La naissance de Prolog

Alain Colmerauer et Philippe Roussel

Juillet 1992

Table des matières

Avant-propos

Prolog est né d'un projet, dont le but n'était pas de faire un langage de programmation mais de traiter les langages naturels, en l'occurrence le français. Ce projet a donné naissance à un Prolog préliminaire à la fin 1971 et un Prolog plus définitif à la fin de l'année 1972. Cet article relate l'histoire de ce projet, décrit en détail la version préliminaire de Prolog puis sa version définitive. Les auteurs ont aussi jugé bon de décrire les systèmes-Q un langage qui a joué un rôle important dans la genèse de Prolog.

Nous avons aussi ajouté un appendice avec six exemples de programmes pour se familiser avec le Prolog actuel. Nous sommes en 2014.

Introduction

Il est de notoriété publique que le nom de Prolog a été créé à Marseille en 1972 [1]. C'est Philippe Roussel qui l'a choisi, comme abréviation de « PROgrammation en LOGique », pour désigner l'outil informatique conçu pour implanter un système de communication homme machine en langage naturel. On peut dire que Prolog a été le fruit d'un mariage réussi entre le traitement du langage naturel et la démonstration automatique. L'utilisation directe du français pour raisonner et dialoguer avec un ordinateur, était un rêve un peu fou : c'était le projet élaboré dès l'été 70, par Alain Colmerauer qui avait une certaine expérience dans le traitement informatique des langages naturels, et qui souhaitait élargir sa recherche.

Les deux auteurs de cet article, Alain Colmerauer et Philippe Roussel, sont donc présentés, mais il est évident que bien d'autres personnes ont participé à un tel projet. Pour rester objectifs en racontant l'histoire de la naissance de Prolog, qui a déjà vingt ans maintenant, nous avons repris tous les documents qui nous restaient et nous avons joué les historiens. Nous avons d'abord suivi la chronologie de très près, pour exposer les faits et décrire les acteurs de l'été 70 à la fin 76. Ceci constitue la première partie de l'article. Les autres parties sont plus techniques. Elles sont consacrées aux trois langages de programmation qui se sont succédés rapidement : les systèmes-Q conçus pour la traduction automatique, le Prolog préliminaire créé en même temps que son application et le Prolog définitif créé indépendamment de toute application.

Ce papier n'est pas le premier sur l'histoire de Prolog. Signalons celui de Jacques Cohen [9], directement sur Prolog, et celui de Robert Kowalski [28], sur la naissance de la discipline « Logic Programming ». Il est aussi intéressant de prendre connaissance de l'histoire de la démonstration automatique vue par Donald Loveland [29] et de l'existence antérieure d'un possible concurrent à Prolog,

1. Il existe une version anglaise de ce papier dans un livre : *History of Programming Languages,* par Thomas J. Bergin et Richard G. Gibson, Addison-WesleyPublishing Company, 1996

le langage Absys vu par Elcock [17].

Chapitre 1

La chronologie des faits

Au début de juillet 1970, Robert Pasero et Philippe arrivent à Montréal. Ils sont invités par Alain, alors professeur assistant d'informatique à l'Université de Montréal et responsable du projet de traduction automatique TAUM. Tous sont à un tournant de leurs carrières. Robert et Philippe ont 25 ans et viennent juste d'être nommés assistants (en informatique) à la nouvelle Faculté des sciences de Luminy. Alain a 29 ans et s'apprête à rentrer en France après un séjour de 3 ans au Canada.

Durant leurs deux mois de séjour à Montréal, Robert et Philippe se familiarisent avec le traitement informatique des langues naturelles, ils programment des analyseur context-free non-déterministes en Algol 60 et un générateur de paraphrases françaises avec les systèmes-Q, le langage de programmation qu'Alain avait développé pour le projet de traduction (voir deuxième partie).

Parallèlement Jean Trudel, chercheur canadien, inscrit en thèse de doctorat avec Alain, poursuit ses travaux sur la démonstration automatique. Son article de référence est celui de Alan Robinson [37] sur le principe de résolution. C'est un article difficile à comprendre, mais Jean a l'avantage d'avoir suivi un cours de logique avec Martin Davis à New York. Il a maintenant déjà écrit un démonstrateur complet dans lequel l'unification est programmée dans un style moderne : tous les calculs consistent à modifier des pointeurs.

1.1 Année 1971, les premiers travaux

Tout le monde se retrouve à Marseille au début de l'année 1971. Alain a obtenu un poste de maître de conférence en informatique et Jean Trudel a pu le suivre grâce à une bourse d'Hydro-Québec de deux ans. Le projet est donc de faire des déductions sur des textes écrits en français. Le travail se répartit comme suit :

Jean Trudel et Philippe s'occupent de la partie déduction, Robert Pasero et Alain de la partie langue naturelle.

Nous avons accès à la machine du centre de calcul de l'université de Marseille, un IBM 360-44 dans nos locaux, doté de 900Ko environ de mémoire centrale et muni d'un système d'exploitation sans mémoire virtuelle. Grâce à un moniteur interactif que Jean Trudel réalise, et en utilisant la machine de nuit pour bénéficier de toute la mémoire disponible, l'équipe peut travailler dans des conditions exceptionnelles pour l'époque en France : quasiment 1 Mo de mémoire pour exécuter les programmes et une conception interactive des communications entre utilisateurs et programmes à travers la console de l'opérateur.

Jean Trudel améliore son démonstrateur puis, à partir du mois de mai, Philippe en produit toute une série écrits en Algol-W. Un système très naïf de communications en langue naturelle est alors réalisé par toute l'équipe [12]. Les interfaces entre les formules logiques et le français consistent en 50 règles de systèmes-Q en entrée et 17 règles de système-Q en sortie. La partie raisonnement est réalisée par l'un des démonstrateurs de Philippe. Il est alors possible d'avoir la conversation

suivante avec l'ordinateur :

Utilisateur
Les chats tuent les souris.
Minou est un chat qui n'aime pas les souris qui mangent du fromage.
Miquette est une souris qui mange du fromage.
Max n'est pas une souris.
Que fait Minou ?
Machine
Le Minou ne aime pas les souris qui mangent le fromage.
Le Minou tue les souris.
Utilisateur
Qui est un chat ?
Machine
Le Minou.
Utilisateur
Qu'est-ce que mange Miquette ?
Machine
Le fromage.
Utilisateur
Qui n'aime pas les souris qui mangent du fromage ?
Machine
Le Minou.
Utilisateur
Qu'est-ce que mange Minou ?
Machine
Ce que mange les chats qui ne aiment pas les souris qui mangent le fromage.

Les formules logiques créées font intervenir : des constantes qui représentent des éléments,
Minou, *Miquette*, *Max*, *LeFromage* ;
des constantes qui représentent des ensembles,
LesChats, *LesSouris*, *LesSourisQuiMangentDuFromage*,
LesChatsQuiNaimentPasLesQuiMangentDuFromage ;
des constantes qui représentent des relations binaires portant sur des ensembles,
Tuent, *NaimePas*, *Mange* ;
un symbole fonctionnel d'arité 1 et deux symboles relationnels d'arité 2 et 3,
Le, *Dans*, *Vrai*.

Un terme de la forme $Le(a)$ représente l'ensemble constitué du seul élément a. Une formule de la forme $Dans(x, y)$ exprime que l'ensemble x est inclus dans l'ensemble y et une formule de la forme $Vrai(r, x, y)$ exprime que les ensembles x et y sont dans la relation r. Aux clauses qui codent les phrases, Jean Trudel ajoute quatre clauses liant les trois symboles *Le*, *Dans*, *Vrai* :

$(\forall x)[Dans(x, x)]$,
$(\forall x)(\forall y)(\forall z)[Dans(x, y) \wedge Dans(y, z) \Rightarrow Dans(x, z)]$,
$(\forall a)(\forall b)[Dans(Le(a), Le(b)) \Rightarrow Dans(Le(b), Le(a))]$,
$(\forall x)(\forall y)(\forall r)(\forall x')(\forall y')[Vrai(r, x, y) \wedge Dans(x, x') \wedge Dans(y, y')] \Rightarrow Vrai(r, x', y')$.

Le problème majeur est d'éviter une production intempestive d'inférences due aux axiomes de transitivité et de réflexivité de la relation d'inclusion *Dans*.

En continuant ses recherches sur la démonstration automatique Jean Trudel tombe sur une méthode très intéressante, la SL-résolution [25]. Il nous convainc d'inviter l'un de ses créateurs, Robert Kowalski. Celui-ci nous rend visite en juin 1971 et reste une semaine. La rencontre reste inoubliable. Pour la première fois nous discutons avec un spécialiste de démonstration automatique qui nous fait saisir ce qu'est vraiment le principe de résolution, ses variantes et ses raffinements. Quant à Robert Kowalski il rencontre des gens passionnés par ses recherches et décidés à les mettre en application dans le traitement des langues naturelles.

Avec Jean Trudel nous revoyons Bob Kowalski à l'occasion d'un congrès IJCAI en septembre 71 à Londres, où nous assistons à un exposé de Terry Winograd [42]. sur un interface en langue naturelle qui nous rend perplexe. C'est à cette occasion que nous prenons connaissance de l'existence du langage de programmation Planner de Carl Hewitt [21]. Le manque de formalisation de ce langage, notre méconnaissance de Lisp et surtout le fait que nous tenons à tout prix à la logique, tout cela fait que ces travaux eurent peu d'influence sur la suite de nos recherches.

1.2 Année 1972, l'application qui crée Prolog

L'année 72 est la plus fertile. Tout d'abord en février l'équipe obtient une subvention de 122 000FF, (à l'époque environ 20 000$), de l'Institut de Recherche d'Informatique et d'Automatique, une institution dépendant du ministère français de l'industrie et cela pour une durée de 18 mois. Ce contrat permet d'acquérir un terminal *teletype* (30 caractères par seconde) et de le connecter sur l'IBM 360-67 (doté du merveilleux système d'exploitation CP-CMS qui gérait des machines

virtuelles) de l'Université de Grenoble au moyen d'une ligne spécialisée de 300 bauds. Ce fut de loin le moyen de calcul le plus agréable dont l'équipe disposa durant les trois années qui suivirent et tout le monde l'utilisa, y compris les nombreux chercheurs qui nous rendirent visite. Nous avons mis plusieurs années à éponger la dette en heures-machines que nous avons ainsi accumulée à Grenoble. Le contrat permet enfin d'engager une secrétaire et un chercheur, Henry Kanoui, étudiant de DEA qui s'occupera de la morphologie du français. De son coté Robert Kowalski obtient un financement de l'OTAN qui financera de nombreux échanges entre Edinburgh et Marseille.

De tous les systèmes de résolution qui furent implantés par Philippe, « SL-résolution » de R. Kowalski et D. Kuehner semble le plus intéressant. Son fonctionnement en pile similaire à la gestion des appels de procédures dans un langage de programmation classique est particulièrement adapté à un traitement du non déterminisme par « backtracking » à la Robert Floyd [19] plutôt que par recopie et sauvegarde des résolvantes. SL-résolution servira donc de support à la thèse de Philippe qui porte sur le traitement de l'égalité formelle en démonstration automatique [35]. L'égalité formelle est moins expressive que l'égalité classique mais par contre peut être traitée efficacement. La thèse de Philippe débouchera sur l'introduction du prédicat *dif* (pour $\neq$) dans la toute première version de Prolog.

Nous invitons à nouveau Robert Kowalski, mais cette fois ci pour une période plus longue, avril et mai. Ensemble nous avons tous maintenant une connaissance plus calculatoire de la démonstration automatique. Nous savons axiomatiser de petits problèmes (addition de nombres entiers, concaténation de listes, retournement de listes etc.) de façon à ce qu'un démonstrateur de type SL-résolution calcule rapidement le résultat. Mais le paradigme clause de Horn nous est inconnu et Alain ne voit toujours pas comment se passer des systèmes-Q pour tout ce qui concerne l'analyse des langues naturelles.

Après le départ de Robert Kowalski, Alain trouve enfin une façon de réaliser des analyseurs puissants. A chaque symbole non-terminal N de la grammaire il associe un prédicat binaire $N(x, y)$ signifiant que x et y sont des mots terminaux pour lesquels le mot u défini par $x = uy$ existe et est dérivable de N. En représentant x et y par des listes, chaque règle de la grammaire peut alors être codée par une clause ayant exactement le même nombre de littéraux que d'occurrences de symboles non-terminaux. Il n'est jamais nécessaire de faire appel à la concaténation de liste. (Cette technique est connue maintenant sous le nom de technique de différence de listes.) De plus, dans chaque non-terminal Alain introduit des para-

mètres supplémentaires pour propager et calculer des informations. Comme dans les systèmes-Q, l'analyseur non seulement vérifie que la phrase est correcte mais extrait une formule représentant l'information contenue dans celle-ci. Plus rien ne s'oppose alors à faire un système de communication homme machine entièrement en « logique ».

Une décision draconienne est prise : quitte à être incomplets nous choisissons une résolution linéaire avec unification uniquement entre les têtes de clauses. Sans le savoir nous avons découvert une stratégie qui est complète lorsque l'on se limite aux clauses de Horn. Robert Kowalski [26] montrera ce point et, avec Maarten van Emden, donnera la sémantique moderne par point fixe de la programmation avec clauses de Horn [27].

Pendant l'automne 1972 le premier système Prolog est implanté par Philippe en Algol-W, le langage de Niklaus Wirth. Parallèlement Alain et Robert Pasero réalisent le système de communication homme-machine en français tant désiré [13]. Une interaction constante a lieu entre Philippe qui implante Prolog et Alain et Robert Pasero qui programment dans un langage qui se crée au fur et à mesure. Cette version préliminaire de Prolog est décrite en détails dans la troisième partie de ce papier. C'est à cette époque qu'est choisi également le nom définitif du langage, nom suggéré par Jacqueline, l'épouse de Philippe, à partir de mots clés qui lui avaient été fournis.

Le système de communication homme-machine est le premier programme Prolog conséquent qui fut jamais écrit [13]. Il comporte 610 clauses. Alain en a écrit 334, essentiellement la partie analyse, Robert Pasero 162, la partie purement déductive, et Henry Kanoui a écrit en 104 clauses une morphologie du français qui permet de faire le lien entre le singulier et le pluriel de tous les noms communs et de tous les verbes, même irréguliers, à la troisième personne du présent. Voici un exemple de texte soumis au système de communication homme machine et les

réponses en 1972,

Utilisateur
  Tout psychiatre est une personne.
  Chaque personne qu'il analyse est malade.
  Jacques est un psychiatre à Marseille.
  Est-ce que Jacques est une personne ?
  Où est Jacques ?
  Est-ce que Jacques est malade ?
Machine
  Oui.
  A Marseille.
  Je ne sais pas.

Le texte original suivi des trois réponses était en fait,

Utilisateur

```
TOUT PSYCHIATRE EST UNE PERSONNE.
CHAQUE PERSONNE QU'IL ANALYSE, EST MALADE.
*JACQUES EST UN PSYCHIATRE A *MARSEILLE.
EST-CE QUE *JACQUES EST UNE PERSONNE?
OU EST *JACQUES?
EST-CE QUE *JACQUES EST MALADE?
```

Machine

```
OUI.
A MARSEILLE.
JE NE SAIS PAS.
```

Toutes les inférences sont faites à partir des pronoms (il, ils, elle, ...), des articles (le, les, un, tout, ...) et des fonctions sujet et complément avec ou sans prépositions (de, à, ...). En fait le système ne connaissait que les pronoms, les articles et les prépositions (le vocabulaire était codé en 164 clauses), il reconnaissait les noms propres par l'astérisque obligatoire qui devaient les précéder et les verbes et les noms communs par les 104 clauses de morphologie générale du français.

Pendant le mois de novembre nous entreprenons, avec Robert Pasero, une vaste tournée des laboratoires de recherche américains après une visite d'Edinburgh. Nous emportons un rapport préliminaire sur notre système de communication en langue naturelle et sur notre tout premier Prolog. Nous laisserons des copies de ce rapport un peu partout. Jacques Cohen nous accueille à Boston et nous

introduit au MIT. Nous y sommes très bien accueillis et nous discutons avec Minsky, Charniak, Hewitt et Winograd. Nous rendons aussi visite à Woods à BBN. Nous allons ensuite à Stanford, visitons le SRI, le laboratoire d'IA de John McCarthy, rencontrons Cordell Green, exposons nos travaux devant J. Feldman très critique et passons Thanks Giving chez Robert Floyd.

1.3 Année 1973, le Prolog définitif

Au début de l'année 73, plus exactement en avril, notre groupe acquiert une existence officielle. Le CNRS le reconnaît comme « équipe de recherche associée » avec l'intitulé « Dialogue homme-machine en langue naturelle » et le dote d'un budget de 39 000F (environ 6500$) pour la première année. Cette somme doit être comparée avec les 31 6880F (environ 50 000$) que nous obtenons en octobre de l'IRIA pour renouveler le contrat « Communication homme-machine en langue naturelle avec déductions automatiques » pour une durée de 2 ans et demi.

Les utilisateurs de la version préliminaire de Prolog au sein du laboratoire ont maintenant suffisamment programmé pour que leur expérience serve de base à une deuxième version de Prolog, version résolument orientée vers un langage de programmation et non pas simplement vers un système déductif plus ou moins automatisé. Outre le système de communication en français de 1972, deux autres applications sont réalisées à l'aide de cette version initiale de Prolog : un système de calcul formel [3, 4, 23] et un système général de résolution de problèmes appelé *Sugiton* [22]. D'autre part Robert Pasero continue à l'utiliser dans le cadre de son travail sur la sémantique du français qu'il conclut en soutenant une thèse au mois de mai [33].

De février à avril 73, Philippe fait un séjour à la School of Artificial Intelligence de l'Université d'Edimbourg au sein du Département de Logique Computationnelle dirigée par Bernard Meltzer, et où Robert Kowalski l'avait invité. Outre les discussions nombreuses qu'il a avec lui et David Warren, il rencontre également Robert Boyer et Jay Moore qui avaient bâti une implantation de la résolution selon un procédé extrêmement ingénieux basé sur une technique de partage de structures pour représenter les formules logiques engendrées au cours d'une déduction. Le résultat de ce séjour et la nécessité pour le laboratoire de disposer d'un véritable langage de programmation nous décident à jeter les bases d'un deuxième Prolog.

De mai à juin 1973, nous fixons les grandes lignes du langage, en particulier les

choix de la syntaxe, des primitives de base et des principes de calcul de l'interprète, le tout allant dans le sens d'une simplification par rapport à la version initiale. De juin à la fin de l'année, Gérard Battani, Henry Méloni et René Bazzoli, alors étudiants de DEA, réalisent l'interprète en Fortran et son superviseur en Prolog.

Ainsi que le lecteur pourra en juger dans la quatrième partie de ce papier, ce nouveau Prolog, qui y est décrit en détails, introduit toutes les fonctionnalités essentielles des Prolog actuels. Signalons au passage que c'est à cette occasion que disparaît « l'occur check » jugé trop coûteux.

1.4 Année 1974 et 1975, la diffusion de Prolog

La version interactive de Prolog qui fonctionne sur Grenoble à l'aide d'une console *teletype* est très sollicitée. David Warren qui est chez nous pendant le mois de janvier et de février 1974 l'utilise pour écrire son système de générateur de plans *Warplan* [39]. A cette occasion il fait le commentaire suivant sur le système Prolog :

« The present system is implemented partly in Fortran, partly in Prolog itself, and running on an IBM 360-67 achieves roughly 200 unifications per second. »

Henry Kanoui et Marc Bergman l'utilisent pour réaliser un système de manipulation formelle d'une certaine envergure appelé Sycophante [5, 24]. Gérard Battani et Henry Meloni l'utilisent pour réaliser un système de reconnaissance de la parole permettant de poser des questions sur le mode d'emploi du système d'exploitation CP-CMS de l'IBM de Grenoble [2, 31]. L'interface entre le signal acoustique et la suite de phonèmes qu'il représente est empruntée au CNET à Lannion et bien entendu n'est pas écrite en Prolog.

Au début de l'année 1975 Alain a complètement réécrit le superviseur en gardant les déclarations d'opérateurs infixés en Prolog mais en lui adjoignant un compilateur de grammaires dites de « métamorphoses ». Cette fois-ci, contrairement à René Bazzoli, il utilise un analyseur « top-down » pour lire les règles Prolog. Il s'agit d'un bon exercice de méta-programmation. Plus tard David Warren inclura de telles règles de grammaires dans sa version compilée de Prolog [40] et, avec Fernando Pereira, rebaptisera « definite clauses grammars » une variante simplifiée des grammaires de métamorphoses [34]. Les grammaires de métamorphoses permettent d'écrire directement des règles de grammaire paramétrées comme on avait coutume de les écrire en système-Q. Le superviseur compile ces règles en des clauses Prolog efficaces en ajoutant deux paramètres supplémen-

taires. Pour montrer l'efficacité et l'expressivité des grammaires de métamorphoses Alain écrit un petit compilateur modèle d'un langage style Algol vers un langage machine imaginaire et un système complet de dialogue homme-machine en français avec déductions automatiques. Tout ce travail avec une justification théorique des grammaires de métamorphose est publié dans [14].

Gérard Battani et Henry Meloni sont très sollicités pour diffuser Prolog. Il l'envoient à Budapest, Varsovie, Toronto, Waterloo (Canada) et se déplacent à Edinburgh pour aider David Warren à mieux l'installer sur un PDP 10. Hélène Le Gloan une ex-étudiante l'installe à l'Université de Montréal. Michel Van Caneghem l'installe à l'IRIA à Paris avant de venir travailler chez nous. Enfin Maurice Bruynooghe après un séjour de trois mois à Marseille (octobre à décembre 75) l'emporte à Leuven (Belgique).

En fait, comme nous l'a fait remarquer David Warren, Prolog s'est peut être surtout diffusé par l'intermédiaire de personnes qui s'y sont intéressées et qui en ont pris une copie soit directement de Marseille, soit d'un site intermédiaire comme Edinburgh. Ainsi Prolog n'a pas tellement été distribué, il s'est « échappé » et s'est « multiplié ».

Pendant l'année 1975, toute l'équipe réalise un portage de l'interprète sur un mini-ordinateur de 16 bits : le T1600 de la compagnie française Télémécanique. La machine ne dispose que de 64K octets et il faut écrire une gestion de mémoire virtuelle de toute pièce. Pierre Basso s'en occupe et gagne aussi le concours de la séquence d'instructions la plus courte qui effectue un adressage sur 32 bits tout en testant le défaut de page. Chaque membre du laboratoire reçoit alors deux pages de Fortran à traduire en langage machine. On rassemble tous les bouts de programme traduits et cela marche ! Après 5 ans nous disposons enfin d'une machine bien à nous sur laquelle notre cher Prolog fonctionne lentement, mais fonctionne.

Chapitre 2

Un ancêtre de Prolog, les systèmes-Q

L'histoire de la naissance de Prolog s'arrête donc à la fin de l'année 1975. Passons maintenant à des aspects plus techniques et tout d'abord décrivons les systèmes-Q, le résultat d'un premier pari : développer un langage de programmation de très haut niveau, même si les temps d'exécution qu'il implique puissent sembler effarants [11]. Ce pari et l'expérience acquise dans l'implantation des systèmes-Q furent déterminants dans le deuxième pari : Prolog.

2.1 Unification unidirectionnelle

Un système-Q consiste en un ensemble de règles de réécriture portant sur des suites de symboles complexes séparés les uns des autres par le signe +. Chaque règle est de la forme

$e_1 + e_2 + \cdots + e_m \rightarrow f_1 + f_2 + \cdots + f_n$

et signifie : dans la suite de symboles que l'on est en train de manipuler on peut remplacer toute sous-suite de la forme $e_1 e_2 \cdots e_m$ par la sous-suite de la forme $f_1 f_2 \cdots f_n$. Les e_i et les f_i sont des expressions parenthésées représentant des symboles complexes, en fait des arbres. Ces expressions ressemblent étrangement à des termes Prolog, mais font intervenir trois types de variables qui se terminent toutes par une astérisque : suivant que la variable commence par une lettre de l'ensemble $\{a, b, c, d, e, f\}$, $\{i, j, k, l, m, n\}$ ou $\{u, v, w, x, y, z\}$ elle désigne une étiquette, un arbre, ou une suite (éventuellement vide) d'arbres séparés par des virgules. Par exemple la règle

```
P + A*(X*,I*,Y*) -> I* + A*(X*,Y*)
```

appliquée sur la suite

```
P + Q(R,S,T) + P
```

produit trois suites possibles

```
R + Q(S,T) + P,
S + Q(R,T) + P,
T + Q(R,S) + P.
```

La notion d'unification est donc déjà présente, mais elle est unidirectionnelle : les variables apparaissent dans les règles mais jamais dans la suite d'arbres que l'on est en train de transformer. Par contre l'unification prend en compte l'associativité de la concaténation et, comme dans l'exemple précédent, peut produire plusieurs résultats.

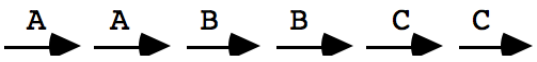
2.2 Stratégie d'application des règles

Ceci est pour la partie unification. En ce qui concerne la stratégie d'application des règles Alain [11] écrit :

« Utiliser un ordinateur pour analyser une phrase est une entreprise difficile. Le problème principal est de nature combinatoire : pris isolément chaque groupe d'éléments de la phrase peut se combiner de différentes façons avec d'autres groupes et en former de nouveaux qui à leur tour peuvent se recombiner, et ainsi de suite. En général il n'existe qu'une seule façon de regrouper la totalité des éléments, mais pour la découvrir, il faut essayer tous les groupements possibles. Pour représenter d'une façon économique cette multitude de regroupements, j'utilise un graphe orienté où chaque flèche est surmontée d'une expression parenthésée représentant un arbre. Un système-Q n'est rien d'autre qu'un ensemble de règles permettant de transformer un tel graphe en un autre graphe. Cette transformation peut correspondre à une analyse, à une synthèse de phrase, ou à une manipulation formelle de ce genre. »

Par exemple la suite

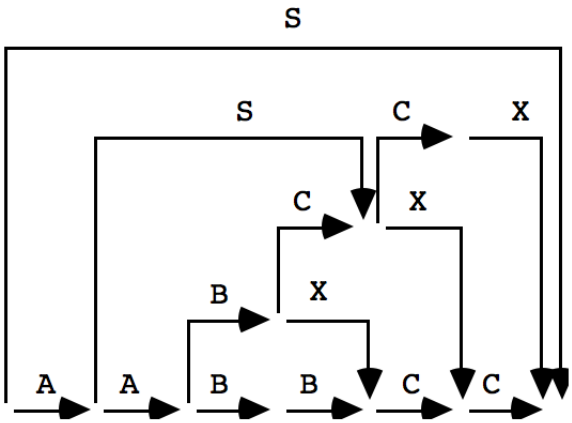
```
A + A + B + B + C + C
```



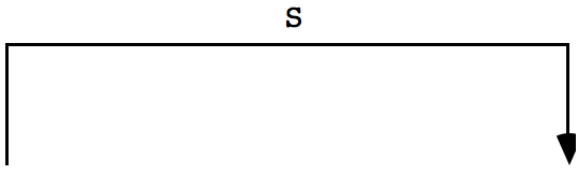
et l'application des 4 règles

```
A + B + C -> S
A + S + X + C -> S
X + C -> C + X
B + B -> B + X
```

produit le graphe



On conserve tous les chemins qui vont du point d'entrée au point de sortie et qui ne comportent pas de flèches ayant été utilisées pour produire d'autres flèches. On conserve donc l'unique flèche



c'est-à-dire la suite réduite au seul symbole s.

Cette façon de procéder est relativement efficace car elle préserve le maximum de parties communes entre toutes les suites. Un autre aspect des systèmes-Q est la possibilité de les enchaîner les uns à la suite des autres. Chacun prend comme donnée le graphe résultant du précédent. Cette technique était largement utilisée dans le projet de traduction automatique, puisqu'une phrase anglaise ne subissait pas moins de 15 systèmes-Q avant d'être traduite en français. Deux systèmes-Q s'occupaient de la morphologie, un autre de l'analyse de l'anglais, deux du passage d'une structure anglaise à un structure française, un de la synthèse du français et 9 de la morphologie française [38].

Signalons le coté réversible des système-Q. Le signe de réécriture que nous avons noté `->` était en fait noté `==` et, suivant l'option choisie et spécifiée en tête de programme, était interprété comme une réécriture de gauche à droite ou une réécriture de droite à gauche. Ceci veut dire que le même programme pouvait être utilisé pour décrire une transformation et la transformation inverse, comme l'analyse et la synthèse d'une phrase.

Il est intéressant de noter que contrairement aux analyseurs écrits en Prolog qui utilisent une stratégie descendante (top-down), les analyseurs écrits en systèmes-

Q utilisent une stratégie remontante (bottom-up). En fait Alain avait une longue expérience de ce type de stratégie. Sa thèse passée à Grenoble (en France) portait déjà sur des analyseurs remontants mais fonctionnant par backtracking [10]. Le non-déterminisme était alors limité par des relations de précédences très proches de celles de Robert Floyd [18]. De plus, juste avant de concevoir les systèmes-Q et toujours dans le cadre du projet de traduction automatique, Alain avait écrit un analyseur et un synthétiseur général pour W-grammaire, le formalisme introduit par A. van Wijngardeen pour décrire Algol 68 [41]. Là aussi un analyseur remontant était utilisé pour retrouver la structure des symboles complexes (définis par une méta-grammaire) ainsi qu'un deuxième analyseur remontant pour l'analyse du texte proprement dite [7].

2.3 Réalisation

Les systèmes-Q furent écrits en Algol par Alain et furent opérationnels dès octobre 1969. Michel van Caneghem et François Stellin, terminant alors une maîtrise d'informatique, en produirent une version Fortran et Gilles Steward une version ultra rapide en langage machine pour l'ordinateur CDC 6400 de l'Université de Montréal.

Ces systèmes-Q furent utilisés pour construire une chaîne complète de traduction automatique anglais-français par toute l'équipe du projet TAUM : Brian Harris écrivit la morphologie de l'anglais, Richard Kittredge écrivit une importante grammaire pour l'analyse de l'anglais, Gilles Stewart écrivit la phase de transfert, Jules Danserau écrivit la grammaire pour la synthèse du français et Michel van Caneghem produisit une morpholgie complète du français [38]. Les systèmes-Q furent aussi utilisés quelques années plus tard pour écrire le système METEO dont une version industrielle actuelle traduit tous les jours les bulletins météorologiques canadiens d'anglais en français.

Chapitre 3

Le Prolog préliminaire

Passons maintenant à la version préliminaire de Prolog qui fut créée au cours de l'écriture du système de communication homme machine en automne 1972 [13].

Rappelons que l'objectif fixé était extrêmement ambitieux : disposer d'un outil nous permettant de traiter les problèmes syntaxiques et sémantiques de la langue naturelle, en considérant la logique du premier ordre non seulement comme un langage de programmation, mais aussi comme un langage de représentation des connaissances. Autrement dit, la formulation logique devait servir non seulement pour les différents modules du système de communication mais aussi pour les informations échangées y compris avec l'utilisateur.

Le choix s'étant porté sur la logique du premier ordre (sous forme clausale) et non pas sur une logique d'ordre supérieure, il y avait là une apparente impossibilité à vouloir que des programmes puissent manipuler d'autres programmes. C'est bien sûr par le biais de mécanismes extra-logiques que nous avons résolu ce problème. En fait, cette version initiale de Prolog fut conçue plutôt comme l'outil d'une l'application que comme un langage de programmation universel. Les prémisses d'un tel langage étaient cependant là.

3.1 Choix de la méthode de résolution

La décision concernant le choix du système logique et le mécanisme d'inférence de base que nous allions adopter était crucial pour le succès du projet. Si le principe de résolution de A. Robinson s'imposait naturellement par la simplicité de la forme clausale, l'unicité de la règle d'inférence et certaines similitudes avec l'appel de procédures des langages classiques, il fut difficile de décider quel type d'adaptation était nécessaire pour répondre à nos objectifs. Il fallait prendre en compte la validité et la complétude logique du système, les problèmes d'implantation en

machine et surtout les risques d'explosion combinatoire que nous connaissions parfaitement après nos expérimentations.

Parmi les systèmes de questions-réponses et les techniques de résolution de problèmes que nous avions étudiés, figuraient entre autres celui de D. Luckham [30] et N.J. Nilson, celui de J.L. Darlington [15] et celui de Cordell Green [20]. Cette étude, les essais de Jean Trudel et Robert Pasero qui utilisaient les versions expérimentales des démonstrateurs de Philippe, les recherches d'Alain sur la formulation logique des grammaires ainsi que les nombreuses discussions avec Robert Kowalski, tout cela nous amena à considérer le principe de résolution de A. Robinson selon un point de vue différent de celui qui prévalait alors. Plutôt que de vouloir démontrer un théorème par l'absurde, nous voulions calculer un ensemble « intéressant » de clauses déductibles d'un ensemble donné de clauses. De tels ensembles constituant dans notre esprit des programmes, nous avions ainsi des programmes engendrant d'autres programmes. C'est cette idée qui fut constamment sous-jacente dans la conception de cette version préliminaire du langage comme dans la réalisation de l'application.

Le choix final porta sur une adaptation de la méthode de résolution proche de la philosophie générale du Prolog que l'on connaît maintenant, mais comportant cependant des éléments originaux. Chaque exécution s'effectuait avec un ensemble de clauses constituant le « programme » et un ensemble de clauses constituant les « questions », le tout produisant un ensemble de clauses constituant les « réponses ». Les littéraux des clauses étaient ordonnés de gauche à droite, l'unification s'effectuait entre le littéral de tête de la résolvante et celui d'une des clauses du programme. L'originalité résidait dans le fait que dans chaque clause une partie des littéraux, séparée par le signe « / », n'étaient pas traités au cours de l'exécution, mais étaient accumulés pour produire une des clauses réponses en fin de déduction. En outre, certains prédicats (tel *dif*) étaient traités par une évaluation retardée et pouvaient également être transmis en réponse. Enfin, il fut décidé que le non-déterminisme serait traité par backtracking, ce qui voulait dire qu'une seule branche de l'arbre de recherche résidait à un instant donné en mémoire.

Formellement, la méthode de déduction choisie peut se décrire selon les trois règles ci-dessous, où « question », « clause choisie » et « réponse » sont trois clauses prises respectivement dans les ensembles « questions », « programme » et « réponses » tandis que « résolvante » est une clause intermédiaire.

Règle d'initialisation de la déduction

$$\frac{\text{question} : L_1 \dots L_m / R_1 \dots R_n}{\text{resolvante} : L_1 \dots L_m / R_1 \dots R_n}$$

Règle de déduction de base

$$\frac{\text{résolvante : } L_0 L_1 \ldots L_m / R_1 \ldots R_n, \text{ clause choisie : } L'_0 L'_1 \ldots L'_{m'} / R'_1 \ldots R'_{n'}}{\text{résolvante : } \sigma(L'_1) \ldots \sigma(L'_{m'}) \sigma(L_1) \ldots \sigma(L_m) / \sigma(R'_1) \ldots \sigma(R'_{n'}) \sigma(R_1) \ldots \sigma(R_n)}$$

Règle de fin de déduction

$$\frac{\text{résolvante : } / R'_1 \ldots R'_n}{\text{réponse : } R'_1 \ldots R'_n}$$

où, bien entendu, L_0 et L'_0 sont des littéraux complémentaires unifiables et la substitution la plus générale qui les unifie. (Un exemple est donné plus tard).

La raison première du choix de cette technique de résolution linéaire avec ordre de sélection prédéfini des littéraux, était sa simplicité et le fait que nous pouvions produire des clauses logiquement déductibles du programme, ce qui garantissait donc d'une certaine façon la validité des résultats produits. Nous nous étions largement inspiré pour cela de la SL-résolution de Robert Kowalski que Philippe avait implanté pour sa thèse sur l'égalité formelle. Cependant, malgré son fonctionnement en pile analogue à l'appel de procédures des langages classiques, nous avions constaté que cette méthode introduisait des calculs, certes nécessaires dans le cas général, mais inutiles pour la plupart de nos exemples. C'est donc cette version extrêmement simplifiée de SL-Resolution (que nous venons de décrire formellement) qui fut adoptée et qui continue d'être la base de tous les Prologs.

Le choix concernant le traitement du non-déterminisme fut lié à des problèmes d'efficacité. La programmation par Philippe d'un certain nombre de méthodes avait montré qu'un des problèmes cruciaux était celui de l'explosion combinatoire et donc du manque de mémoire. Le « backtracking » pour la gestion du non-déterminisme fut rapidement adopté plutôt qu'un système de gestion de plusieurs branches de calcul résidant simultanément en mémoire, système qui aurait eu pour effet d'augmenter considérablement la taille de la mémoire nécessaire à l'exécution des déductions. Alain avait une prédilection pour cette méthode introduite par Robert Floyd [19] pour traiter les langages non-déterministes et l'enseignait à tous ses élèves. L'utilisation du « backtracking » amenait certes à la perte de la complétude dans le cas de déductions comportant des branches infinies, mais nous jugions que compte tenu de la simplicité de la stratégie de déduction, traitement des littéraux de gauche à droite, et choix des clauses dans l'ordre où elles étaient écrites, c'était au programmeur à veiller à ce que l'exécution de son programme se termine.

3.2 Caractéristiques du Prolog préliminaire

Outre le mécanisme de déduction dont nous avons parlé, un certain nombre de prédicats prédéfinis étaient rajoutés au système au fur et à mesure des besoins d'Alain et de Robert Pasero : des prédicats pour tracer une exécution, *copy* (pour copier un terme), *boum* (pour décomposer un identificateur en une liste de caractères ou pour le composer), *dif* (pour traiter l'égalité formelle de la thèse de Philippe). Il est à noter que nous nous refusions à inclure des prédicats d'entrée-sortie dans cette liste, car considérés comme trop éloignés de la logique. Ces entrées-sorties, la spécification des résolvantes initiales, ainsi que l'enchaînement entre programmes, étaient spécifiés dans un langage de commandes agissant sur des ensembles de clauses (lecture, copie, écriture, fusion, démonstration, etc). Ce langage sans instruction de contrôle ne permettait de définir que des enchaînements définis statiquement mais il avait comme vertu de traiter uniformément les communications à travers la notion d'ensemble de clauses.

Soulignons que déjà, dans cette première version, figurait l'évaluation par nécessité (ou co-routinée si l'on préfère) de certains prédicats, en l'occurrence *dif* et *boum*. Le prédicat *dif* fut abandonné dans la version suivante mais est réapparu dans des Prologs modernes. Les seuls opérateurs de contrôle étaient placés en fin de clauses comme signes de ponctuation et avaient pour effet de procéder à des coupures dans l'espace de recherche. Le signe

« .. » effectuait une coupure après la tête de la clause,
« .; » effectuait une coupure après l'exécution de toute la règle,
« ;. » effectuait une coupure après production d'au moins une réponse,
« ;; » était sans effet.

Ces opérateurs extra-logiques étaient suffisamment exotiques pour poser problème à leurs usagers. Ils furent donc abandonnés par la suite. Curieusement cette ponctuation avait été introduite par Alain en souvenir des règles de transformations optionnelles et obligatoires du linguiste Noam Chomsky [8].

Sur le plan de la syntaxe, l'écriture des termes se faisait sous forme fonctionnelle, avec cependant la possibilité d'introduire des opérateurs unaires ou binaires définis par précédences, ainsi qu'un opérateur binaire infixé pouvant être représenté par absence de signe (comme un produit en mathématique et fort utile dans les entrées sorties de chaînes). Voici un exemple d'enchaînement de programmes qui produisait et écrivait un ensemble de clauses définissant les petits-neveux d'une personne dénommée Marie :

```
LIRE
   REGLES
   +DESC(*X,*Y) -ENFANT(*X,*Y);;
   +DESC(*X,*Z) -ENFANT(*X,*Y) -DESC(*Y,*Z);;
   +FRERESOEUR(*X,*Y)
      -ENFANT(*Z,*X) -ENFANT(*Z,*y) -DIF(*X,*Y);;
   AMEN

LIRE
   FAITS
   +ENFANT(PAUL,MARIE);;
   +ENFANT(PAUL,PIERRE);;
   +ENFANT(PAUL,JEAN);;
   +ENFANT(PIERRE,ALAIN);;
   +ENFANT(PIERRE,PHILIPPE);;
   +ENFANT(ALAIN,SOPHIE);;
   AMEN

LIRE
   QUESTION
   -FRERESOEUR(MARIE,*X)
      -DESC(*X,*Y) / +PETITNEVEUX(*Y) -MASC(*Y)..
   AMEN

CONCATENER(LIENSDEPARENTE,REGLES,FAITS)

DEMONTRER(LIENSDEPARENTE,QUESTION,REPONSE)

ECRIRE(REPONSE)

AMEN
```

La sortie de ce programme n'est donc pas un terme mais l'ensemble de clauses binaires :

```
+PETITNEVEUX(ALAIN) -MASC(ALAIN);.
+PETITNEVEUX(SOPHIE) -MASC(SOPHIE);.
+PETITNEVEUX(PHILIPPE) -MASC(PHILIPPE);.
```

La commande

- *lire* lit un l'ensemble de clause précédé d'un nom x et se terminant par *amen* et lui attribue le nom x ;
- *concatener*$(y, x_1, \ldots, x_n)$ unit les ensembles de clauses $x_1, \ldots, x_n$ pour obtenir l'ensemble de clauses y ;
- *demontrer*(x, y, z) lance une exécution, le programme étant x, l'ensemble des résolvantes de départ étant y et l'ensemble des réponses étant z ;
- *ecrire*(x) imprime l'ensemble x de clauses.

Le système de communication homme-machine fonctionnait en quatre phases et faisait intervenir 4 programmes, c'est-à-dire 4 ensembles de clauses,

C1 pour analyser un texte *T0* et produire une structure profonde *T1*,
C2 pour retrouver dans *T1* les antécédents des pronoms et produire une forme logique *T2*,
C3 pour décomposer la formule logique *T2* en un ensemble *T3* d'informations élémentaires,
C4 pour faire des déductions à partir de *T3* et produire les réponses *T4* en français.

La suite des commandes était donc

```
DEMONTRER(C1,T0,T1) DEMONTRER(C2,T1,T2) DEMONTRER(C3,T2,T3) DEMONTRER(C4,T3,T4)
```

où *T0* le texte français à traiter et *T4* les réponses produites étaient représentées par des faits élémentaires portant sur des listes de caractères.

3.3 Réalisation du Prolog préliminaire

Le centre de calcul localisé à Luminy ayant été déplacé, l'interprète fut réalisé par Philippe en Algol-W, sur la machine IBM 360-67 du Centre de calcul de l'Université de Grenoble, machine munie du système d'exploitation CP-CMS basé sur le concept de machines virtuelles. Cette machine, à laquelle nous étions connectés par ligne téléphonique spécialisée, possédait deux qualités pour ainsi dire uniques à cette époque, et essentielles à nos travaux : celle de permettre à un programmeur de disposer d'une mémoire virtuelle de 1Mo s'il le désirait (nous n'avions droit qu'à 750 Ko), et celle de permettre l'écriture de programmes interactifs. Ce fut donc sur une seule console, fonctionnant à 300 bauds, que fut réalisé non seulement l'interprète, mais également le système de questions-réponses lui-même. Le choix d'Algol-W s'était imposé par le fait qu'il était le seul langage de haut niveau dont nous disposions qui permettait la création dynamique d'objets structurés, tout en étant muni d'une récupération de mémoire.

L'implantation de la résolution était basée sur un codage des clauses sous forme de structures entre-pointées avec recopie anticipée de chaque règle utilisée dans une déduction, le non-déterminisme étant géré par une pile de backtracking et les substitutions effectuées uniquement par création de chaînes de pointeurs. Cette approche permettait d'éviter les copies de termes lors des unifications, et donc améliorait grandement les temps de calculs et la place mémoire utilisée.

L'analyseur des clauses était écrit lui aussi en Algol-W, les atomes étant gérés par une technique classique de « hash-code ». Cet analyseur constituait une partie non négligeable du système, ce qui conforta Alain dans son désir de résoudre ces problèmes de syntaxe en Prolog même ; mais l'expérience manquait encore sur ce sujet, l'objet de la première application étant de dégager les principes même de l'analyse syntaxique en programmation logique !

Chapitre 4

Le Prolog définitif

Ayant longuement décrit jusqu'à maintenant les deux ancêtres il est temps de donner la fiche signalétique du Prolog définitif de 1973. Notre souci majeur après la version préliminaire, est de renforcer l'aspect langage de programmation de Prolog en minimisant les concepts et en améliorant ses fonctionnalités interactives dans la gestion des programmes. Prolog devient un langage basé sur le seul principe de résolution et sur la mise à disposition d'un ensemble de de prédicats (procédures) prédéfinis permettant de tout faire dans le langage lui-même. Cet ensemble est conçu comme un ensemble minimal permettant de :

— créer et modifier des programmes en mémoire,
— lire des programmes source, les analyser et les charger en mémoire,
— interpréter dynamiquement des requêtes ayant une structure analogue aux autres éléments du langage,
— accéder dynamiquement à la structure et aux éléments d'une déduction,
— contrôler l'exécution d'un programme le plus simplement possible.

4.1 Stratégie de résolution

L'expérience de la première version nous amène à considérer une version simplifiée de la stratégie de résolution. Ce choix nous est dicté par les expériences des premiers programmeurs, mais aussi par des considérations d'efficacité et par le choix de Fortran comme langage de programmation de l'interprète qui nous oblige à gérer nous même l'espace mémoire. Les différences essentielles avec l'ancienne version sont les suivantes :

— plus d'évaluation retardée (*dif*, *boum*),
— remplacement du prédicat *boum* par le prédicat plus général *univ*,
— plus de mécanisme de génération de clauses comme résultat d'une déduction

mais les fonctions assert et retract alors notées *ajout* et *supp*,
— un seul opérateur pour le contrôle du backtracking , l'opération de coupure de l'espace de recherche « ! » notée alors « / »,
— le concept d'appel calculé (meta-call) permettant d'utiliser une variable à la place d'un littéral,
— l'accès aux littéraux ancêtres et à la résolvante en cours (considérée comme un terme), pour les programmeurs désireux de définir leurs propres mécanismes de résolution, à l'aide des prédicats évaluables *ancetre* et é*tat* qui ont disparu des Prologs actuels.

Le backtracking et le fait d'ordonner chaque ensemble de clauses définissant un prédicat, sont les éléments de base conservés comme technique de gestion du non-déterminisme. La version préliminaire de Prolog nous avait donné grande satisfaction sur ce point. La réduction par Alain à une seule primitive (le cut « / ») de la gestion du contrôle du backtracking, au lieu des trop nombreux concepts de la première version simplifie extraordinairement le langage. Il devient possible au programmeur non seulement de réduire la taille de son espace de recherche par des critères purement pragmatiques, mais aussi de réaliser un traitement de la négation, certes simplifié et réducteur dans sa sémantique, mais extrêmement utile pour la programmation la plus courante.

D'autre part, Philippe, après le séjour à Edinburgh a en tête les bases d'une architecture extrêmement simple à réaliser du point de vue de la gestion de la mémoire, et beaucoup plus efficace en temps et en espace, surtout si l'on conserve la gestion du non-déterminisme par « backtracking ». Enfin, toutes les premières expériences de programmation ont montré que cette technique a permis à nos usagers d'intégrer assez facilement le non-déterminisme comme une dimension additionnelle au contrôle de l'exécution de prédicats.

En ce qui concerne le traitement du « ou » implicite entre littéraux à l'intérieur d'une clause, la séquentialité s'impose là encore comme l'interprétation la plus naturelle de cet opérateur puisque formellement l'ordre d'exécution des littéraux n'affectait en rien l'ensemble des résultats obtenus (modulo l'éviction de branches infinies), comme l'avait prouvé Robert Kowalski à propos de SL-resolution.

En résumé, ces deux choix concernant le traitement du « et » et du « ou » se justifient pleinement par les objectifs assignés :

— utiliser une stratégie simple, prévisible et contrôlable par le programmeur, permettant de donner à tout prédicat extra-logique (comme les entrées-sorties) une définition opérationnelle.

— disposer d'un interprète capable de faire des déductions dont les profondeurs pouvaient être de mille ou même de dizaines de mille (chose totalement impensable dans les systèmes déductifs existant à l'époque).

4.2 Syntaxe et primitives

Dans l'ensemble, la syntaxe retenue est la même que celle de la version préliminaire. Sur le plan lexical, la syntaxe des identificateurs est proche de celle de la plupart des langages, qui à l'époque n'utilisaient pas les lettres minuscules (les claviers et systèmes d'exploitation ne le permettant pas systématiquement). Il faut noter que parmi les primitives de base traitant des problèmes de morphologie, une seule d'entre elles *univ* sert tout à la fois à créer dynamiquement un atome à partir d'une suite de caractères, à construire un objet structuré à partir de ses éléments, ou au contraire à effectuer les opérations inverses de décomposition. Cette primitive devient un des outils de base servant à la création dynamique de programmes ainsi qu'à la manipulation d'objets dont les structures ne sont pas connue avant l'exécution du programme.

La possibilité laissée à l'usager de définir ses propres opérateurs unaires et binaires par l'emploi de précédences numériques s'est avérée par la suite très utile et très souple d'emploi quoique compliquant les analyseurs de clauses. Elle subsiste toujours telle quelle dans les divers Prolog actuels.

Dans la version préliminaire de Prolog, il existait un moyen de créer des clauses logiquement déductibles à partir d'autre clauses. L'expérience nous a montré cependant qu'il est souvent nécessaire de manipuler des clauses dans des perspectives quelquefois très éloignées de la logique du premier ordre. Ce type de traitement intervient par exemple dans la programmation de raisonnements de type temporel, dans la gestion d'informations persistantes d'une durée de vie aléatoire, ou encore dans la simulation de logiques exotiques. Nous sentons que sur le plan de la sémantique il faudra encore de nombreuses recherches pour modéliser ces problèmes de mise à jour d'ensembles de clauses. D'où le choix extrêmement pragmatique de primitives extra-logiques agissant par effet de bord pour créer ou modifier un programme (*ajout*, *supp*). Ce choix semble avoir été le bon, puisque ces fonctions ont été conservées par la suite.

Un des manques ressentis avec la version préliminaire de Prolog est celui d'un mécanisme permettant de calculer un terme qui puisse être ensuite considéré comme un littéral à résoudre. Cette fonction essentielle pour réaliser des méta-

programmes, comme par exemple un interprète de commandes, est introduite très simplement du point de vue syntaxique. Dans toute clause, une variable désignant un terme peut figurer comme littéral.

Dans le même esprit, et ceci originellement à l'intention des spécialistes de la logique computationnelle, sont introduites des primitives permettant d'accéder à la déduction en cours en la considérant comme un objet de type Prolog (*add*, *delete*). De même le prédicat « / » (prononcé « cut » par les édimbourgeois) devient paramétrable d'une façon très puissante par accès aux ancêtres.

4.3 Un exemple de programme

Pour que le lecteur ait une idée de ce à quoi ressemble un programme Prolog de l'époque nous reprenons un vieil exemple. Il s'agit de déterminer des itinéraires d'avions respectant certaines contraintes d'horaires, à partir de la connaissance des vols directs.

Les données de base (vols directs entre deux villes) sont représentées par des clauses unaires de la forme :

+vol(<ville départ>, <ville arrivée>, <heure départ>, <heure arrivée>,
<nom de vol>).

où les heures sont représentées par des couples d'entiers de la forme *<heure>* : *<minute>*. Nous supposons que tous les vols s'effectuent entièrement dans une même journée.

Le prédicat suivant est utilisé par l'usager pour établir un itinéraire.

itineraire(<ville départ>, <ville arrivée>, <heure départ>, <heure arrivée>,
<heure mini départ>, <heure maxi arrivée>)

Il énumère (et écrit en sortie) tous les couples de villes reliées par un vol avec ou sans escales (la même ville ne figurant qu'une seule fois, sauf pour les circuits qui partent et arrivent au même endroit). Les paramètres *<heure départ>* et *<heure arrivée>* désignent respectivement l'heure de départ et l'heure d'arrivée calculées.

Enfin, *<heure mini départ>* et *<heure maxi arrivée>* sont des contraintes d'horaires (et les seuls paramètres que doit fournir l'usager du prédicat). Tout plan de vol calculé par le prédicat fournira un premier vol dont l'heure de départ suit *<heure mini départ>* et un dernier vol dont l'heure d'arrivée précède *<heure maxi arrivée>*.

Pour définir ce prédicat *itineraire*, plusieurs prédicats intermédiaires sont définis. Le prédicat :

route(<*ville départ*>, <*ville arrivée*>, <*heure départ*>, <*heure arrivée*>,
<*itinéraire*>, <*escales*>, <*heure mini départ*>, <*heure maxi arrivée*>)

est analogue à *itineraire*. Il comporte deux paramètres supplémentaires : en entré <*escales*> qui désigne la liste $c_k.c_{k-1}\ldots c_1.nil$ des villes déjà visitées (dans l'ordre inverse du parcours) et en sortie <*itinéraire*> qui désigne la liste $f_1\ldots f_k.nil$ des noms des vols constituant le voyage.

Les prédicats *precedehoraires*(h_1, h_2) et *plushoraires*(h_1, h_2, h_3) concernent le traitement arithmétique des horaires. Le prédicat *ecrireplan* permet d'écrire un plan comme une suite de noms de vols. Enfin, *non*(p) définit une négation par échec pour p, et *element*(e, l) s'évalue à vrai si l'élément e figure dans la liste l d'éléments.

Voici donc le programme final avec ses données et le lancement d'une exécution.

```
* OPERATEURS INFIXES
--------------------.

-AJOP( ".", 1, "(X|X)|X")  -AJOP(":",  2, "(X|X)|X") !

* PREDICAT USAGER
-----------------.

+ITINERAIRE(*DEP, *ARR, *HDEP, *HARR, *HMINDEP, *HMAXARR)
   -ROUTE(*DEP, *ARR, *HDEP, *HARR, *PLAN, *VDEP.NIL, *HMINDEP, *HMAXARR)
   -SORM("-------------------------------------------")
   -LIGNE
   -SORM("PLAN DE  VOL  ENTRE:  ")
   -SORT(*DEP)
   -SORM(" ET: ")
   -SORT(*ARR)
   -LIGNE
   -SORM("-------------------------------------------")
   -LIGNE
   -SORM("HORAIRE DEPART:      ")
   -SORT(*HDEP)
   -LIGNE
   -SORM("HORAIRE ARRIVEE:    ")
   -SORT(*HARR)
   -LIGNE
   -SORM(" VOLS:    ")
   -ECRIREPLAN(*PLAN)
   -LIGNE
   -LIGNE.
```

```
* PREDICATS  DE CALCUL
----------------------.

+ROUTE(*DEP, *ARR, *HDEP, *HARR, *NOMVOL.NIL, *VISITES, *HMINDEP, *HMAXARR)
   -VOL(*DEP, *ARR, *HDEP, *HARR, *NOMVOL)
   -PRECEDEHORAIRES(*HMINDEP, *HDEP)
   -PRECEDEHORAIRES(*HARR, *HMAXARR).

+ROUTE(*DEP,*ARR, *HDEP,*HARR, *PLAN, *VISITES, *HMINDEP, *HMAXARR)
   -VOL(*DEP, *ESCALE, *HDEP, *HARRESCALE, *NOMVOL)
   -PRECEDEHORAIRES(*HMINDEP, *HDEP)
   -PLUSHORAIRES(*HARRESCALE, 00:15, *HMINDEPESCALE)
   -PRECEDEHORAIRES(*HMINDEPESCALE, *HMAXARR)
   -NON(ELEMENT(*ESCALE, *VISITES))
   -ROUTE(*ESC, *ARR, *HDEPESCALE, *HARR, *NOMVOL.*PLAN,
   *ESCALE.*VISITES, *HMINDEPESCALE, *HMAXARR).

+PRECEDEHORAIRES(*H1:*M1, *H2:*M2) -INF(*H1, *H2).
+PRECEDEHORAIRES(*H1:*M1, *H1:*M2) -INF(*M1, *M2).

+PLUSHORAIRES(*H1:*M1, *H2:*M2, *H3:*M3)
   -PLUS(*M1, *M2, *M)
   -RESTE(*M, 60, *M3)
   -DIV(*M, 60,*H)
   -PLUS(*H, *H1, *HH)
   -PLUS(*HH,*H2,*H3).

+ECRIREPLAN( *X. NIL) -/ -SORT(*X).
+ECRIREPLAN( *X.*Y) -SORT(*X) -ECRIT(-) -ECRIREPLAN(*Y).

+ELEMENT(*X, *X.*Y).
+ELEMENT(*X, *Y.*Z) -ELEMENT(*X, *Z).

+NON(*X) -*X -/ -ECHEC.
+NON(*X).

* LISTE DES VOLS
----------------.

+VOL(PARIS, LONDRES, 06:50, 07:30, AF201).
+VOL(PARIS, LONDRES, 07:35, 08:20, AF210).
+VOL(PARIS, LONDRES, 09:10, 09:55, BA304).
+VOL(PARIS, LONDRES, 11:40, 12:20, AF410).
+VOL(MARSEILLE, PARIS, 06:15, 07:00, IT100).
+VOL(MARSEILLE, PARIS, 06:45, 07:30, IT110).
+VOL(MARSEILLE, PARIS, 08:10, 08:55, IT308).
+VOL(MARSEILLE, PARIS, 10:00, 10:45, IT500).
```

```
+VOL(MARSEILLE, LONDRES, 08:15, 09:45, BA560).
+VOL(MARSEILLE, LYON, 07:45, 08:15, IT115).
+VOL(LYON, LONDRES, 08:30, 09:25, TAT263).


* SAUVEGARDE DU PROGRAMME
------------------------.

-SAUVE!

* INTERROGATION
--------------.

-ITINERAIRE(MARSEILLE, LONDRES, *HD, *HA, 00:00, 09:30)!
```

Voici les réponses fournies par l'ordinateur

```
-----------------------------------------
PLAN DE  VOL  ENTRE:   MARSEILLE  ET: LONDRES
-----------------------------------------
HORAIRE DEPART:   06:15
HORAIRE ARRIVEE:  08:20
VOLS:  IT100-AF210

-----------------------------------------
PLAN DE  VOL  ENTRE:   MARSEILLE  ET: LONDRES
-----------------------------------------
HORAIRE DEPART:   07:45
HORAIRE ARRIVEE:  09:25
VOLS:  IT115-TAT263
```

4.4 Réalisation de l'interprète

Le système de résolution, dans lequel le non-déterminisme est géré par backtracking, est réalisé à l'aide d'une représentation des clauses tout à fait originale se situant à mi-chemin entre la technique basée sur le partage de structures qu'utilisaient Robert Boyer et Jay Moore dans leurs travaux sur la preuve de programmes [6, 32] et la technique par backtracking utilisée dans la version préliminaire de Prolog. C'est au cours de son séjour à Edimbourg que Philippe, après de nombreuses discussions avec Robert Boyer, imagine cette solution. Dans cette nouvelle approche, les clauses d'un programme sont codées en mémoire comme un ensemble de squelettes (templates) pouvant être instanciés sans recopie plusieurs fois dans une même déduction grâce à des contextes contenant les substitutions à effectuer

sur les variables. Cette technique a de nombreux avantages par rapport à celles utilisées alors en démonstration automatique :

— l'unification s'effectuait, dans tous les systèmes connus, en des temps au mieux linéaires par rapport à la taille des termes unifiés ; dans notre système, la plupart des unifications s'effectuent dans des temps constants, fonction non pas de la taille des données, mais de celle des squelettes mis en jeu par les clauses du programme appelé ; ainsi la concaténation de deux listes s'effectue en un temps linéaire par rapport à la taille de la première, et non pas comme dans tous les systèmes basés sur des techniques de recopie, en des temps quadratiques ;

— selon le même schéma, la place mémoire nécessaire à une étape de déduction est fonction, non pas des données, mais de la clause du programme utilisée ; ainsi globalement la concaténation de deux listes ne mettra en oeuvre qu'une quantité de mémoire proportionnelle à la première des deux listes ;

— la gestion du non déterminisme ne nécessite pas, en première approche, de récupération de mémoire sophistiquée, mais simplement l'emploi de plusieurs piles synchronisées sur le « backtracking », facilitant ainsi une réalisation rapide et néanmoins économe de l'interprète.

En ce qui concerne la représentation en mémoire des squelettes, il est décidé d'utiliser une représentation préfixée (exactement l'inverse de la notation polonaise). Le système est composé de l'interprète proprement dit (c'est-à-dire la machine inférentielle munie de la bibliothèque des prédicats prédéfinis), d'un chargeur destiné à lire des clauses dans une syntaxe restreinte et enfin d'un superviseur écrit en Prolog. Ce superviseur contient entre autres un évaluateur de requêtes, un analyseur acceptant la syntaxe avec opérateurs, un chargeur de clauses en mémoire et la définition d'une bibliothèque de prédicats d'entrée-sorties de haut niveau.

Alain, qui comme nous tous n'aime guère Fortran, finit cependant par convaincre l'équipe de l'adopter pour programmer l'interprète. Ce choix essentiel repose sur la large diffusion de Fortran sur l'ensemble des machines de l'époque, et sur le fait que la machine à notre disposition n'accepte pas d'autres langages adaptés à cette tâche. Nous espérons ainsi disposer d'un système portable, prévision qui se révélera exacte.

Gérard Battani [1] et Henri Meloni, sous la supervision de Philippe, réalisent l'interprète proprement dit entre juin 1973 et octobre 1973 sur un CII 10070 (variante du SIGMA 7) tandis que René Bazzoli est chargé d'écrire le superviseur dans le langage Prolog lui-même sous la direction d'Alain. L'ensemble est constitué

d'environ 2000 instructions, sensiblement de la même taille que le programme Algol-W de la version initiale.

La machine est dotée d'un système d'exploitation par lots (batch) sans possibilité d'interactions à travers un terminal. Données et programmes sont donc introduits sur cartes perforées. On mesure la performance de ces jeunes chercheurs pour réaliser un système aussi complexe que celui-là dans un tel contexte et dans un délai aussi court, en sachant qu'aucun d'entre eux ne connaissait Fortran. Cet interprète est mis au point de façon définitive en décembre 1973 par Gérard Battani et Henry Méloni après un portage sur la machine IBM 360-67 de Grenoble, et donc dans des conditions d'exploitation plus raisonnables. Philippe écrit le manuel de référence et d'utilisation de ce nouveau Prolog seulement deux ans plus tard [36].

Chapitre 5

Appendice

5.1 Le programme de frère et sœur

Entre-temps Prolog et sa syntaxe c'est stabilisé : voire Prolog ISO-standard [16]. Les variables sont des identificateurs commençant par une majuscule. Une clause, c'est-à-dire une instruction,

$+p_0 - p_1 \cdots - p_n$ s'écrit p_0 `:-` $p_1, \cdots, p_n$.

Cette clause est interprétée comme,

p_0 est vrai si p_1 et ... et p_n est vrai

ou encore, d'une façon plus procédurale, comme,

pour exécuter p_0 il faut exécuter successivement p_1 et ... et p_n.

Considérons une petite banque de données : Hélène est fille de Marguerite, Nicole est fille de Rosalie, Rachel est fille de Rosalie, Jules est fils de Marguerite, Paul est fils de Marguerite, Robert est fils Rosalie. Plus une règle : x et z sont frère et sœur si x est le fils y et z est la fille de y. Cela se traduit par le programme Prolog :

```
freresoeur(X,Z) :- estfilsde(X,Y), estfillede(Z,Y).

estfillede(helene,marguerite).
estfillede(nicole,rosalie).

estfilsde(jules,marguerite).
estfilsde(paul,marguerite).
```

La question « quel est le x qui est tel que Paul et x sont frère et sœur ? » devient :

```
:- freresoeur(paul,X).
X=helene.
```

qui produit la réponse : Hélène.

Cette réponse se calcule à l'aide de trois paramètres : la *contrainte* courante, la *question* courante et la *règle* choisie courante.

1. Initialiser la *contrainte* à vide. Initialiser la *question* à la question initiale.
2. Si la *question* est vide alors une réponse est la *contrainte* limitée aux variables interrogées. Sinon choisir une *règle* dont le membre gauche, *l'appelant*, a pour prédicat le même que celui du premier littéral, *l'appelé*, de la *question* courante et renommer ses variable.
3. La *contrainte* courante est égale à $\{appelant = appelé\} \cup contrainte$
4. Résoudre la *contrainte*, c'est-à-dire moyennant les trois transformations, décrites ci-dessous, la mettre sous l'une des formes :
   (a) $\{x_1 = t_1[x_2], x_2 = t_1[x_3], \ldots, x_n = t_n[x_1]\}$, les x_i étant des variables et $t_i[x_j]$ étant un terme contenant la variable x_j. Cette contrainte est *insoluble* est un *cycle.*
   (b) $\{f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n)\}$, avec f et g des symboles fonctionnels différents et les s_i et t_i des termes quelconques. Cette contrainte est *insoluble.*
   (c) $\{x_1 = t_1, \ldots, x_n = t_n\}$, les x_i étant des variables toutes différentes, les t_i des termes quelconque et le système ne contenant pas de cycle. Cette contrainte est *soluble.*
5. Si la *contrainte* est insoluble alors s'arrêter. Sinon considérer la *question* comme la *question* dans laquelle on a remplacé l'appelant par la queue de *règle.*
6. Aller à l'étape 2.

Ces transformations sont :

1. $t = x \rightarrow x = t$, avec x une variable et t un terme.
2. $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \rightarrow s_1 = t_1, \ldots, s_n = t_n$, avec f un symbole fonctionnel et les s_i et t_i des termes quelconque.
3. $insoluble \cup C \rightarrow insoluble$, où C est une contrainte quelconque.

Cela donne :

```
contrainte :
 vide
question : :- freresoeur(paul,X).
 règle :
  freresoeur(X1,Z1) :- estfilsde(X1,Y1), estfillede(Z1,Y1).
 contrainte :
  {freresoeur(paul,X)=freresoeur(X1,Z1)} ∪ vide
  {X1=paul, X=Z1}
 question : :- estfilsde(X1,Y1), estfillede(Z1,Y1).
  règle : estfilsde(jules,marguerite) :- .
   contrainte :
    {estfilsde(X1,Y1)=estfilsde(jules,marguerite) ∪
    {X1=paul, X=Z1}}
    {paul=jules}
   fin : echec
  règle : estfilsde(paul,marguerite) :- .
   contrainte :
    {estfilsde(X1,Y1)=estfilsde(paul,marguerite) ∪
    {X1=paul, X=Z1}
    {X1=paul, X=Z1, Y1=marguerite}
   question : :- estfillede(Z1,Y1).
    règle : estfillede(helene,marguerite) :- .
    contrainte :
     {estfillede(Z1,Y1)=estfillede(helene,marguerite) ∪
     {X1=paul, X=Z1, Y1=marguerite}
     {X1=paul, X=Z1, Y1=marguerite, Z1=helene}
    question : :- .
    fin : {X=helene}
    règle : estfillede(nicole,rosalie) :- .
    contrainte :
     {estfillede(Z1,Y1)=estfillede(nicole,rosalie)} ∪
     {X1=paul, X=Z1, Y1=marguerite}
     {marguerite=rosalie}
    fin: echec
```

Une autre question « quel est le x qui est tel que x et Hélène sont frère et sœur ? » produit deux réponses :

```
:- freresoeur(X,helene).
X=jules.
X=paul.
```

« Ce que je ne comprends pas c'est pourquoi mon frère a deux sœurs alors que je n'en ai qu'une. »

5.2 Le programme des repas

Considérons le programme :

```
repas(E,P,D) :- entree(E), plat(P), dessert(D).

plat(P) :- viande(P).
plat(P) :- poisson(P).

entree(artichautsMelanie).
entree(cressonOeufPoche).
entree(truffesSousLeSel).

viande(grilladeDeBoeuf).
viande(pouletAuTilleul).
```

```
poisson(barAuxAlgues).
poisson(loupAuFenouile).

dessert(fraisesChantilly).
dessert(melonEnSurprise).
dessert(sorbetAuxPoires).
```

Si on demande quels sont les repas on obtient :

```
:- repas(E,P,D).
E=artichautsMelanie, P=grilladeDeBoeuf, D=fraisesChantilly;
E=artichautsMelanie, P=grilladeDeBoeuf, D=melonEnSurprise;
E=artichautsMelanie, P=grilladeDeBoeuf, D=sorbetAuxPoires;
E=artichautsMelanie, P=pouletAuTilleul, D=fraisesChantilly;
E=artichautsMelanie, P=pouletAuTilleul, D=melonEnSurprise;
E=artichautsMelanie, P=pouletAuTilleul, D=sorbetAuxPoires;
E=artichautsMelanie, P=barAuxAlgues, D=fraisesChantilly;
E=artichautsMelanie, P=barAuxAlgues, D=melonEnSurprise;
E=artichautsMelanie, P=barAuxAlgues, D=sorbetAuxPoires;
E=artichautsMelanie, P=loupAuFenouile, D=fraisesChantilly;
E=artichautsMelanie, P=loupAuFenouile, D=melonEnSurprise;
E=artichautsMelanie, P=loupAuFenouile, D=sorbetAuxPoires;

E=cressonOeufPoche, P=grilladeDeBoeuf, D=fraisesChantilly;
E=cressonOeufPoche, P=grilladeDeBoeuf, D=melonEnSurprise;
E=cressonOeufPoche, P=grilladeDeBoeuf, D=sorbetAuxPoires;
E=cressonOeufPoche, P=pouletAuTilleul, D=fraisesChantilly;
E=cressonOeufPoche, P=pouletAuTilleul, D=melonEnSurprise;
E=cressonOeufPoche, P=pouletAuTilleul, D=sorbetAuxPoires;
E=cressonOeufPoche, P=barAuxAlgues, D=fraisesChantilly;
E=cressonOeufPoche, P=barAuxAlgues, D=melonEnSurprise;
E=cressonOeufPoche, P=barAuxAlgues, D=sorbetAuxPoires;
E=cressonOeufPoche, P=loupAuFenouile, D=fraisesChantilly;
E=cressonOeufPoche, P=loupAuFenouile, D=melonEnSurprise;
E=cressonOeufPoche, P=loupAuFenouile, D=sorbetAuxPoires;

E=truffesSousLeSel, P=grilladeDeBoeuf, D=fraisesChantilly;
```

```
E=truffesSousLeSel, P=grilladeDeBoeuf, D=melonEnSurprise;
E=truffesSousLeSel, P=grilladeDeBoeuf, D=sorbetAuxPoires;
E=truffesSousLeSel, P=pouletAuTilleul, D=fraisesChantilly;
E=truffesSousLeSel, P=pouletAuTilleul, D=melonEnSurprise;
E=truffesSousLeSel, P=pouletAuTilleul, D=sorbetAuxPoires;
E=truffesSousLeSel, P=barAuxAlgues, D=fraisesChantilly;
E=truffesSousLeSel, P=barAuxAlgues, D=melonEnSurprise;
E=truffesSousLeSel, P=barAuxAlgues, D=sorbetAuxPoires;
E=truffesSousLeSel, P=loupAuFenouile, D=fraisesChantilly;
E=truffesSousLeSel, P=loupAuFenouile, D=melonEnSurprise;
E=truffesSousLeSel, P=loupAuFenouile, D=sorbetAuxPoires.
```

5.3 Le programme des repas équilibrés

Dans le Prolog il y a un certain nombre de prédicat « prédéfinis ». Entre autres il est possible de calculer le résultat y d'une expression arithmétique sous forme fonctionnelle $f(x_1, \ldots, x_n)$ par $y\ is\ f(x_1, \ldots, x_n)$. Il est aussi possible de tester si x_1 est plus petit que x_2 par $x_1 < x_2$.

Nous pouvons compléter le programme précédent par :

```
repasEquilibre(E,P,D) :- repas(E,P,D), equilibre(E,P,D).

equilibre(E,P,D) :-
   calories(E,X), calories(P,Y), calories(D,Z),
   V is X+Y+Z, V < 800.

calories(artichautsMelanie,147).
calories(barAuxAlgues,302).
calories(cressonOeufPoche,202).
calories(fraisesChantilly,398).
calories(grilladeDeBoeuf,530).
calories(loupAuFenouile,254).
calories(melonEnSurprise,122).
calories(pouletAuTilleul,400).
calories(sorbetAuxPoires,223).
calories(truffesSousLeSel,212).
```

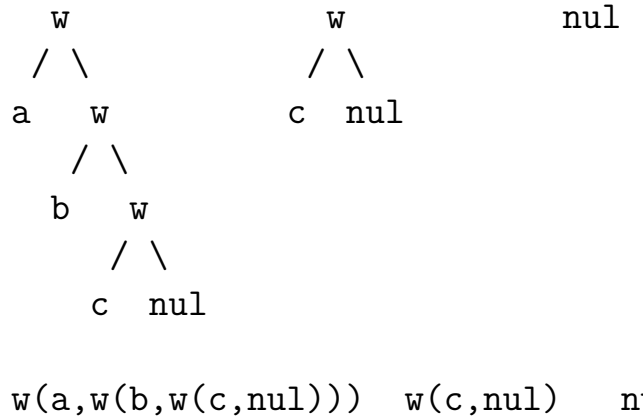
On calcule tous les repas équilibrés :

```
:- repasEquilibre(E,P,D).
E=artichautsMelanie, P=grilladeDeBoeuf, D=melonEnSurprise;
E=artichautsMelanie, P=pouletAuTilleul, D=melonEnSurprise;
E=artichautsMelanie, P=pouletAuTilleul, D=sorbetAuxPoires;
E=artichautsMelanie, P=barAuxAlgues, D=melonEnSurprise;
E=artichautsMelanie, P=barAuxAlgues, D=sorbetAuxPoires;
E=artichautsMelanie, P=loupAuFenouile, D=fraisesChantilly;
E=artichautsMelanie, P=loupAuFenouile, D=melonEnSurprise;
E=artichautsMelanie, P=loupAuFenouile, D=sorbetAuxPoires;
E=cressonOeufPoche, P=pouletAuTilleul, D=melonEnSurprise;
E=cressonOeufPoche, P=barAuxAlgues, D=melonEnSurprise;
E=cressonOeufPoche, P=barAuxAlgues, D=sorbetAuxPoires;
E=cressonOeufPoche, P=loupAuFenouile, D=melonEnSurprise;
E=cressonOeufPoche, P=loupAuFenouile, D=sorbetAuxPoires;
E=truffesSousLeSel, P=pouletAuTilleul, D=melonEnSurprise;
E=truffesSousLeSel, P=barAuxAlgues, D=melonEnSurprise;
E=truffesSousLeSel, P=barAuxAlgues, D=sorbetAuxPoires;
E=truffesSousLeSel, P=loupAuFenouile, D=melonEnSurprise;
E=truffesSousLeSel, P=loupAuFenouile, D=sorbetAuxPoires.
```

5.4 Le programme de l'ajout de listes

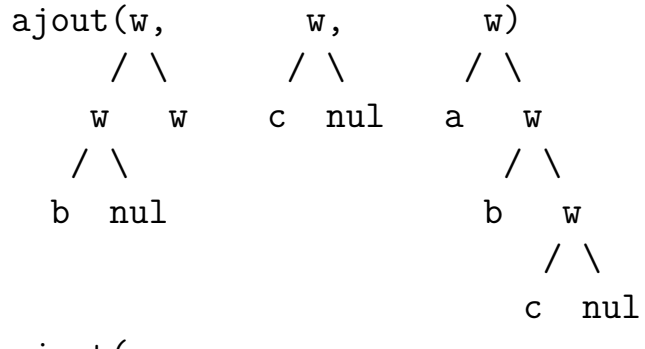
Maintenant quelque chose de plus compliqué.

Nous allons travailler sur des *listes.* Voici quelques exemples de listes :




```
w(a,w(b,w(c,nul)))  w(c,nul)   nul
```

Nous introduisons la relation ternaire *d'ajout* :




```
ajout(
w(a,w(b,nul)), w(c,nul), w(a,w(b,w(c,nul))))
```

Elle se programme ainsi dans le cas général :

```
ajout(nul,Y,Y).
ajout(w(A,X),Y,w(A,Z)) :- ajout(X,Y,Z).
```

Voici une première question,

```
:- ajout(w(a,w(b,nul)), w(c,nul), Z).

Z=w(a,w(b,w(c,nul))).
```

et une deuxième question plus subtile,

```
:- ajout(X,Y,w(a,nul)).

X=nul, Y=w(a,nul).
X=w(a,nul), Y=nul.
```

Intéressons nous au détail de l'exécution :

```
contrainte : vide
question : :- ajout(X,Y,w(a,nul)).
 règle : ajout(nul,Y1,Y1) :- .
 contrainte :
  {ajout(X,Y,w(a,nul))=ajout(nul,Y1,Y1)} ∪ vide
  {X=nul, Y=Y1, Y1=w(a,nul))}
 question : :- .
 fin : {X=nul, Y=w(a,nul))}
 règle : ajout(w(A1,X1),Y1,w(A1,Z1)) :- ajout(X1,Y1,Z1).
 contrainte :
  {ajout(X,Y,w(a,nul))=
  ajout(w(A1,X1),Y1,w(A1,Z1))}∪vide
  {X=w(A1,X1), Y=Y1, A1=a, Z1=nul}
  question : :- ajout(X1,Y1,Z1).
   règle : ajout(nul,Y2,Y2):- .
   contrainte :
    {ajout(X1,Y1,Z1) = ajout(nul,Y2,Y2)} ∪
    {X=w(A1,X1), Y=Y1, A1=a, Z1=nul}
    {X=w(A1,X1), Y=Y1, A1=a, Z1=nul, X1=nul, Y1=Y2, Y2=Z1}
   question : :- .
   fin : {X=w(a,nul),Y=nul}
   règle : ajout(w(A2,X2),Y2,w(A2,Z2)) :- ajout(X2,Y2,Z2).
   contrainte :
    {ajout(X1,Y1,nul) = ajout(w(A2,X2),Y2,w(A2,Z2))} ∪
    {X=w(A1,X1), Y=Y1, A1=a, Z1=nul}
    {nul=w(A3,X3)}
   fin : echec
```

5.5 Le programme des mutants avec des parenthèses

En partant de l'ensemble d'animaux : alligator, bouquetin, caribou, cheval, chevre, hibou, lapin, ours, pintade, tortue, vache, avec beaucoup de parenthèse nous produirons : alligatortue, caribouquetin, caribours, chevalligator, chevalapin, hibouquetin, hibours, lapintade, vacheval, vachevre avec toujours des parenthèses.

```
mutantprovisoire(Z) :-
   animalprovisoire(X),
   animalprovisoire(Y),
   ajout(A,B,X),
   ajout(B,C,Y),
   nonnul(B),
   nonnul(A),
   ajout(X,C,Z).

nonnul(w(A,X)).

ajout(nul,Y,Y).
ajout(w(A,X),Y,w(A,Z)) :- ajout(X,Y,Z).
animalprovisoire(w(a,w(l,w(l,w(i,w(g,w(a,w(t,w(o,w(r,nul))))))))).
animalprovisoire(w(b,w(o,w(u,w(q,w(u,w(e,w(t,w(i,w(n,nul))))))))).
animalprovisoire(w(c,w(a,w(r,w(i,w(b,w(o,w(u,nul))))))).
animalprovisoire(w(c,w(h,w(e,w(v,w(a,w(l,nul))))))).
animalprovisoire(w(c,w(h,w(e,w(v,w(r,w(e,nul))))))).
animalprovisoire(w(h,w(i,w(b,w(o,w(u,nul)))))).
animalprovisoire(w(l,w(a,w(p,w(i,w(n,nul)))))).
animalprovisoire(w(p,w(i,w(n,w(t,w(a,w(d,w(e,nul))))))).
animalprovisoire(w(o,w(u,w(r,w(s,nul))))).
animalprovisoire(w(p,w(i,w(n,w(t,w(a,w(d,w(e,nul))))))).
animalprovisoire(w(t,w(o,w(r,w(t,w(u,w(e,nul))))))).
animalprovisoire(w(v,w(a,w(c,w(h,w(e,nul)))))).
:- mutantprovisoire(Z).

Z = w(a,w(l,w(l,w(i,w(g,w(a,w(t,w(o,w(r,w(t,w(u,w(e,nul))))))))))) ;
Z = w(c,w(a,w(r,w(i,w(b,w(o,w(u,w(q,w(u,w(r,(s,nul))))))))))) ;
Z = w(c,w(a,w(r,w(i,w(b,w(o,w(u,w(r,w(s,nul))))))))) ;
Z = w(c,w(h,w(e,w(v,w(a,w(l,w(l,w(i,w(g,w(a,w(t,w(o,w(r,nul))))))))))
    ))))) ;
Z = w(c,w(h,w(e,w(v,w(a,w(l,w(a,w(p,w(i,w(n,nul))))))))))) ;
Z = w(h,w(i,w(b,w(o,w(u,w(q,w(u,w(e,w(t,w(i,w(n,nul))))))))))) ;
Z = w(h,w(i,w(b,w(o,w(u,w(r,w(s,nul))))))) ;
Z = w(l,w(a,w(p,w(i,w(n,w(t,w(a,w(d,w(e,nul))))))))) ;
Z = w(l,w(a,w(p,w(i,w(n,w(t,w(a,w(d,w(e,nul))))))))) ;
Z = w(v,w(a,w(c,w(h,w(e,w(v,w(a,w(l,nul))))))))) ;
Z = w(v,w(a,w(c,w(h,w(e,w(v,w(r,w(e,nul))))))))) ;
false.
```

5.6 Le programme des mutants

En fait les listes jouent un rôle tellement important en Prolog qu'il existe des notations spéciales pour les manipuler. Le symbole fonctionnel unaire w est remplacé par « . » et la constante *nul* par []. On écrit $[t_1|t_2]$ au lieu de $.(t_1, t_2)$ et $[t_1, \ldots, t_n]$ au lieu de la liste $.(t_1, .(t_2, \cdots .(t_n, [])\cdots))$.

On dispose du prédicat prédéfini *atom_chars*(x, y) qui crée l'identificateur x formé de la liste y de caractères ou la liste y de caracteres de l'identificateur x. Avec ce prédicat prédéfini on peut enfin calculer des vrais mutants. Il suffit de compléter le programme précédent et de remplacer le prédicat *nonnul* par *nonvide* et le prédicat *ajouter* par *adjonction*.

```
mutant(Zp) :-
   animal(Xp),
   animal(Yp),
   atom_chars(Xp,X),
   atom_chars(Yp,Y),
   adjonction(A,B,X),
   adjonction(B,C,Y),
   nonvide(B),
   nonvide(A),
   adjonction(X,C,Z),
   atom_chars(Zp,Z).

nonvide([A|X]).

adjonction([],Y,Y).
adjonction([A|X],Y,[A|Z]) :- adjonction(X,Y,Z).


animal(alligator).
animal(bouquetin).
animal(caribou).
animal(cheval).
animal(chevre).
animal(hibou).
animal(lapin).
animal(ours).
animal(pintade).
animal(tortue).
animal(vache).
```

A la question « quelles sont les mutants ? » on obtient :

```
:- mutant(X).

X = alligatortue ;
X = caribouquetin ;
X = caribours ;
X = chevalligator ;
X = chevalapin ;
X = hibouquetin ;
X = hibours ;
X = lapintade ;
X = vacheval ;
X = vachevre ;
false.
```

Conclusion

Après toutes ces péripéties et tous ces détails techniques il serait intéressant de prendre du recul et de situer la naissance de Prolog dans une perspective plus vaste.

L'article de Alan Robinson publié en janvier 1965 A machine-oriented logic based on the resolution principle contenait en germe le langage Prolog. Cet article a été à la base d'un courant important de travaux sur la démonstration automatique et nul doute que Prolog est essentiellement un démonstrateur de théorèmes « à la Robinson ».

Notre contribution à été de transformer ce démonstrateur en un langage de programmation. Pour cela nous n'avons pas eu peur d'introduire des mécanismes et des restrictions purement informatiques qui étaient des hérésies pour le modèle théorique existant. Ce sont ces modifications, si souvent critiquées, qui ont assuré la viabilité et donc le succès de Prolog. Le mérite de Robert Kowalski a été de dégager la notion de « clause de Horn » qui a légitimé notre principale hérésie : une stratégie de démonstration linéaire avec « backtracking » et des unifications uniquement sur les têtes de clauses.

Le langage Prolog est si simple que l'on a l'impression que tôt ou tard quelqu'un devait le découvrir. Pourquoi nous plutôt que d'autres ? Tout d'abord Alain était bien armé pour créer un langage de programmation. Il appartenait à la première génération de docteurs d'informatique en France et sa spécialité était la théorie des langages. Il avait accumulé une précieuse expérience en concevant un premier langage de programmation, les systèmes-Q, dans le cadre du projet de Traduction Automatique de l'Université de Montréal. Ensuite notre rencontre, la créativité de Philippe et les conditions de travail particulières à Marseille firent le reste ! Nous avons bénéficié d'une grande liberté d'action dans un centre scientifique nouvellement créé et sans pression extérieure nous avons donc pu nous consacrer entièrement à notre projet.

C'est sans doute pourquoi cette période de notre vie reste une des plus heureuses dans nos souvenirs communs. Nous avons pris plaisir à l'évoquer pour écrire ce papier tout en goûtant les amandes fraîches autour d'un Martini dry.

Bibliographie

[1] Battani Gérard et Henry Méloni, *Interpréteur du langage PROLOG,* rapport de DEA, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, 1973.

[2] Battani Gérard, *Mise en oeuvre des contraintes phonologiques, syntaxiques et sémantiques dans un système de compréhension automatique de la parole,* thèse de 3ième cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Juin 1975.

[3] Bergman Marc and Henry Kanoui, *Application of mechanical theorem proving to symbolic calculus,* Third International Colloquium on advanced Computing Methods in Theoretical Physics, Marseille, France, Juin 1973.

[4] Bergman Marc, *Résolution par la démonstration automatique de quelques problèmes en intégration symbolique sur calculateur,* thèse de 3ième cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Octobre 1973.

[5] Bergman Marc et Henry Kanoui, *SYCOPHANTE, système de calcul formel sur ordinateur,* rapport de fin de contrat DRET (Direction des Recherches et Etudes Techniques), Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, 1975.

[6] Boyer Robert S. et J S. Moore, *The sharing of Structure in Theorem Proving Programs,* Machine Intelligence 7, édité par B. Melzer et D. Michie, Edinburgh University Press, New-York, pages 101 à 116, 1972.

[7] Chastellier (de) Guy et Alain Colmerauer, *W-Grammar,* Proccedings of the ACM Congress, San Francisco, Aout, ACM, New York, pages 511 à 518, 1969.

[8] Chomsky Noam, *Aspects of the Theory of Syntax,* MIT Press, Cambridge, 1965

[9] Cohen Jacques, *A view of the origins and development of Prolog,* Commun. ACM 31, 1, pages 26 à 36, janvier 1888.

[10] Colmerauer Alain, *Total precedence relations,* J. ACM 17, 1, pp. 14-30, Janvier 1970.

[11] Colmerauer Alain, *Les systèmes-q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur,* Internal publication 43, Département d'informatique de l'Université de Montréal, Septembre 1970.

[12] Colmerauer Alain, Fernand Didier, Robert Pasero, Philippe Roussel, Jean Trudel, *Répondre,* Internal publication, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Mai 1971. Cette publication est un listing avec des annotations éccrits à la main.

[13] Colmerauer Alain, Henry Kanoui, Robert Pasero et Philippe Roussel, *Un système de communication en français,* rapport préliminaire de fin de contrat IRIA, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Octobre 1972.

[14] Colmerauer Alain, *Les grammaires de métamorphose*, publication interne, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, novembre 1975. English version, *Metamorphosis grammars, Natural Language Communication with Computers,* Lectures Notes in Computer Science 63, édité par L. Bolc, Springer Verlag, Berlin Heidelberg, New York, pp. 133-189, 1978, ISBN 3-540-08911-X.

[15] Darlington J. L., *Theorem-proving and information retrieval,* Machine Intelligence 4, Edinburgh University Press, pp. 173-707, 1969.

[16] P. Deransart, A Ed-Dbali et L. Cervoni, *Prolog : The Standard Reference Manual* Springer, 1996.

[17] Elcok E. W. *Absys : the first logic programming language, a restrospective and a commentary,* Journal of Logic Programming, 9(1) :1-17, Juillet 1990.

[18] Floyd Robert W., *Syntactic analysis and operator precedence,* J.ACM 10, pp. 316-333, 1963.

[19] Floyd Robert W., *Nondeterministic algorithms,* J. ACM 14, 4, pp. 636-644, Octobre 1967.

[20] Green Cordell C., *Application of theorem-proving to problem-solving,* Proceedings of First International Joint Conference on Artificial Intelligence, Washington D.C., pp. 219-239, 1969.

[21] Hewitt Carl, *PLANNER : A language for proving theorems in robots,* Proceedings of First International Joint Conference on Artificaila Intelligence, Washington D.C., pp. 295-301, 1969.

[22] Joubert Michel, *Un système de résolution de problémes à tendance naturelle,* thèse de 3ième cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Février 1974.

[23] Kanoui Henry, *Application de la démonstration automatique aux manipulations algébriques et à l'intégration formelle sur ordinateur,* thèse de 3ième cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Octobre 1973.

[24] Kanoui Henry, *Some aspects of Symbolic Integration via Predicate Logic Programming,* ACM SIGSAM Bulletin, 1976.

[25] Kowalski Robert A. et D. Kuehner, *Linear resolution with selection function,* memo 78, University of Edinburgh, School of Artificial Intelligence, 1971. Also in Artificial Intelligence Vol. 2, pp. 227-60, 1971.

[26] Kowalski Robert A., *Predicate Logic as Programming Language,* memo 70, University of Edimburgh, School of Artificial Intelligence, November 1973. Also in Proceedings of IFIP 1974, North Holland Publishing Company, Amsterdam, pp. 569-574, 1974.

[27] Kowalski Robert A. and Maarten van Emden, *The semantics of predicate logic as programming language,* memo 78, University of Edinburgh, School of Artificial Intelligence, 1974. Aussi dans JACM 22, 1976, pages 733 à 742.

[28] Kowalski Robert A., *The early history of logic programming,* CACM vol. 31, no. 1, pp 38-43, 1988.

[29] Loveland, D.W. *Automated theorem proving : A quarter-century review,* Am. Math. Soc. 29 , pages 1 à 42, 1984.

[30] Luckam D. and N.J. *Nilson, Extracting information from resolution proof trees,* Artificial Intelligence 12, 1, pages 27 à 54, 1971

[31] Meloni Henry, *Mise en œuvre des contraintes phonologiques, syntaxiques et sémantiques dans un système de compréhension automatique de la parole,* thèse de 3ème cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, juin 1975.

[32] Moore, J, *Computational Logic : Structure sharing and proof of program properties,* part I and II, memo 67, University of Edinburgh, School of Artificial Intelligence, 1974.

[33] Pasero Robert, *Représentation du français en logique du premier ordre en vue de dialoguer avec un ordinateur,* thèse de 3ème cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, mai 1973.

[34] Pereira Fernando C. and David H.D. Warren, *Definite clause grammars for language analysis,* Artificial Intelligence. 13, pages 231 à 278, 1980.

[35] Roussel Philippe, Définition et traitement de l'égalité formelle en démonstration automatique, thèse de 3ième cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, mai 1972.

[36] Roussel Philippe, *Prolog, manuel de référence et d'utilisation,* Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, septembre 1975.

[37] Robinson J.A., *A machine-oriented logic based on the resolution principle,* J. ACM 12, 1, pages 23 à 41, janvier 1965.

[38] *TAUM 71,* rapport annuel du projet de Traduction Automatique de l'Université de Monréal, Université de Montréal, Janvier 1971.

[39] Warren David H. D.,Warplan, *A System for Generating Plans,* research report, University of Edimburgh, Department of Computational Logic, memo 76, juin 1974.

[40] Warren David H. D., Luis M. Pereira et Fernando Pereira, *Prolog the language and its implementation,* Proceedings of the ACM, Symposium on Artificial Intelligence and Programming Languages, Rochester, N.Y., août 1977.

[41] Wijngardeen(van) A., B. J. Mailloux, J. E. L Peck et G.H.A. Koster, *Final Draft Report on the Algorithmic Language Algol 68,* Mathematish Centrum, Amsterdam, décembre 1968.

[42] Winograd Terry, *Understanding Natural Language,* Academic Press, 1973.