRO : 234
ANNE FAGER NRO : 112
MARC DUCLOS NRO : 105
CLAIRE HAUTIN NRO : 222
ANTOINE MARTIN NRO : 101
--SUCCÈS--(7)--
```

● Pour connaître les employés dont le salaire est supérieur à 7000 francs :

```
$?: SUPER (7000)
EMPLOYÉS AYANT UN SALAIRE > A: 7000 FRS
JEAN-PIERRE DURAND NRO : 108
LUCIEN DUBOIS NRO : 234
ANNE FAGER NRO : 112
ANTOINE MARTIN NRO : 101
--SUCCÈS--(4)--
```

● Pour connaître les employés dont le salaire est supérieur à 8000 francs :

```
$?: SUPER (8000)
EMPLOYÉS AYANT UN SALAIRE > A: 8000 FRS
LUCIEN DUBOIS NRO : 234
ANTOINE MARTIN NRO : 101
--SUCCÈS--(2)--
```

● Pour connaître les employés dont le salaire est supérieur à 10000 francs :

```
$?: SUPER (10000)
EMPLOYÉS AYANT UN SALAIRE > A : 10000 FRANCS
--ÉCHEC--
```

1.3 - LE PROGRAMME MUTANT

Ce programme part d'une base de données composées de noms relatifs à un domaine précis (ici, les animaux), donnés sous forme de chaînes de caractères (donc de listes particulières permettant l'analyse des caractères qui les composent).

Il affiche tous les "mutants", c'est-à-dire tous les noms composés par deux chaînes de la base, telles que la fin de l'une des chaînes corresponde au début de l'autre.

Il est basé sur l'utilisation du sous-programme de concaténation de deux listes CONCAT. La concaténation de deux listes consiste à créer une nouvelle liste, composée de la première, concaténée (ou mise bout à bout) avec le début de la deuxième.

Exemple :
Si on veut concaténer les deux chaînes de caractères : "bon" et "jour", on obtiendra la chaîne de caractères "bonjour".

● PRINCIPES DE CONCAT (*X0, *X1, *X2).

● CONCAT est un prédicat à trois arguments : les deux premiers correspondent aux deux listes à concaténer, le troisième à la liste de concaténation résultante.

Exemples :

```
$?: CONCAT ((A, B, C), (1, 2), (A, B, C, 1, 2))
--SUCCÈS--(1)--
$?: CONCAT ("BON", "JOUR", *X0)
*X0 = "BONJOUR"
--SUCCÈS--(1)--
$?: CONCAT (*X0, (B, C, D), (A, B, C, D))
*X0 = (A)
--SUCCÈS--(1)--
```

● Comment fonctionne CONCAT ?

La concaténation de deux listes s'effectue en mémorisant progressivement tous les termes de la première liste, dans une liste intermédiaire, jusqu'à arriver à son dernier élément. Il suffit alors de considérer que la queue de la liste intermédiaire correspond à la deuxième liste donnée en argument. Aucun traitement n'est donc effectué sur les termes qui composent la deuxième liste.

La condition d'arrêt est donc que la première liste n'ait plus qu'un élément.

● CONCAT s'énonce de la façon suivante :

```
CONCAT ((*X0), *X1, (*X0, *X1)).
CONCAT ((*X0, *X1), *X2, (*X0, *X3)) : CONCAT (*X1, *X2,
*X3).
```

● Utilisation de CONCAT dans le programme MUTANT :

Pour chaque couple de faits de la base, on essaye de satisfaire la concaténation des deux chaînes de caractères correspondantes, de la manière suivante : on regarde si la fin de la première chaîne de caractères (qui est diminuée d'un élément, c'est-à-dire de son premier caractère, à chaque appel de CONCAT) peut correspondre au début de l'autre : on concatène alors le début de la première chaîne de caractères avec la deuxième.

Listing du programme MUTANT :

```
$LIST
= 1 =<20> ANIMAL ("ALLIGATOR")
= 2 =<14> ANIMAL ("TORTUE")
= 3 =<16> ANIMAL ("CARIBOU").
= 4 =<10> ANIMAL ("OURS").
= 5 =<14> ANIMAL ("CHEVAL").
= 6 =<12> ANIMAL ("VACHE").
= 7 =<12> ANIMAL ("LAPIN").
= 8 =<16> ANIMAL ("PINTADE").
= 9 =<12> ANIMAL ("HIBOU").
= 10 =<20> ANIMAL ("BOUQUETIN").
= 11 =<14> ANIMAL ("CHÈVRE").

= 1 =<8> CONCAT ((*X0), *X1, (*X0; *X1)).
= 2 =<12> CONCAT ((*X0; *X1), *X2, (*X0; *X3)) :
          CONCAT (*X1, *X2, *X3).

= 1 =<18> MUTANT (*X0) :
          ANIMAL (*X1) &
          ANIMAL (*X2) &
          CONCAT (*X3, *X4, *X1) &
          CONCAT (*X4, *X5, *X2) &
          CONCAT (*X3, *X2, *X0).
```

Si on demande à l'interpréteur de nous donner toutes les solutions, la résolution est la suivante :

```
$?: MUTANT (*MUTANTS)
*MUTANTS = "ALLIGATORTUE"
*MUTANTS = "CARIBOURS"
*MUTANTS = "CARIBOUQUETIN"
*MUTANTS = "CHEVALLIGATOR"
*MUTANTS = "CHEVALAPIN"
*MUTANTS = "VACHEVAL"
*MUTANTS = "VACHÈVRE"
*MUTANTS = "LAPINTADE"
*MUTANTS = "HIBOURS"
*MUTANTS = "HIBOUQUETIN"
--SUCCÈS--(10)--
```

1.4 - LES TOURS DE HANOI

Les "Tours de Hanoï" est un jeu qui se joue avec trois piquets et une série de disques. Ces disques sont tous de taille différente et possédent un trou central permettant de les enfiler dans les piquets.

- Au départ, les disques sont tous sur le piquet de gauche, empilés les uns sur les autres, du plus grand au plus petit, pour former une sorte de pyramide.

● Le but du jeu est de les mettre tous sur le piquet du centre, en utilisant le piquet de droite comme piquet intermédiaire, selon les règles suivantes :

— à la fin, les disques du piquet central doivent avoir le même arrangement qu'au départ : empilés du plus grand au plus petit.

— on ne peut déplacer qu'un disque à la fois.

— seul le disque du sommet d'un piquet peut être déplacé.

— on ne peut poser un disque sur un autre disque qui est plus petit que lui.

● La stratégie qui permet de réussir ce jeu à tous les coups, avec trois piquets et un nombre quelconque N de disques, est fort simple. Afin de vous permettre d'épater vos amis, nous allons exposer ici les principes à suivre, et leur traduction en programme PROLOG :

— le jeu est terminé quand il n'y a plus de disque sur le piquet gauche initial. Ceci constitue la condition d'arrêt du programme.

— le jeu comprend effectivement trois étapes :

A. On déplace N-1 disques (tous sauf le dernier) du piquet initial du gauche vers le piquet de réserve (celui de droite), en utilisant le piquet final (celui du centre) comme intermédiaire. Ce coup est un coup récursif, puisqu'à chaque fois on utilise un seul disque.

B. On déplace un disque unique du piquet de gauche au piquet central, et on envoie un message exprimant le coup effectué.

C. La dernière étape consiste finalement à effectuer l'inverse de ce qui a été fait en a) : on déplace les N-1 disques du piquet de droite (de réserve) vers le piquet central, en utilisant le piquet de gauche comme intermédiaire.

● Le programme PROLOG correspondant utilise le prédicat HANOI (*N) qui imprime la succession de coups à effectuer avec *N disques sur le piquet de gauche :

```
HANOI (*N) : DÉPLACE (*N, GAUCHE, CENTRE, DROIT).
```

Le prédicat DÉPLACE utilise quatre arguments. Le premier correspond au nombre de disques à déplacer, les trois autres correspondent aux piquets source (GAUCHE), destination (CENTRE) et réserve (DROIT), utilisés pour déplacer ces disques. DÉPLACE se réfère à deux clauses, dont la première définit la condition d'arrêt et la deuxième l'appel récursif.

```
DÉPLACE (0, GAUCHE, CENTRE, DROIT) : CUT.
DÉPLACE (*N, *X1, *X2, *X3) : - (*N, 1, *R) &
                                DÉPLACE
                                (*R, *X1, *X3, *X2) &
                                INFO (*X1, *X2) &
                                DÉPLACE (*R, *X3, *X2, *X1).
```

A chaque fois qu'un disque est déplacé, on utilise le prédicat INFO, qui permet d'envoyer un message donnant les noms des piquets concernés :

```
INFO (*Y1, *Y2) : LINE &
                  PUT ('DISQUE DÉPLACE DU PIQUET', *Y1,
                  'AU PIQUET', *Y2) &
                  LINE.
```

- Le programme PROLOG est donc le suivant :

```
$LIST
= 1 =<7> HANOI (*X0) :
         DÉPLACE (*X0, GAUCHE, CENTRE, DROIT).

= 1 =<6> DÉPLACE (0, *X0, *X1, *X2) :
         CUT.

= 2 =<22> DÉPLACE (*X0, *X1, *X2, *X3) :
          - (*X0, 1, *X4) &
          DÉPLACE (*X4, *X1, *X3, *X2) &
          INFO (*X1, *X2) &
          DÉPLACE (*X4, *X3, *X2, *X1).

= 1 =<10> INFO (*X0, *X1) :
          LINE &
          PUT ('DISQUE DÉPLACE DU PIQUET', *X0, 'AU
          PIQUET', *X1) &
          LINE.
```

Avec trois disques, PROLOG utilisera la stratégie précédente pour la résolution et gagnera le jeu en sept coups :

```
$?: HANOI (3)
DISQUE DÉPLACE DU PIQUET GAUCHE AU PIQUET
CENTRE

DISQUE DÉPLACE DU PIQUET GAUCHE AU PIQUET DROIT

DISQUE DÉPLACE DU PIQUET CENTRE AU PIQUET DROIT

DISQUE DÉPLACE DU PIQUET GAUCHE AU PIQUET
CENTRE

DISQUE DÉPLACE DU PIQUET DROIT AU PIQUET GAUCHE

DISQUE DÉPLACE DU PIQUET DROIT AU PIQUET CENTRE

DISQUE DÉPLACE DU PIQUET GAUCHE AU PIQUET
CENTRE

--SUCCÈS--(1)--
```

1.5 - PROBLÈME DE TRAVERSÉE DE RIVIÈRE

Le problème est le suivant : un loup, un chou et une chèvre doivent traverser une rivière.

Pour cela, le fermier qui les accompagne dispose d'un bateau qui ne peut transporter que l'un d'entre eux à la fois.

De plus, en l'absence du fermier, il ne faut pas laisser sur une des rives l'un des couples suivants : le loup qui mangerait la chèvre, et la chèvre qui mangerait le chou. Le but de ce programme est de trouver la suite des traversées à effectuer.

Au départ, les trois individus sont placés sur la rive N (Nord). L'objectif est qu'ils se retrouvent tous sur la rive S (Sud), sans qu'aucun ne se soit fait manger par un autre...

Chaque traversée possède un *état* représenté par une liste (*X0, *Y0, *R) où :

— *X0 représente la liste des individus situés sur la rive N ;

— *Y0 représente la liste des individus situés sur la rive S ;

— *R représente la rive sur laquelle se trouve le bateau (donc également le fermier).

● Au départ, on lance TRANSPOR avec l'état initial (LOUP, CHOU, CHÈVRE), NIL, N).

● Pour effectuer une traversée de la rive *R1 à la rive *R2, par une action *TRAV, il faut s'assurer des points suivants :

— on choisit un individu *IND sur la rive *R1 ;

— on examine (par la clause ADMIS) que la liste des individus restant sur la rive *R1 est permise, c'est-à-dire qu'aucun des couples (LOUP, CHÈVRE), (CHOU, CHÈVRE) n'est laissé sur cette rive ;

— on ajoute *IND aux individus de la rive *R2 ;

— si aucun individu ne peut traverser, le bateau traversera à vide (cas particulier de CHOISIR).

● On vérifie que l'état proposé n'a pas déjà été rencontré, sinon on tournerait en rond (EXISTE). Pour cela, on mémorise chaque traversée par un couple (*R, *X1) (position du bateau, liste des individus sur la rive N), dans une liste *ÉTAT.

Lors d'un essai de traversée, il suffira de s'assurer que le couple (*R, *X1) n'existe pas dans la liste des états passés (*ÉTAT). Dans ce cas, on rangera ce couple dans liste *ÉTAT et on relance TRANSPOR.

● La traversée est terminée lorsqu'il n'y a plus d'individus sur la rive N. On imprime alors la succession des traversées effectuées.

```
PARCOURS :
    LINE&
    PUT ('DÉROULEMENT DE LA TRAVERSÉE') &
    LINE &
    TRANSPOR(((LOUP, CHOU, CHÈVRE), NIL, N),
    ((N, (LOUP, CHOU, CHÈVRE))), *PARCOURS) &
    ÉCRIRE (*PARCOURS) &
    CUT.

TRANSPOR((NIL, *YO, *R), *LISTE, NIL).
TRANSPOR(*Z, *ÉTAT, (*TRAV; *PARCOURS)) :
    TRAVERS(*Z, (*X1, *Y1, *R), *TRAV) &
    NOT (EXISTE ((*R, *X1), *ÉTAT)) &
    TRANSPOR((*X1, *Y1, *R), ((*R, *X1); *ÉTAT), *PARCOURS).

TRAVERS ((*X0, *Y0, N), (*X1, *Y1, S), TRAV (*IND, N, S)) :
    CHOISIR (*IND, *X0, *X1) &
    ADMIS (*X1) &
    AJOUTER (*IND, *Y0, *Y1).
TRAVERS ((*X0, *Y0, S), (*X1, *Y1, N), TRAV (*IND, S, N)) :
    CHOISIR (*IND, *Y0, *Y1) &
    ADMIS (*Y1) &
    AJOUTER (*IND, *X0, *X1).

CHOISIR (*IND, (*IND; *Q), *Q).
CHOISIR (*IND, (*X1; *X2), (*X1; *X3)) :
    CHOISIR (*IND, *X2, *X3).
CHOISIR (VIDE, NIL, NIL).

ADMIS (*X) :
    DIFF ((LOUP, CHÈVRE), *X) &
    CUT &
    DIFF ((CHOU, CHÈVRE), *X).

AJOUTER (VIDE, *X, *X).
AJOUTER (*X0, NIL, (*X0)) :
    <> (*X0, VIDE).
AJOUTER (LOUP, (*X0), (LOUP, *X0)).
AJOUTER (CHOU, (LOUP), (LOUP, CHOU)).
AJOUTER (CHOU, (CHÉVRE), (CHOU, CHÈVRE)).
AJOUTER (CHOU, (*X0, *X1), (*X0, CHOU, *X1)).
AJOUTER (CHÈVRE, (*X0), (*X0, CHÉVRE)).
AJOUTER (CHÈVRE, (*X0, *X1), (*X0, *X1, CHÈVRE)).

EXISTE (*X0, (*X1; *X2)) :
    EQ (*X0, *X1) &
    CUT.
EXISTE (*X0, (*X1; *X2)) :
    EXISTE (*X0, *X2).

ÉCRIRE ((*TRAV 1; *TRAVSUIT)) :
    PUT (*TRAV 1) &
    LINE &
    ÉCRIRE (*TRAVSUIT).

ÉCRIRE (NIL).
```

La demande de résolution nous donne le déroulement de la traversée.

```
$?: PARCOURS
DÉROULEMENT DE LA TRAVERSÉE
TRAV (CHÈVRE, N, S)
TRAV (VIDE, S, N)
TRAV (LOUP, N, S)
TRAV (CHÈVRE, S, N)
TRAV (CHOU, N, S)
TRAV (VIDE, S, N)
TRAV (CHÈVRE, N, S).

--SUCCÈS--(1)--
```

1.6 - Problème : SEND + MORE = MONEY ?

Ce problème énigmatique consiste à remplacer chaque lettre par un chiffre de telle sorte que l'addition suivante soit vérifiée.

```
     S  E  N  D
+    M  O  R  E
---------------
 M   O  N  E  Y
```

Pour cela, on doit considérer les retenues (éventuellement nulles) qui sont à ajouter à chaque colonne de l'opération :

```
R4  R3  R2  R1
     S   E   N   D
+    M   O   R   E
------------------
 M   O   N   E   Y
```

Chaque mot va être représenté par la liste de ses caractères. Pour trouver la solution, on procède de la façon suivante :

— on effectue l'opération de la droite vers la gauche ;

— chaque colonne de l'opération doit être admissible (ADMIS), c'est-à-dire que :

- les valeurs affectées aux lettres de l'opération (par N) sont différentes,
- on calcule l'addition des deux lettres et de la retenue précédente qui donne un résultat. Suivant la valeur de ce résultat, les clauses UNIDIZ calculent la valeur de la lettre résultante (unité) et de la retenue (dizaine). Si la lettre résultante a déjà été calculée, il faudra bien évidemment que ses valeurs soient identiques,

— la lettre M doit être différente de zéro et égale à la retenue R4 ;

— enfin, toutes les lettres doivent avoir des valeurs différentes (DIFFER).

Le programme se compose des faits N (permettant de donner une valeur à une lettre) et ÉGAL (entraînant l'unification de ses arguments), et des règles permettant d'aboutir à la solution :

```
N(0).
N(1).
N(2).
N(3).
N(4).
N(5).
N(6).
N(7).
N(8).
N(9).
ÉGAL (*X0, *X0).

SOL ((*S, *E, *N, *D), (*M, *O, *R, *E), (*M, *O, *N, *E, *Y)) :
ADMIS (*D, *E, 0, *Y, *R1) &
ADMIS (*N, *R, *R1, *E, *R2) &
ADMIS (*E, *O, *R2, *N, *R3) &
ADMIS (*S, *M, *R3, *O, *R4) &
<> (*M, 0) &
= (*M, *R4) &
DIFFER ((*S, *E, *N, *D, *M, *O, *R, *Y)) &
CUT.

ADMIS (*X0, *X1, *RET 1. *RESULTAT. *RET 2) :
N (*X0) &
N (*X1) &
<> (*X0, *X1) &
+ (*X0, *X1, *N1) &
+ (*N1, *RET1, *N2) &
UNIDIZ (*N2, *RESULTAT, *RET2).

UNIDIZ (*NOMBRE, *UNITÉ, *DIZAINE) :
>= (*NOMBRE, 10) &
— (*NOMBRE, 10, *UNITÉ) &
ÉGAL (*DIZAINE, 1) &
CUT.
UNIDIZ (*NOMBRE, *UNITÉ, *DIZAINE) :
ÉGAL (*UNITÉ, *NOMBRE) &
ÉGAL (*DIZAINE, 0).

DIFFER (NIL) : CUT.
DIFFER (*X0, (*X1, *X2)) : HORS (*X0, *X1) &
DIFFER (*X1).

HORS (*X0, NIL) : CUT.
HORS (*X0, (*X1; *X2)) : <> (*X0, *X1) &
HORS (*X0, *X2).
```

La résolution (temps d'exécution : environ 5 minutes) nous donne le résultat suivant :

```
$?: SOL (*SEND, *MORE, *MONEY)
*SEND = (9, 5, 6, 7)                          9567
*MORE = (1, 0, 8, 5)                        + 1085
*MONEY = (1, 0, 6, 5, 2) ..................-------
--SUCCES--(1)--                              10652
```

1.7 - DÉFINITION D'UN "MICRO-MONDE" DE ROBOTS

Le programme ci-dessous permet de définir et de commander le comportement de robots dans un "micro-univers" prédéfini. Ainsi, on peut demander à un robot d'aller chercher un objet qui se trouve dans un lieu précis, ou de le déposer quelque part.

Le "micro-univers" est défini par l'ensemble des clauses DANS (tel objet est dans tel lieu), PRIS (Tel objet peut être pris) et TENIR (tel individu tient tel objet, sachant qu'on ne peut tenir qu'un objet à la fois). Ces clauses constituent la base de faits du programme.

Cette base est bien évidemment évolutive, en fonction des actions qui sont effectuées et qui entraînent des changements d'état.

Les actions commandées aux robots sont définies par les clauses CHERCHER et METTRE, appelant les clauses ALLER, PRENDRE et MET.

● CHERCHER (*X0, *X1). Si l'objet ou robot *X0 ne détient pas déjà un objet, il va chercher l'objet *X1). Pour cela, il faut savoir où se trouve l'objet *X1, commander au robot d'aller dans ce lieu et de prendre cet objet.

● ALLER (*X0, *X1). On ordonne à l'objet ou robot *X0 d'aller dans le lieu *X1. Deux cas sont possibles, donc deux clauses sont définies pour cette action :

— soit le robot est déjà dans ce lieu. C'est la première clause, qui permet d'envoyer le message *X0 est dans *X1 ;

— soit le robot n'y est pas. On va chercher le lieu dans lequel il se trouve et le supprimer de la base des faits. Puis on envoie le message *X0 va dans *X1, et rajoute dans la base de faits la nouvelle clause spécifiant où se trouve le robot (DANS *X0, *X1)).

● PRENDRE (*X0, *X1). On ordonne au robot *X0 de prendre l'objet *X1. Pour cela, on s'assure que l'objet *X1 peut être pris (que l'on supprime ensuite de la base) et on rajoute dans la base le fait TENIR (*X0, *X1) (indiqué à l'exécution par le message *X0 prend *X1).

● METTRE (*X0, *X1, *X2). On ordonne au robot *X0 de mettre l'objet *X1 dans *X2.

Deux cas sont possibles:
*X0 tient déjà *X1,
*X0 doit aller chercher *X1.

Puis le robot dépose l'objet dans *X2 (entraînant la suppression des faits TENIR (*X0, *X1) et DANS (*ÉX1, *X3), et la création des faits PRIX (*X1) et DANS (*X1, *X2)).

Voyons plus en détail la composition du programme:

```
$LIST
=1=<3>DANS (CARL, ASTRONEF).  -> base de faits initiale
=2=<3>DANS (IGOR, HANGAR).
=3=<3>DANS (FUEL, HANGARD).
=4=<3>DANS (ARME, HANGAR).
=5=<3>DANS (TRÉSOR, CAVERNE).
=6=<3>DANS (ASTRONEF, ESPACE).

=1=<2>PRIS (FUEL).
=2=<2>PRIS (TRÉSOR).
=3=<2>PRIS (ARME).

=1=<21>CHERCHER (*X0, *X1)   -> définition des règles
          NOT (TENIR (*X0, *X2)) &
          DANS (*X1, *X3) &
          LINE &
          PUT (*X1, 'EST DANS', *X3) &
          ALLER (*X0, *X3) &
          PRENDRE (*X0, *X1).

=2=<18>ALLER (*X0, *X1) :
          PACK (DANS) &
          LINE&
          PUT (*X0, 'EST DANS', *X1) &
          CUT.

=2=<18>ALLER (*X0, *X1) :
          PACK (DANS) &
          SUPDANS (*X0, *X2) &
          ERASE &
          LINE &
          PUT (*X0, 'VA DANS', *X1) &
          APPEND (DANS (*X0, *X1)).

=1=<20>PRENDRE (*X0, *X1) :
          PRIX (*X1) &
          PACK (PRIS) &
          SUPPRIS (*X1) &
          ERASE &
          LINE &
          PUT (*X0, 'PREND', *X1) &
          APPEND (TENIR (*X0, *X1)) &
          CUT.
```

```
=1=<20>METTRE (*X0, *X1, *X2) :
         TENIR (*X0, *X1) &
         LINE&
         PUT (*X0, 'TIENT' *X1) &
         ALLER (*X0, *X2) &
         MET (*X0, *X1, *X2) &
         CUT.

=2=<15>METTRE (*X0, *X1, *X2) :
         CHERCHER (*X0, *X1) &
         ALLER (*X0, *X2) &
         MET (*X0, *X1, *X2) &
         CUT.

=1=<30>MET (*X0, *X1, *X2) :
         PACK (TENIR) &
         SUPTENIR (*X0, *X1) &
         ERASE &
         APPEND (PRIX (*X1)) &
         PACK (DANS) &
         SUPDANS (*X1, *X3) &
         ERASE &
         APPEND (DANS (*X1, *X2)) &
         LINE &
         PUT (*X0, 'A MIS' *X1, 'DANS', *X2).

=1=<9>SUPDANS (*X0, *X1) :
         CLAUSE (DANS (*X0, *X1), *X2) &
         CUT.

=2=<7>SUPDANS (*X0, *X1) :
         FORWARD &
         SUPDANS (*X0, *X1).

=1=<9>SUPTENIR (*X0, *X1) :
         CLAUSE (TENIR (*X0, *X1), *X2) &
         CUT.

=2=<7>SUPTENIR (*X0, *X1) :
         FORWARD &
         SUPTENIR (*X0, *X1).

=1=<7>SUPRIS (*X0) :
         CLAUSE (PRIS (*X0), *X1) &
         CUT.

=2=<5>SUPRIS (*X0) :
         FORWARD &
         SUPPRIS (*X0).
```

Nous avons nommé nos robots IGOR et CARL. Voyons désormais s'ils obéissent bien à nos ordres :

- Demandons à IGOR d'aller chercher du FUEL :

```
$?: CHERCHER (IGOR, FUEL)
FUEL EST DANS HANGARD
IGOR EST DANS HANGAR
IGOR PREND FUEL
--SUCCÈS--(1)--
```

La base de faits n'est pas modifiée, puisque les arguments encompte n'ont pas changé de lieu. Par contre, la base s'est enrichie d'un nouveau fait de prédicat TENIR :

```
$?: LIST (TENIR)
=1=<3>TENIR (IGOR, FUEL).
```

Le fait PRIS (FUEL) a été supprimé :

```
$LIST (PRIS)
=1=<2>PRIS (TRÉSOR).
=2=<2>PRIS (ARME).
```

- Demandons à CARL d'aller chercher le TRÉSOR :

```
$?: CHERCHER (CARL, TRÉSOR)
TRÉSOR EST DANS CAVERNE
CARL VA DANS CAVERNE
CARL PREND TRÉSOR
--SUCCÈS--(1)--
```

La base de faits a été modifiée (les modifications sont visualisées en caractère gras) : CARL est maintenant dans la CAVERNE :

```
$LIST (DANS)
=1=<3>DANS (IGOR, HANGAR).
=2=<3>DANS (FUEL, HANGAR).
=3=<3>DANS (ARME, HANGAR).
=4=<3>DANS (TRÉSOR, CAVERNE).
=5=<3>DANS (ASTRONEF, ESPACE).
=6=<3>DANS (CARL, CAVERNE).
```

Le sous-programme TENIR s'est enrichi d'une nouvelle clause :

```
$LIST (TENIR)
=1=<3>TENIR (IGOR, FUEL).
=2=<3>TENIR (CARL, TRÉSOR).
```

Le sous-programme PRIS ne contient plus qu'une clause :

```
$LIST (PRIS)
=1=<2>PRIS (ARME).
```

- Demandons à IGOR d'aller mettre du FUEL dans l'ASTRONEF :

```
$?: METTRE (IGOR, FUEL, ASTRONEF)
IGOR TIENT FUEL
IGOR VA DANS ASTRONEF
IGOR A MIS FUEL DANS ASTRONEF
--SUCCÈS--(1)--
```

La base des faits a été ainsi modifiée :

```
$LIST (DANS)
=1=<3>DANS (ARME, HANGAR).
=2=<3>DANS (TRÉSOR, CAVERNE).
=3=<3>DANS (ASTRONEF, ESPACE).
=4=<3>DANS (CARL, CAVERNE).
=5=<3>DANS (IGOR, ASTRONEF).
=6=<3>DANS (FUEL, ASTRONEF).
```

Le sous-programme TENIR ne contient plus qu'une clause :

```
$LIST (TENIR)
=1=<3>TENIR (CARL, TRÉSOR).
```

Le FUEL peut de nouveau être pris :

```
$LIST (PRIS)
=1=<3>PRIS (ARME).
=2=<2>PRIS (FUEL).
```

- Demandons à IGOR de mettre l'ARME dans la CAVERNE :

```
$?: METTRE (IGOR, ARME, CAVERNE)
ARME EST DANS HANGAR
IGOR VA DANS HANGAR
IGOR PREND ARME
IGOR VA DANS CAVERNE
IGOR A MIS ARME DANS CAVERNE
--SUCCÈS--(1)--
```

Les modifications apportées à la base des faits sont les suivantes (les sous-programmes TENIR et PRIS n'ont pas été modifiés) :

```
=1=<3>DANS (TRÉSOR, CAVERNE).
=2=<3>DANS (ASTRONEF, ESPACE).
=3=<3>DANS (CARL, CAVERNE).
=4=<3>DANS (FUEL, ASTRONEF).
=5=<3>DANS (IGOR, CAVERNE).
=6=<3>DANS (ARME, CAVERNE).
```

1.8 - ANALYSE DU LANGAGE

Nous vous proposons ci-dessous un programme permettant à partir d'une grammaire élémentaire disposant d'un vocabulaire limité, d'analyser simplement une phrase.

La grammaire est composée d'un ensemble de règles de réécriture, permettant de vérifier qu'une phrase est syntaxiquement correcte (P : phrase, GN : groupe nominal, GV : groupe verbal) :

```
P        -> GN GV
GN       -> art nom
GN       -> art adj nom
GV       -> verbe GN
art      -> le, la, un, une, les, des, ...
adj      -> petite, belle, gros, ...
nom      -> chat, souris, gruyère, garçon, fille, ...
verbe    -> mange, regarde...
```

Cette grammaire permet de produire et d'analyser des phrases du type : le gros chat mange une souris, ou le garçon regarde la belle fille.

Les phrases à analyser sont données sous forme de liste, dont les termes sont les mots ordonnés de celles-ci, permettant une analyse "mot à mot".

Les prédicats grammaticaux (P, GN, GV) possèdent trois arguments :

— le premier est une liste de termes définissant la règle de réécriture du prédicat ;

— le deuxième correspond à la phrase, ou partie de phrase, à analyser, donnée sous forme de liste des mots qui la composent ;

— le troisième est une liste correspondant au reste de la phrase, c'est-à-dire à la partie non analysée (lorsque la phrase est analysée intégralement, le résultat est une variable libre).

Les faits portant sur le vocabulaire (ART, NOM, VERBE) possèdent également trois arguments :

— le premier identifie le mot correspondant ;

— le deuxième porte sur la phrase, ou partie de phrase à traiter ;

— le troisième donne la liste restante, c'est-à-dire la liste initiale privée du mot que l'on vient d'analyser (tête de la liste).

```
$LIST
=1=<18>P ((GN (*X0), GV (*X1), *X2, *X3))
       GN (*X0, *X2, *X2) &
       GV (*X1, *X4, *X3).

=1=<19>GN ((ART (*X0), NOM (*X1)), *X2, *X3)
       ART (*X0, *X2, *X4) &
       NOM (*X1, *X4, *X3) &
       CUT.

=2=<26>GN ((ART (*X0), ADJ (*X1), NOM (*X2)), *X3, *X4) :
       ART (*X0, *X0, *X5) &
       ADJ (*X1, *X5, *X5) &
       NOM (*X2, *X6, *X4) &
       CUT.
```

```
=1=<18>GV ((VERBE (*X0), GN (*X1)), *X2, *X3) :
        VERBE (*X0, *X2, *X4) &
        GN (*X1, *X4, *X5).

=1=<6>ART (LE, (LE; *X0), *X0).
=2=<6>ART (LA; *X0), *X0).
=3=<6>ART (UN; *X0, *X0).
=4=<6>ART (UNE; *X0), *X0).
=5=<6>ART (LES; *X0), *X0).
=6=<6>ART (DES; *X0), *X0).

=1=<6>ADJ (PETITE, (PETITE; *X0), *X0).
=2=<6>ADJ (BELLE, (BELLE; *X0), *X0).
=3=<6>ADJ (GROS, (GROS; *X0), *X0).

=1=<6>NOM (CHAT, (CHAT; *X0), *X0).
=2=<6>NOM (SOURIS, (SOURIS; *X0), *X0).
=3=<6>NOM (GRUYÈRE, (GRUYÈRE; *X0), *X0).
=4=<6>NOM (GRUYÈRE, (GARÇON; *X0), *X0).
=5=<6>NOM (FILLE, (FILLE; *X0), *X0).

=1=<6<VERBE (MANGE, (MANGE; *X0), *X0).
=2=<6>VERBE (REGARDE. (REGARDE; *X0), *X0).
```

Les demandes de résolution et dessous permettent d'analyser les phrases suivantes :

```
$?: ART (*X, (LE, GROS, CHAT, MANGE), *Y)
*X = LE
*Y = (GROS, CHAT, MANGE)
--SUCCÈS--(1)--

$?: GN (*X, (LE, GROS, CHAT, MANGE), *Y)
*X = (ART (LE), ADJ (GROS), NOM (CHAT))

*Y = (MANGE)
--SUCCÈS--(1)--
```

Si on inclut dans le programme les clauses suivantes (on ne peut demander directement les résolutions correspondantes en mode commande, par manque de place) :

```
P1(*X, *Y) : P (*X, (LE, GARÇON, REGARDE, LA PETITE,
FILLE), *Y).
P (*X, *Y) : P (*X, (LA, PETITE, SOURIS, REGARDE, LE,
GRUYERE), *Y).
P3 (*X, *Y) : P (*X, (LE, GROS, CHAT, MANGE, LA, PETITE,
SOURIS), *Y).
```

On obtient :

```
$?: P1 (*X, *Y)
*X = (GN ((ART (LE), NOM (GARÇON))), GV
((VERBE (REGARDE), GN (ART, (LA), ADJ (PETITE)
NOM (FILLE))))))
*Y = *1
--SUCCÈS--(1)--
```

```
$?: P2 (*X, *Y)
*X = (GN ((ART (LA), ADJ (PETITE), NOM (SOURIS))),
GV ((VERBE (REGARDE), GN (ART (LE), NOM (GRUYERE)))))
*Y = *1
--SUCCES--(1)--

$?: P3 (*X. *Y)
*X = (GN ((ART (LE), ADJ (GROS), NOM (CHAT))), GV
(VERBE (MANGE), GN (ART (LA), ADJ (PETITE, NOM
(SOURIS)))))
*Y = *1
--SUCCÈS--(1)--
```

2 - LEXIQUE

Architecture
L'architecture interne d'un ordinateur correspond à l'ensemble des éléments qui composent l'unité centrale de l'ordinateur.

Algorithme
Un algorithme est une méthode de résolution, consistant à décrire pas à pas le cheminement à effectuer pour aboutir à la solution d'un problème. Transcrit dans un certain langage de programmation, un algorithme donne lieu à un programme.

Base de connaissances
Partie d'un système expert, contenant les faits et relations valides dans le domaine d'expertise. Elle est donc extérieure au système expert proprement dit : le système expert est le véritable programme qui interprète ces connaissances.

Base de données
Ensemble de faits bruts et élémentaires d'un domaine. Elles correspondent à des vérités du domaine.

Bit
Le bit est l'unité élémentaire d'information, utilisé par l'ordinateur. Un bit peut contenir un seul des deux symboles utilisés dans le code binaire : "0" ou "1".

Code binaire
Le code binaire permet de représenter des nombres en base 2. On utilise seulement deux symboles, notés généralement 0 et 1. Ce code est compréhensible par l'ordinateur.

Le plus grand nombre que l'on puisse écrire en binaire dépend du nombre de bits utilisés : plus il y a de bits, plus on peut écrire de nombres. Le bit dit de "poids faible" (le plus à droite) donne le nombre d'unités, le deuxième bit le nombre de paires d'unité, le troisième bit le nombre de paquets de quatre unités, etc.

Exemples :
01 correspond au nombre 1 écrit en décimal.
10 correspond au nombre 2 (une paire d'unité)
11 correspond au nombre 3 (une unité + une paire d'unité).
11 correspond au nombre 3 (une unité + une paire d'unité).

Ainsi :
Avec 2 bits, on peut écrire le nombre de 0 (00) à 3 (11).
Avec 3 bits, on peut écrire le nombre de 0 (000) à 8 (111).
Avec 6 bits, on peut écrire le nombre de 0 (000000) à 63 (111111).
Avec 10 bits, on peut écrire le nombre de 0 à 1023, etc.

Cogniticien

Nouveau terme permettant de décrire un nouveau métier de l'informatique. Le cogniticien, ou ingénieur de la connaissance, est chargé de collecter les connaissances, stratégies et "trucs" utilisés par une expert humain pour aboutir rapidement et sûrement au bon diagnostic. Son rôle est ensuite de les "traduire" en système expert.

Compilation

Voir Langages.

Hexadécimal

Mode de représentation des nombres en base 16. Comme le code hexadécimal exige l'utilisation de 16 symboles, on a ajouté aux 10 chiffres (0 à 9), les lettres suivantes : A (pour 10), B (pour 11), C (pour 12), D (pour 13), E (pour 14), F (pour 15).

Langages

En informatique, le mot "langage" désigne un ensemble composé d'une part de caractères et de symboles (ou vocabulaire) et d'autres part de règles de syntaxe (ou grammaire) qui permettent à l'utilisateur de "dialoguer" avec l'ordinateur, en vue de lui faire exécuter un ensemble d'instructions (ou programme). Les langages informatiques sont actuellement très nombreux, et nous n'en donnerons pas ici, bien sûr, une liste exhaustive (ce qui est impossible). Cependant, on peut regrouper ces langages en trois grandes catégories : les langages machine et assembleur, les langages compilés et les langages interprétés.

Les langages machine et assembleur

- Le langage machine est, en fait, le seul langage que peut "comprendre" un ordinateur, car c'est celui qui en est le plus proche. Ainsi quel que soit le langage de programmation utilisé, un programme devra toujours être traduit, à un moment ou à un autre (et de façon plus ou moins automatique et transparente pour l'utilisateur), en langage machine, avant d'être exécuté.

Le vocabulaire du langage machine est réduit à une liste figée de codes d'instruction (codée sous forme de 0 et de 1) et sa seule règle de syntaxe consiste à définir la longueur associée à chaque type d'instructions.

Les codes d'instruction correspondent aux instructions élémentaires utilisables par l'ordinateur.

- Le langage assembleur est très proche du langage machine. Les codes d'instruction sont remplacés par un terme minémonique, rendant la lecture des programmes plus aisée.

- Ces langages sont très efficaces (car très proche de l'ordinateur), mais les programmes correspondants sont très longs à écrire et peut lisibles. De plus, ces langages sont propres à chaque modèle d'ordinateurs, puisque le jeu d'instructions qu'ils utilisent en dépend. Ainsi, on ne pourra réutiliser un programme machine ou assembleur sur un ordinateur de type différent.

Les langages compilés

Ils possèdent, au contraire, un vocabulaire et une syntaxe indépendants de l'ordinateur utilisé. Ils ont été définis à partir des opérations et des fonctions les plus couramment utilisées, pour un type d'applications donné. De ce fait leur syntaxe est plus riche et permet de commander des opérations plus complexes dans une seule ligne de programme.

- Avant d'être exécuté, un programme, écrit dans un de ces langages, doit d'abord être *compilé*, c'est-à-dire traduit dans le langage spécifique de l'ordinateur. Cette traduction est réalisée par un programme particulier, propre à chaque machine et à chaque langage *le compilateur*. Au cours de cette phase, certaines erreurs de programmation (en particulier des erreurs de syntaxe) peuvent être détectées par le compilateur.

- Le résultat de la compilation est une suite d'instructions écrites en langage machine, réalisant toutes les opérations indiquées dans le programme d'origine (ou programme source).

- En général, cette suite d'instructions doit être "assemblée", avant d'être exécutable, avec d'autres sous-programmes qui ont été écrits et compilés séparément. Cet assemblage est réalisé par *l'éditeur de liens*.

- Un programme écrit en langage compilé sera très facile et clair à écrire. Cependant, sa mise au point est relativement longue, puisqu'elle s'affectue en trois phases ; chaque erreur détectée par le compilateur nécessite sa correction, une nouvelle compilation (en espérant qu'il n'y ait pas d'autres erreurs !), l'édition des liens et la vérification pendant l'exécution.

Exemples :

FORTRAN fut le premier langage compilé (destiné aux calculs numériques et scientifiques), suivi, entre autres, de langages plus spécifiques (COBOL pour la gestion, ALGOL pour le calcul).

Les langages interprétés

Ils possèdent, comme pour les langages compilés, un vocabulaire et une syntaxe très riches, indépendants de l'ordinateur utilisé. Mais, cette fois-ci, c'est pendant son exécution que chaque ligne du programme est traduite et exécutée par l'interpréteur. On élimine ainsi les phases de compilation et d'édition des liens : la mise au point des programmes est beaucoup plus rapide. Par contre, l'exécution sera évidemment ralentie par le temps de traduction et de vérification du programme.

● Certains langages existent sous forme compilée et interprétée. Le BASIC est le plus connu des langages interprétés et sa facilité d'emploi est légendaire.

● Le cadre de l'Intelligence Artificielle a permis le développement et l'introduction de nombreux langages interprétés : par exemple LISP, et pour le cas qui nous concerne PROLOG. Ces deux langages peuvent aussi être "compilés".

Moteur d'inférence
Voir Structure de contrôle.

Octet
L'octet est la quantité d'information définie par 8 bits.
Un octet peut contenir un nombre compris entre 0 et 255.
Un kilooctet (abréviation Ko contient 1204 octets).

Pile
Une pile est une organisation particulière de la mémoire qui fonctionne par empilements successifs de chaque nouvel élément à stocker.

Processeur
Voir Unité centrale.

Programme
Voir Langages.

Programme source
État du programme lors de sa saisie, et avant toute transformation (compilation, interprétation, exécution).

Règles de production
Couples "situation-action" ou "si-alors".

Représentation symbolique

Abstraction permettant de représenter des objets concrets (et non plus simplement des nombres).

Structure de contrôle

Stratégie de raisonnement permettant d'utiliser les connaissances d'un domaine, pour résoudre des problèmes.

Unité centrale

L'unité centrale est la partie "pilote" de l'ordinateur. Elle gère la mémoire, échange des informations avec les périphériques, effectue des calculs, etc.

Un ordinateur complet comprendra, outre son unité centrale, des périphériques, des mémoires et des circuits d'alimentation.

3 - INDEX

* 110, 150
\+ 110, 143, 172
— 110, 154, 172
< 111, 125, 145
> 96, 111, 145
=< 111, 145
=> 111, 145, 172
= 111, 127, 145, 172
< > 111, 145, 172
; 92
. ..... 36, 64, 97
: 36, 77, 97
& 19, 71
$ 47
?: 60, 99, 152
= = ... 48
abandon d'une résolution 60, 145, 155
arbre 87, 98
arithmétique 109
argument 27, 64, 87, 96
APPEND 121, 143
ATOM 112, 143
atome 84, 95, 112
BACKWARD 120, 149
base de données 16, 64, 159,
base de faits 18, 175
BOTTOM 119, 143
but 19, 60, 66, 99, 131
calcul des prédicats 26
caractère 84, 112
caractères de contrôle 94, 199
CH 48, 56, 144
chaîne de caractères 93, 164
CHAR 112, 144
clause 34, 82, 95, 96, 100
clause courante 119
CLAUSE 121, 144
CLAUSES 50, 103
CLEAR 48, 102, 144
CLS 48, 144
cohérence 78. 85, 98, 129
coïncidence 101
commande 46, 109
comparaison 111, 145

condition d'arrêt ... 101, 137
conjonction de buts ... 71, 76, 97
connecteur ET (∧) ... 28, 34
connecteur NON (¬) ... 28, 33
connecteur OU (v) ... 29, 34
connecteur => ... 29, 32
connecteur <=> ... 30
CONSTANT ... 112, 145
constante ... 26, 84, 94, 111, 112
constante de skolem ... 33
constructeur de liste "." ... 90
corps de clause ... 97
corps de règle ... 77
correspondance ... 66, 95, 132
CUT ... 118, 130, 141, 145
DEL ... 48, 107, 145
démonstration par l'absurde ... 38
DIC ... 52, 103, 116, 146
DICO ... 50, 103
DIFF ... 115, 146
DIR ... 61, 146
DIV ... 110, 146
DOFF ... 48, 127, 146
DON ... 48, 127, 147
échec d'un but ... 19, 67, 135
ED ... 48, 55, 147
effet de bord ... 109
EQ ... 114, 147
ERASE ... 122, 147
fait ... 18, 64, 95, 96
FORMAT ... 61, 148
forme clausale ... 32, 36
FORWARD ... 120, 130, 148
GET ... 125, 148
GETC ... 127, 148
hexadécimal ... 94, 182
Horn (clause de) ... 39
INSERT ... 121, 148, 149
instanciation ... 68, 86, 132
INTEGER ... 112, 149
KILL ... 62, 149
LINE ... 94, 112, 125, 130, 149
liste ... 25, 89, 177
liste de termes ... 89, 91
liste vide ... 90
LISP ... 25, 90
LIST ... 49, 130, 149
littéral ... 33
LOAD ... 62, 149
logique ... 26, 40

MOD 110, 150
mode commande 47, 54
mode éditeur 54
Modus Ponens 18, 33
moteur d'inférence 18
NŒUDS 50, 55
NOT 117, 150, 170, 171, 174
nombre entier 84, 109, 112
objet 26, 84
PACK 120, 130, 151
paquet de clauses 95, 100, 120,132
PAUSE 127, 151
pile 50, 103
PILE 50, 55, 85
prédicat 64, 84, 86, 95
prédicat prédéfini ou primitive 95, 108
Principe de Résolution 25, 37
PRINT 122, 130, 151
PRINTON 49, 124, 151
PRINTOFF 49, 124, 152
PROLOG 25, 40, 43
proposition atomique 27
proposition composée 28
PUT 94, 112, 124, 152
quantificateur 30, 33, 34
question 60, 66, 199
queue d'une liste 90, 92
récursivité 101, 166
règle 18, 77, 95, 97
règles d'inférence 31
resatisfaire des buts 70, 73, 80, 101, 135
résolvante 60, 76, 86, 99, 131, 152
retour arrière 74, 77, 81, 118, 129, 134, 137, 161
satisfaire des buts 69, 72, 79, 101, 132, 135
SAVE 61, 152
SIZE 52, 62, 153
sous-programme 95, 100, 120, 134
STAT 49, 103, 154
STOP 58
structure 26, 86, 112
STRUCT 112, 155
symbole fonctionnel 27
symbole de prédicat 27
syntaxe 43, 83
terme 26, 83
terme composé 27, 86, 112
terme mémoire 52, 83, 84, 85, 103
TERMES 50, 103
tête de clause 97
tête de liste 90, 92

tête de règle 77
texte 93, 162
TEXTE 50, 103
TOP 119, 130, 155
TROFF 49, 128, 155
TRON 49, 128, 138, 155
unification 37, 81, 86, 114, 132
VALUE 116, 155
variable 26, 68, 85, 86, 97
variable libre 69, 86, 112, 115
VAR 112, 156
LOAD 62, 149

4 - LISTE DE MESSAGES D'ERREURS

● On distingue plusieurs types d'erreurs :
— des erreurs dans la formulation des clauses du programme détectées lors de l'analyse syntaxique,
— des erreurs relatives à l'état de la mémoire à un moment donné,
— des erreurs d'Entrée/Sortie, mettant en cause les périphériques utilisés,
— des erreurs rencontrées lors de l'exécution d'un programme.

● Lorsque l'interpréteur rencontre une erreur, il vous envoie un message du type :

ERR X : <nom-du-type-d'erreur>

— le <nom-du-type-d'erreur> vous renvoit à l'un des types exposés ci-dessus, selon les conventions suivantes :
SYNTAXE : erreur détectée dans l'analyse syntaxique.
MÉMOIRE : erreur dûe à un débordement d'une zone mémoire.
E/S : erreur d'Entrée/Sortie sur cassette, disquette ou imprimante.
EXÉCUTION : erreur détectée lors de l'exécution d'un programme ou d'un prédicat prédéfini,
— 'X' est un nombre qui spécifie plus particulièrement l'erreur d'un type donné, en vous renvoyant au numéro correspondant dans la liste suivante.

● Lorsqu'une erreur est détectée sous le mode éditeur, le message s'affiche en dessous de la fenêtre de l'éditeur et l'éditeur est automatiquement rappelé : le curseur est positionné sur la première ligne de texte où a été constatée une erreur — non loin de la faute —et attend votre correction.

Remarque :
Les erreurs SOFT sont des erreurs système, il faut appuyer sur le bouton d'initialisation pour en sortir.

4.1 - ERREURS DÉTECTÉES DANS L'ANALYSE SYNTAXIQUE (SYNTAXE)

ERR 1
Mauvaise syntaxe de la clause : un des symboles ".", "à" ou "&" est attendu.

ERR 2
Mauvaise syntaxe de la clause : les arguments d'un prédicat sont mal gérés : le symbole")" est attendu.

ERR 3
Nombre incorrect d'arguments pour un prédicat prédéfini.

ERR 4
Un caractère utilisé est interdit.

ERR 5
L'argument utilisé en paramètre d'un prédicat prédéfini est interdit.

ERR 6
Le terme de type "texte" est trop long, ou mal écrit (il manque une apostrophe pour le délimiter).

ERR 7
Le nombre de paramètres, ou arguments, utilisés dans la définition d'unprédicat est trop grand. On rappelle que le nombre maximum d'arguments pour un prédicat est 15.

ERR 8
La syntaxe du nom d'une variable utilisée est incorrecte. On rappelle qu'une variable commence par le symbole "*" suivi d'une séquence de huit caractères au plus, donnant son nom.

ERR 9
Le nombre entier utilisé dépasse les limites autorisées (la valeur absolue maximale d'un nombre entier est limitée à 32767).

ERR 10
La clause à analyser contient trop de termes mémoire. On rappelle qu'une clause peut utiliser au maximum 255 termes mémoire (correspondant à des termes élémentaires de 4 octets).

ERR 11
L'utilisation de la commande, spécifiée dans la queue de la clause, est interdite ici.

ERR 12
Un prédicat prédéfini apparaît dans une tête de clause : ceci est interdit, vous ne pouvez redéfinir les primitives connues de l'interpréteur.

ERR 13
Une erreur est constatée dans la syntaxe d'une liste : il manque une parenthèse, ou la tête et la queue de la liste sont mal gérées,...

ERR 14

Le terme "nul" est interdit.
Cette erreur se produit, en particulier, lorsque l'utilisateur répond par **ENTREE** (donc ne tape rien) à l'exécution du prédicat prédéfini GET.

ERR 15

Le prédicat de tête de clause est mal défini.

4.2 - ERREURS DE DÉBORDEMENT D'UNE ZONE MÉMOIRE (MÉMOIRE)

Ces erreurs sont dues à une mauvaise gestion, une mauvaise évaluation de la taille du programme à saisir. Rappelez-vous que la taille mémoire initiale disponible est visualisable par la commande STAT. Elle peut être modifiée, selon vos besoins, par la commande SIZE (N1, N2, N3, N4, N5), *avant* la saisie du programme (elle a également pour effet de détruire le programme en mémoire) !

ERR 1

Impossible de stocker le texte : la zone de stockage des textes est pleine.

ERR 2

Le dictionnaire est plein : impossible de stocker tout nouveau nom d'atomes ou de prédicats.

ERR 3

La pile des clauses est pleine : impossible de stocker toute nouvelle clause.

ERR 4

Impossible de stocker les termes de la clause : la pile des termes est pleine.

ERR 5

Impossible de terminer la résolution : la pile des nœuds de résolution est pleine. Cela veut peut-être dire que votre programme "boucle".

ERR 6 et ERR 7

La pile des substitutions, utilisée lors de la résolution est pleine.

ERR 8

Le buffer, ou zone temporaire de travail, de l'éditeur (zones mémoire NŒUDS et PILE utilisées normalement lors de la résolution) est trop petit pour éditer le texte source de la clause.

Dans ce cas, on vous demande de sortir de l'éditeur (par la touche **STOP**), mais le travail effectué n'est pas perdu. Par prudence, il est conseillé d'éditer les paquets de clauses importants par petits morceaux.

Il se peut aussi que la commande de l'éditeur soit inaccessible, si l'éditeur est considéré comme plein. Il faut alors augmenter la mémoire des piles (par la commande SIZE (N1, N2, N3, N4, N5)).

Une taille minimum de 3 Ko (12* 256 octets) est souhaitable pour les zones PILE et NŒUDS.

ERR 9

Impossible de construire la clause : les arguments "frères" (de même niveau) d'un des prédicats sont trop loin l'un de l'autre. Cela veut dire que l'un d'entre eux représente des structures trop importantes en mémoire (plus de 127 termes) pour pouvoir permettre la construction des liaisons entre arguments de la clause. La clause correspondante est détruite. Le même problème peut se poser entre deux prédicats voisins, dans la queue d'une clause.

ERR 10

Un élément du dictionnaire (nom d'atome ou de prédicat défini par l'utilisateur), apparaît trop souvent dans le programme : le nombre maximal d'occurences d'un élément dans le dictionnaire est de 250.

ERR 11

La place mémoire demandée par SIZE (N1, N2, N3, N4, N5) est trop importante par rapport à la capacité mémoire de l'ordinateur.

ERR 12

Il n'y a pas assez de mémoire allouée au buffer des symboles (limitée à 1 Ko), lors de l'analyse syntaxique.

ERR 13

La demande de buffer mémoire, lors d'une résolution, est refusée (plus de place pour l'exécution de GET).

ERR 14

Lors de la construction des termes, il y a débordement de la pile Système.

De plus, une structure ne doit pas dépasser une certaine "profondeur". Pour une liste (y compris les chaînes de caractères) la limite maximum est de 40 éléments.

ERR 15

Dépassement du nombre d'appels possibles au prédicat prédéfini GET au cours d'une même résolution : la limite maximale est fixée à 40 appels à GET lors d'une résolution.

4.3 - LES ERREURS D'ENTRÉE/SORTIE SUR CASSETTE, DISQUETTE OU IMPRIMANTE (E/S)

ERR 1
Le code de l'opération est erroné : opération inexistante ou interdite.

ERR 2
Disquette ou cassette non prête.

ERR 3
Disquette endommangée ou lecteur de disquette inexistant.

ERR 4
Fichier inexistant.

ERR 5
Disquette pleine.

ERR 9
Erreur système non reconnue ! (lecteur inexistant ou autre...).

ERR 10
Il manque le nom du fichier en Entrée/Sortie (en particulier lors d'une sauvegarde, où le nom du fichier est obligatoire).

ERR 11
Erreur de syntaxe sur le nom de fichier utilisé.

ERR 12
Erreur d'accès imprimante.

ERR 13
Buffer plein au cours d'une sauvegarde : relancez la sauvegarde.

4.4 - ERREURS DÉTECTÉES LORS DE L'EXÉCUTION D'UN PROGRAMME OU PRÉDICAT (EXECUTION)

ERR 1
Impossible de traiter le prédicat prédéfini arithmétique : les paramètres utilisés ne sont pas des nombres.

ERR 2
La clause référencée n'existe pas.

ERR 3

Demande d'affichage d'une chaîne "libre" : impossible! (erreur système).

ERR 4

L'un des arguments du prédicat prédéfini VALUE (<source>, <destination>) est incorrect.

ERR 5

L'un des arguments ou opérande d'un prédicat prédéfini de comparaison, ou relation d'ordre sur des constantes, est incorrect.

ERR 6

La clause courante est inexistante. Cela empêche en particulier le bon fonctionnement de certains prédicats prédéfinis de manipulation de clauses par programme.

ERR 7

L'alternative est inexistante pour une règle : le backtrack ou retour arrière est impossible. L'alternative présente initialement a été détruite par le prédicat prédéfini ERASE (erreur système).

ERR 8

Ce message signifie que vous avez demandé, par appui sur la touche **RAZ**, l'arrêt de la résolution alors que celle-ci contenait le prédicat prédéfini PAUSE (entraînant donc son échec).

ERR 9

La clause à traiter est libre : elle a été détruite au cours de la résolution par le prédicat prédéfini ERASE.

ERR 10

L'argument utilisé dans le prédicat doit être une variable libre.

ERR 11

Erreur portant sur le premier argument, <terme-tête>, des prédicats prédéfinis de manipulation de clauses par programme APPEND ou INSERT.

L'argument <terme-tête> ne correspond pas au nom d'une tête de clause connue.

ERR 12

Erreur portant sur le deuxième argument <liste-queue> (s'il apparaît explicitement dans la formulation) des prédicats prédéfinis de manipulation de clauses par programme APPEND ou INSERT.

L'argument <liste-queue> n'est pas une liste de prédicats.

ERR 13

Essais de division par 0. La division par zéro est impossible, la résolution est abandonnée.

5 - LISTE DES CARACTÈRES DE CONTROLE

PROLOG permet d'utiliser dans le prédicat PUT, des caractères de contrôle, sous forme hexadécimale, permettant de formater l'édition (sans utilisation de la commande ou prédicat PRINTON).

Les caractères utilisables et leurs effets sont les suivants :

CODE	NOM	EFFET
$07	**BELL**	Bip sonore
$08	**BS**	Recul arrière : déplace le curseur d'une position vers la gauche, à partir de sa position courante
$09	**HT**	Tabulation horizontale : déplace le curseur d'une position vers la droite, à partir de sa position courante
$0A	**LF**	Saut de ligne : fait "descendre" le curseur d'une ligne, à partir de sa position courante
$0B	**VT**	Tabulation verticale : fait "remonter" le curseur d'une ligne, à partir de sa position courante : on écrit sur la ligne précédente
$0C	**FF**	Effacement de l'écran et positionnement du curseur dans le coin supérieur gauche de l'écran
$0D	**CR**	Retour à la ligne : positionne le curseur en début de ligne
$11	**DC1**	Allume le curseur
$14	**DC4**	Éteint le curseur pendant l'exécution
$18	**CAN**	Efface la fin de la ligne, à partir de la position du curseur
$1E	**RS**	Positionnement du curseur en haut de page

BIBLIOGRAPHIE

J.-P. AUBERT et R. SCHOMBERG
"Pratiquez l'Intelligence Artificielle"
Éditions Eyrolles (1984)

A. BONNET
"L'intelligence artificielle : Promesses et réalités"
InterÉditions (1984)

K.L. CLARK et F.G. McCABE
"Micro-Prolog" : Programmer en logique"
Éditions Eyrolles (1985)

W.F. CLOCKSIN et C.S. MELLISH
"Programmer en PROLOG"
Éditions Eyrolles (1984)

A. COLMERAUER
"PROLOG, langage de l'intelligence artificielle"
In La Recherche n° 158 (septembre 1984)

GROUPE INTELLIGENCE ARTIFICIELLE LUMIGNY :
"PROLOG"
InterÉditions (1985)

R. KOWALSKI
"Logic for Problem Solving"
Éditions North Holland (1979)

N. NILSSON
"Principles of Artificial Intelligence"
Éditions Springer-Verlag (1980)

S. ROBERTS
"Artificial Intelligence"
In Mini-Micro Systems (décembre 1983)