VII

Prolog Session

Chair: *Randy Hudson*
Discussant: *Jacques Cohen*

THE BIRTH OF PROLOG

Alain Colmerauer

Faculté des Sciences de Luminy
163 avenue de Luminy
13288 Marseille cedex 9, France

Philippe Roussel

Université de Nice–Sophia Antipolis, CNRS, Bat 4
250 Avenue Albert Einstein
06560 Valbonne, France

ABSTRACT

The programming language, Prolog, was born of a project aimed not at producing a programm3ing language but at processing natural languages; in this case, French. The project gave rise to a preliminary version of Prolog at the end of 1971 and a more definitive version at the end of 1972. This article gives the history of this project and describes in detail the preliminary and then the final versions of Prolog. The authors also felt it appropriate to describe the Q-systems because it was a language that played a prominent part in Prolog's genesis.

CONTENTS

7.1 INTRODUCTION

As is well known, the name "Prolog" was invented in Marseilles in 1972. Philippe Roussel chose the name as an abbreviation for "PROgrammation en LOGique" to refer to the software tool designed to implement a man-machine communication system in natural language. It can be said that Prolog was the offspring of a successful marriage between natural language processing and automated theorem-

proving. The idea of using a natural language such as French to reason and communicate directly with a computer seemed like a crazy idea, yet this was the basis of the project set up by Alain Colmerauer in the summer of 1970. Alain had some experience in the computer processing of natural languages and wanted to expand his research.

We have now presented the two coauthors of this article—Alain Colmerauer and Philippe Roussel—but clearly, as in any project of this kind, many other people were involved. To remain objective in our account of how Prolog—a language that is now already twenty years old—came to be, we took a second look at all the documents still in our possession, and played the part of historians. To begin with, we followed the chronology in order to present the facts and describe the participants over the time from the summer of 1970 until the end of 1976. This constitutes the first part of the article. The other parts are of a more technical nature. They are devoted to three programming languages that rapidly succeeded one another: Q-systems, conceived for machine translation; the preliminary version of Prolog, created at the same time as its application; and the definitive version of Prolog, created independently of any application.

This paper is not the first to be written about the history of Prolog. We must mention the paper written by Jacques Cohen [1988], which directly concerns Prolog, and the paper by Robert Kowalski [1988] about the birth of "Logic Programming" as a discipline. Also worthy of attention are the history of automated theorem-proving as described by Donald Loveland [1984] and the prior existence of the Absys language, a possible competitor with Prolog in the opinion of E. Elcok [1988].

7.2 PART I. THE HISTORY

At the beginning of July 1970, Robert Pasero and Philippe arrived in Montreal. They had been invited by Alain who was then Assistant Professor of Computer Science at the University of Montreal and was leading the automatic translation project, TAUM (Traduction Automatique de l'Université de Montréal). All were at turning points in their careers. Robert and Philippe were then 25 years old and had just been awarded teaching positions in Computer Science at the new Luminy Science Faculty. Alain was 29 years old and, after a three-year stay in Canada, was soon to return to France.

During their two-month stay in Montreal, Robert and Philippe familiarized themselves with the computer processing of natural languages. They wrote several nondeterministic context-free analyzers in Algol 60 and a French paraphrase generator using Q-systems, the programming language which Alain had developed for the translation project (see Part II).

At the same time, Jean Trudel, a Canadian researcher and a doctoral student of Alain's, had chosen to work on automated theorem-proving. His point of reference was Alan Robinson's article [1965] on the resolution principle. It was a difficult article to understand in 1970, but Jean had the advantage of having taken a course in logic given by Martin Davis in New York. He had already developed a complete theorem prover in which unification was written in an up-to-date programming style: all the computations consisted in modifying pointers.

7.2.1 1971: The First Steps

At the beginning of 1971, everyone was back in Marseilles. Alain had obtained a teaching position in Computer Science (maître de conférence) and Jean Trudel had been able to accompany him, thanks to a two-year grant from Hydro-Quebec. The project was to make deductions based on texts written in French. The work was divided up as follows: Jean Trudel and Philippe worked on the deduction part, Robert Pasero and Alain on the natural language part.

We had access to a machine at the Marseilles university computing center: an IBM 360-44 installed in our department, offering approximately 900Kb of internal memory and equipped with an operating system without virtual memory. By means of an interactive monitor developed by Jean Trudel and by using the machine at night to take advantage of all the available memory, we were able to work under conditions which, in France at that time, were exceptionally good: practically 1Mb of memory to execute the programs and an interactive mode for communications between users and programs via the operator's console.

Jean Trudel improved his prover and then, beginning in May, Philippe produced a whole series of provers written in Algol-W. A primitive natural-language communication system was then developed by the entire team [Colmerauer 1971]. The interfaces between the logical formulæ and French consisted of 50 Q-system rules for the input and 17 Q-system rules for the output. The reasoning part was implemented through one of Philippe's provers. It was thus possible to have the following conversation with the computer:

```
User
Cats kill mice.
Tom is a cat who does not like mice who eat cheese.
Jerry is a mouse who eats cheese.
Max is not a mouse.
What does Tom do?
Computer
Tom does not like mice who eat cheese.
Tom kills mice.
User
Who is a cat?
Computer
Tom.
User
What does Jerry eat?
Computer
Cheese.
User
Who does not like mice who eat cheese?
Computer
Tom.
User
What does Tom eat?
Computer
What cats who do not like mice who eat cheese eat.
```

The logical formulæ created made use of: (i) constants representing elements,

```
Tom, Jerry, Max, Cheese;
```

(ii) constants representing sets,

```
Cats, Mice, MiceWhoEatCheese, CatsWhoDoNotLikeMiceWhoEatCheese;
```

(iii) constants representing binary relations between sets,

```
Kill, DoesNotLike, Eat;
```

(iv) a functional symbol of arity 1 and two relational symbols of arity 2 and 3,

```
The, Subset, True.
```

A term of the form `The(a)` was taken to represent the set consisting only of the element `a`. A formula of the form `Subset(x,y)` expressed the inclusion of set `x` in set `y` and a formula of the form `True(r,x,y)` expressed that the sets `x` and `y` were in the relation `r`. To the clauses that encoded the sentences, Jean Trudel added four clauses relating the three symbols `The`, `Subset`, `True`:

```
(∀x)[Subset(x,x)],
(∀x)( ∀y)(∀z)[Subset(x,y)∧Subset(y,z) ⇒ Subset(x,z)],
(∀a)(∀b)[Subset(The(a),The(b)) ⇒ Subset(The(b),The(a))],
(∀x)(∀y)(∀r)(∀x')(∀y')
[True(r,x,y)∧Subset(x,x')∧Subset(y,y') ⇒ True(r,x',y')].
```

The main problem was to avoid untimely production of inferences due to the transitivity and reflexivity axioms of the inclusion relation `Subset`.

While continuing his research on automated theorem-proving, Jean Trudel came across a very interesting method: SL-resolution [Kowalski 1971]. He persuaded us to invite one of its inventors, Robert Kowalski, who came to visit us for a week in June 1971. It was an unforgettable encounter. For the first time, we talked to a specialist in automated theorem-proving who was able to explain the resolution principle, its variants, and refinements. As for Robert Kowalski, he met people who were deeply interested in his research and who were determined to make use of it in natural language processing.

While attending an IJCAI convention in September 1971 with Jean Trudel, we met Robert Kowalski again and heard a lecture by Terry Winograd on natural language processing. The fact that he did not use a unified formalism left us puzzled. It was at this time that we learned of the existence of Carl Hewitt's programming language, Planner [Hewitt 1969]. The lack of formalization of this language, our ignorance of Lisp and, above all, the fact that we were absolutely devoted to logic meant that this work had little influence on our later research.

7.2.2 1972: The Application that Created Prolog

The year 1972 was the most fruitful one. First of all, in February, the group obtained a grant of 122,000 FF (at that time about $20,000) for a period of 18 months from the Institut de Recherche d'Informatique et d'Automatique, a computer research institution affiliated with the French Ministry of Industry. This contract made it possible for us to purchase a teletype terminal (30 characters per second) and to connect it to the IBM 360-67 (equipped with the marvelous operating system CP-CMS, which managed virtual machines) at the University of Grenoble using a dedicated 300-baud link. For the next three years, this was to be by far the most convenient computing system available to the team; everyone used it, including the many researchers who visited us. It took us several years to pay Grenoble back the machine-hour debt thus accumulated. Finally, the contract also enabled us to hire a secretary and a researcher, Henry Kanoui, a post-graduate student who would work on the French morphology. At his end, Kowalski obtained funding from NATO, which financed numerous exchanges between Edinburgh and Marseilles.

Of all the resolution systems implemented by Philippe, the SL-resolution of R. Kowalski and D. Kuehner seemed to be the most interesting. Its stack-type operating mode was similar to the management of procedure calls in a standard programming language and was thus particularly

well-suited to processing nondeterminism by backtracking à la Robert Floyd [1967] rather than by copying and saving the resolvents. SL-resolution then became the focus of Philippe's thesis on the processing of formal equality in automated theorem-proving [Roussel 1972]. Formal equality is less expressive than standard equality but it can be processed more efficiently. Philippe's thesis would lead to the introduction of the `dif` predicate (for $\neq$) into the very first version of Prolog.

We again invited Robert Kowalski but this time for a longer period: April and May. Together, we then all had more computational knowledge of automated theorem-proving. We knew how to axiomatize small problems (addition of integers, list concatenation, list reversal, etc.) so that an SL-resolution prover computed the result efficiently. We were not, however, aware of the Horn clause paradigm; moreover, Alain did not yet see how to do without Q-systems as far as natural language analysis was concerned.

After the departure of Robert, Alain ultimately found a way of developing powerful analyzers. He associated a binary predicate $N(x,y)$ with each nonterminal symbol N of the grammar, signifying that x and y are terminal strings for which the string u defined by $x = uy$ exists and can be derived from N. By representing x and y by lists, each grammar rule can then be encoded by a clause having exactly the same number of literals as occurrences of nonterminal symbols. It was thus possible to do without list concatenation. (This technique is now known as "The difference lists technique.") Alain also introduced additional parameters into each nonterminal to propagate and compute information. As in Q-systems, the analyzer not only verified that the sentence was correct but also extracted a formula representing the information that it contained. Nothing now stood in the way of the creation of a man-machine communication system entirely in "logic."

A draconian decision was made: at the cost of incompleteness, we chose linear resolution with unification only between the heads of clauses. Without knowing it, we had discovered the strategy that is complete when only Horn clauses are used. Robert Kowalski [1973] demonstrated this point later and together with Maarten van Emden, he would go on to define the modern fixed point semantics of Horn clause programming [Kowalski 1974].

During the fall of 1972, the first Prolog system was implemented by Philippe in Niklaus Wirt's language Algol-W; in parallel, Alain and Robert Pasero created the eagerly awaited man-machine communication system in French [Colmerauer 1972]. There was constant interaction between Philippe, who was implementing Prolog, and Alain and Robert Pasero, who programmed in a language that was being created step by step. This preliminary version of Prolog is described in detail in Part III of this paper. It was also at this time that the language received its definitive name following a suggestion from Philippe's wife based on keywords that had been given to her.

The man-machine communication system was the first large Prolog program ever to be written [Colmerauer 1972]. It had 610 clauses: Alain wrote 334 of them, mainly the analysis part; Robert Pasero 162, the purely deductive part, and Henry Kanoui wrote a French morphology in 104 clauses, which makes possible the link between the singular and plural of all common nouns and all verbs, even irregular ones, in the third person singular present tense. Here is an example of a text submitted to the man-machine communication system in 1972:

```
Every psychiatrist is a person.
Every person he analyzes is sick.
Jacques is a psychiatrist in Marseille.
Is Jacques a person?
Where is Jacques?
Is Jacques sick?
```

and here are the answers obtained for the three questions at the end:

```
Yes.
In Marseille.
I don't know.
```

The original text followed by the three answers was in fact as follows:

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

All the inferences were made from pronouns (he, she, they, etc.), articles (the, a, every, etc.), subjects and complement relations with or without prepositions (from, to, etc.). In fact, the system knew only about pronouns, articles, and prepositions (the vocabulary was encoded by 164 clauses); it recognized proper nouns from the mandatory asterisk that had to precede them as well as verbs and common nouns on the basis of the 104 clauses for French morphology.

In November, together with Robert Pasero, we undertook an extensive tour of the American research laboratories after a visit in Edinburgh. We took with us a preliminary report on our natural language communication system and our very first Prolog. We left copies of the report almost everywhere. Jacques Cohen welcomed us in Boston and introduced us at MIT, where we received a warm welcome and talked with Minsky, Charniak, Hewitt, and Winograd. We also visited Woods at BBN. We then went to Stanford, visited the SRI and John McCarthy's AI laboratory, met Cordell Green, presented our work to a very critical J. Feldman, and spent Thanksgiving at Robert Floyd's home.

7.2.3 1973: The Final Prolog

At the beginning of the year or, to be exact, in April, our group attained official status. The CNRS recognized us as an "associated research team" entitled "Man-machine dialogue in natural language," and provided financial support in the amount of 39,000 FF (about $6,500) for the first year. This sum should be compared to the 316,880 FF (about $50,000) we received in October from IRIA to renew the contract for "man-machine communication in natural language with automated deduction" for a period of two and a half years.

Users of the preliminary version of Prolog at the laboratory had now done sufficient programming for their experience to serve as the basis for a second version of Prolog, a version firmly oriented toward a programming language and not just a kind of automated deductive system. Besides the communication system in French in 1972, two other applications had been developed using this initial version of Prolog: a symbolic computation system [Bergman 1973a, 1973b; Kanoui 1973] and a general problem-solving system called Sugiton [Joubert 1974]. Also, Robert Pasero continued to use it for his work on French semantics, leading to the successful completion of his thesis in May [Pasero 1973].

Between February and April 1973, at the invitation of Robert Kowalski, Philippe visited the School of Artificial Intelligence at the University of Edinburgh, which was within the Department of Computational Logic directed by Bernard Meltzer. Besides the many discussions with the latter and with David Warren, Philippe also met Roger Boyer and Jay Moore. They had constructed an implementation of resolution using an extremely ingenious method based on a structure-sharing

technique to represent the logical formulæ generated during a deduction. The result of this visit and the laboratory's need to acquire a true programming language prompted our decision to lay the foundations for a second Prolog.

In May and June 1973, we laid out the main lines of the language, in particular the choice of syntax, basic primitives, and the interpreter's computing methods, all of which tended toward a simplification of the initial version. From June to the end of the year, Gérard Battani, Henry Meloni, and René Bazzoli, postgraduate students at the time, wrote the interpreter in FORTRAN and its supervisor in Prolog.

As the reader will see from the detailed description of this new Prolog in Part IV of this paper, all the new basic features of current Prologs were introduced. We observe in passing that this was also the time when the "occur check" disappeared as it was found to be too costly.

7.2.4 1974 and 1975: The Distribution of Prolog

The interactive version of Prolog which operated at Grenoble using teletype was in great demand. David Warren, who stayed with us from January to March, used it to write his plan generation system, Warplan [Warren 1974]. He notes:

> 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 and Marc Bergman used it to develop a symbolic manipulation system of quite some size called Sycophante [Bergman 1975; Kanoui 1976]. Gérard Battani and Henry Meloni used it to develop a speech recognition system enabling questions to be asked about the IBM operating system CP-CMS at Grenoble [Battani 1975; Meloni 1975]. The interface between the acoustic signal and the sequence of phonemes was borrowed from the CNET at Lannion and was obviously not written in Prolog.

Early in 1975, Alain Colmerauer had completely rewritten the supervisor keeping the infix operator declarations in Prolog but adding a compiler of the so-called "metamorphosis" grammars. This time, in contrast to René Bazzoli, he used a top-down analyzer to read the Prolog rules. This was a good exercise in metaprogramming. David Warren later included grammar rules of this sort in his compiled version of Prolog [Warren 1977] and together with Fernando Pereira rechristened a simplified variant of metamorphosis grammars with the name, "definite clause grammars" [Pereira 1980]. Metamorphosis grammars enabled parameterized grammar rules to be written directly as they were in Q-systems. The supervisor compiled these rules into efficient Prolog clauses by adding two additional parameters. To prove the efficiency and expressiveness of the metamorphosis grammars, Alain wrote a small model compiler from an Algol-style language to a fictitious machine language and a complete man-machine dialogue system in French with automated deductions. All this work was published in Colmerauer [1975], along with the theoretical foundations of the metamorphosis grammars.

Gérard Battani and Henry Meloni were kept very busy with the distribution of Prolog. They sent it to Budapest, Warsaw, Toronto, and Waterloo (Canada) and traveled to Edinburgh to assist David Warren in installing it on a PDP 10. A former student, Hélène Le Gloan, installed it at the University of Montreal. Michel Van Caneghem did the same at the IRIA in Paris before coming to work with us. Finally, Maurice Bruynooghe took Prolog to Leuven (Belgium) after a three-month stay in Marseilles (October through December 1975).

Indeed, as David Warren has pointed out, Prolog spread as much, or more, by people becoming interested and taking away copies either directly from Marseilles or from intermediaries such as Edinburgh. Thus, Prolog was not really distributed; rather it "escaped" and "multiplied."

During 1975, the whole team carried out the porting of the interpreter onto a 16-bit mini-computer: the T1600 from the French company, Télémécanique. The machine only had 64K bytes and so a virtual memory management system had to be specially written. Pierre Basso undertook this task and also won the contest for the shortest instruction sequence that performs an addressing on 32 bits while also testing the page default. Each laboratory member then received two pages of FORTRAN to translate into machine language. The translated fragments of program were reassembled and it worked! After five years, we at last had our very own machine and, what is more, our cherished Prolog ran; slowly, but it ran all the same.

7.3 PART II. A FORERUNNER OF PROLOG, THE Q-SYSTEMS

The history of the birth of Prolog thus comes to a halt at the end of 1975. We now turn to more technical aspects and, first of all, describe the Q-systems, the result of a first gamble: to develop a very high-level programming language, even if the execution times it entailed might seem bewildering [Colmerauer 1970b]. That gamble, and the experience acquired in implementing the Q-systems was determinative for the second gamble: Prolog.

7.3.1 One-Way Unification

A Q-system consists of a set of rewriting rules dealing with sequences of complex symbols separated by the sign +. Each rule is of the form

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

and means: in the sequence of trees we are manipulating, any subsequence of the form $e_1 + e_2 + ... + e_m$ can be replaced by the sequence $f_1 + f_2 + ... + f_n$. The e_is and the f_is are parenthesized expressions representing trees, with a strong resemblance to present Prolog terms but using three types of variables. Depending on whether the variable starts with a letter in the set {A,B,C,D,E,F}, {I,J,K,L,M,N}, or {U,V,W,X,Y,Z} it denotes either a label, a tree, or a (possibly empty) sequence of trees separated by commas. For example, the rule

$$P + A^*(X^*,I^*,Y^*) \rightarrow I^* + A^*(X^*,Y^*)$$

(variables are followed by an asterisk) applied to the sequence

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

produces three possible sequences

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

The concept of unification was therefore already present but it was one-way only; the variables appeared in the rules but never in the sequence of trees that was being transformed. However, unification took account of the associativity of the concatenation and, as in the preceding example, could produce several results.

7.3.2 Rule Application Strategy

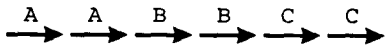
This relates to the unification part. Concerning the rule application strategy, Alain Colmerauer [1970b] wrote:

> It is difficult to use a computer to analyze a sentence. The main problem is combinatorial in nature: taken separately, each group of elements in the sentence can be combined in different ways with other groups to form new groups which can in turn be combined again and so on. Usually, there is only one correct way of grouping all the elements but to discover it, all the possible groupings must be tried. To describe this multitude of groupings in an economical way, I use an oriented graph in which each arrow is labeled by a parenthesized expression representing a tree. A Q-system is nothing more than a set of rules allowing such a graph to be transformed into another graph. This information may correspond to an analysis, to a sentence synthesis or to a formal manipulation of this type.

For example the sequence

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

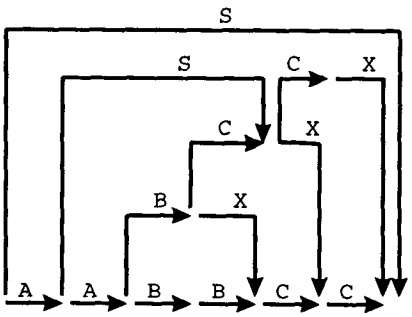
is represented by the graph



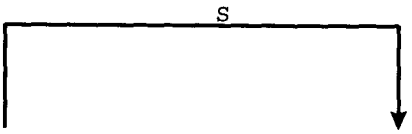
and the application of the four rules

```
A + B + C → S
A + S + X + C → S
X + C → C + X
B + B → B + X
```

produces the graph



One retains all the paths that lead from the entry point to the end point and do not contain any arrows used in the production of other arrows. One thus retains the unique arrow



that is, the sequence reduced to the single symbol S.

This procedure is relatively efficient, because it retains the maximum number of common parts in all the sequences. Another aspect of the Q-systems is that they can be applied one after the other. Each one takes as input the graph resulting from the previous system. This technique was widely used in the automatic translation project, where an English sentence would undergo no fewer than fifteen Q-systems before being translated into French. Two Q-systems dealt with morphology, another with the analysis of English, two more with the transfer from an English structure to a French structure, one with the synthesis of French, and nine with French morphology [TAUM 1971].

Let us draw attention to the reversibility of the Q-systems. The rewriting sign that we have represented by $\rightarrow$ was in fact written == and, depending on which specified option was chosen at the start of the program, it was interpreted as either a rewriting from left to right or from right to left. That is, the same program could be used to describe a transformation and its reverse transformation such as the analysis and the synthesis of a sentence.

It is interesting to note that in contrast to the analyzers written in Prolog that used a top-down strategy, the analyzers written in Q-systems used a bottom-up strategy. In fact, Alain had extensive experience with this type of strategy. The subject of his thesis done in Grenoble (France) had been bottom-up analyzers which, however, operated by backtracking [Colmerauer 1970a]. Nondeterminism was then reduced by precedence relations very similar to those of Robert Floyd [1963]. In addition, just before developing the Q-systems, and still as part of the automatic translation project, Alain had written an analyzer and a general synthesizer for W-grammars, the formalism introduced by A. van Wijngaarden to describe ALGOL 68 [van Wijngaarden 1968]. Here again a bottom-up analyzer was used to find the structure of complex symbols (defined by a metagrammar) as well as a second bottom-up analyzer for analysis of the text itself [Chastellier 1969].

7.3.3 Implementation

The Q-systems were written in ALGOL by Alain Colmerauer and were operational by October 1969. Michel van Caneghem and François Stellin, then completing a master's degree, developed a FORTRAN version, and Gilles Steward developed an ultra-fast version in machine language for the CDC 6400 computer of the University of Montreal.

These Q-systems were used by the entire TAUM project team to construct a complete chain of automatic English-French translations. The English morphology was written by Brian Harris, Richard Kittredge wrote a substantial grammar for the analysis of English, Gilles Stewart wrote the transfer phase, Jules Danserau wrote the grammar for the French synthesis, and Michel van Caneghem developed a complete French morphology [Taum 1971]. The Q-systems were also used a few years later to write the METEO system, a current industrial version of which produces daily translations of Canadian weather forecasts from English into French.

7.4 PART III. THE PRELIMINARY PROLOG

Let us now turn to the preliminary version of Prolog which was created in the fall of 1972 in connection with the development of the man-machine communication system [Colmerauer 1972].

Recall that the objective we had set for ourselves was extremely ambitious: to have a tool for the syntactic and semantic analysis of natural language, by considering first order logic not only as a programming language but also as a knowledge representation language. In other words, the logical formulation was to serve not only for the different modules of the natural language dialogue system but also for the data exchanged between them, including the data exchanged with the user.

The choice having settled on first order logic (in clausal form) and not on a higher order logic, it then seemed that we were faced with an insurmountable difficulty in wanting programs to be able to manipulate other programs. Of course, this problem was solved using nonlogical mechanisms. In fact, this initial version of Prolog was conceived more as an application tool than as a universal programming language. Nevertheless, the basis for such a language was already there.

7.4.1 Reasons for the Choice of Resolution Method

The success of the project hinged on the decision concerning the choice of the logic system and on the basic inference mechanism to be adopted. Although Robinson's resolution principle naturally suggested itself by the simplicity of its clausal form, the uniqueness of the inference rule, and its similarity with procedure calls in standard languages, it was difficult to decide what type of adaptation was necessary to fulfill our requirements. Among the considerations to be taken into account were the validity and logical completeness of the system, the problems for implementation on the machine, and, especially, the risks of combinatorial explosion, which we were well aware of from our experiments.

Among the question-answering systems and the problem-solving techniques that we had explored, there were those of D. Luckam [1971] and N.J. Nilson, of J.L. Darlington [1969] and of Cordell Green [1969]. Our exploration, the tests by Jean Trudel and Robert Pasero using experimental versions of Philippe's provers, Alain's research on the logical formulation of grammars and numerous discussions with Robert Kowalski: all these elements led us to view the resolution principle from a point of view different from the prevailing one. Rather than demonstrating a theorem by reduction ad absurdum, we wanted to calculate an "interesting" set of clauses that were deducible from a given set of clauses. Because, to our way of thinking, such sets constituted programs, we would thus have programs generating other programs. This idea constantly underlay the conception of this preliminary version of the language as it did the realization of the application.

The final choice was an adaptation of the resolution method similar to those of the subsequent Prologs but comprising some very novel elements, even compared with modern Prologs. Each execution was performed with a set of clauses constituting the "program" and a set of clauses constituting the "questions." Both of them produced a set of clauses constituting the "answers." The literals of the clauses were ordered from left to right and the resolution was performed between the head literal of the resolvent and the head literal of one of the program clauses. The novelty resided in the fact that in each clause a part of the literals, separated by the "/" sign, was not processed during the proof. Instead, they were accumulated to produce one of the answer clauses at the end of deduction. In addition, certain predicates (such as `DIF`) were processed by delayed evaluation and could also be transmitted as an answer. Finally, it was decided that nondeterminism should be processed by backtracking, meaning that only a single branch of the search tree was stored at any given time in the memory.

Formally, the chosen deduction method can be described by the three deduction rules below, where "question," "chosen clause," and "answer" denote three clauses taken respectively from the sets "questions," "program," and "answers" and "resolvent" denotes the current list of goals.

Deduction initialization rule

question: $L_1 \ldots L_m$ / $R_1 \ldots R_n$

---

resolvent: $L_1 \ldots L_m$ / $R_1 \ldots R_n$

Basic deduction rule

resolvent: $L_0\ L_1 \ldots\ /\ R_1 \ldots R_n$, chosen clause: $L'_0\ L'_1 \ldots L'_{m'}\ /\ R'_1 \ldots R'_{n'}$

resolvent $\sigma(L'_1) \ldots \sigma(L'_{m'})\sigma(L_1) \ldots \sigma(L_m)\ /\ \sigma(R'_1) \ldots \sigma(R'_{n'})\sigma(R_1) \ldots \sigma(R_n)$

End of deduction rule

resolvent: $/\ R_1 \ldots R_n$

answer: $R_1 \ldots R_n$

where of course, L_0 and L'_0 are complementary unifiable literals and s is the most general substitution that unifies them. (An example is given later.)

The first reason for choosing this linear resolution technique with a predefined order of literal selection was its simplicity and the fact that we could produce clauses that were logically deducible from the program, which thus guaranteed, in a way, the validity of the results. To a large extent, we were inspired by Robert Kowalski's SL-Resolution which Philippe had implemented for his thesis on formal equality. However, despite its stack-like functioning analogous to procedure calling in standard languages, we knew that this method introduced computations that were certainly necessary in the general case but unnecessary for most of our examples. We therefore adopted the extremely simplified version of SL-Resolution described by the preceding three rules, which continues to serve as the basis for all Prologs.

The choice of the treatment of nondeterminism was basically a matter of efficiency. By programming a certain number of methods, Philippe had shown that one of the crucial problems was combinatorial explosion and a consequent lack of memory. Backtracking was selected early on for management of nondeterminism, in preference to a management system of several branch calculations simultaneously resident in memory, whose effect would have been to considerably increase the memory size required for execution of the deductions. Alain had a preference for this method, introduced by Robert Floyd [1967] to process nondeterministic languages, and was teaching it to all his students. Although, certainly, the use of backtracking led to a loss of completeness in deductions comprising infinite branches, we felt that, given the simplicity of the deduction strategy (the execution of literals from left to right and the choice of clauses in the order they were written), it was up to the programmer to make sure that the execution of his or her program terminated.

7.4.2 Characteristics of the Preliminary Prolog

Apart from the deduction mechanism we have already discussed, a number of built-in predicates were added to the system as Alain and Robert Pasero required them: predicates to trace an execution, `COPY` to copy a term, `BOUM` to split an identifier into a list of characters or reconstitute it and `DIF` to process the symbolic (that is, syntactic) equality of Philippe's thesis. It should be noted that we refused to include input-output predicates in this list as they were considered to be too far removed from logic. Input-output, specification of the initial resolvents, and chaining between programs were specified in a command language applied to sets of clauses (read, copy, write, merge, prove, etc.). This language, without any control instructions, allowed chainings to be defined only statically but had the virtue of treating communications uniformly by way of sets of clauses. It should be emphasized that this first version already included lazy evaluation (or coroutined evaluation, if you prefer) of certain predicates; in this case `DIF` and `BOUM`. The `DIF` predicate was abandoned in the next version but reappeared in modern Prologs. The only control operators were placed at the end of clauses as punctuation marks, and their function was to perform cuts in the search space. The annotations:

.. performed a cut after the head of the clause,
.; performed a cut after execution of the whole rule,
;. performed a cut after production of at least one answer,
;; had no effect.

These extra-logical operators were exotic enough to cause their users problems and were therefore subsequently abandoned. Curiously, this punctuation had been introduced by Alain following his discovery of the optional and mandatory transformation rules of the linguist, Noam Chomsky [1965].

On the syntactic level, the terms were written in functional form although it was possible to introduce unary or binary operators defined by precedences as well as an infixed binary operator that could be represented by an absence of sign (like a product in mathematics and very useful in string input-output). Here is an example of a sequence of programs that produced and wrote a set of clauses defining the great-nephews of a person named MARIE :

```
READ
   RULES
   +DESC(*X,*Y) -CHILD(*X,*Y);;
   +DESC(*X,*Z) -CHILD(*X,*Y) -DESC(*Y,*Z);;
   +BROTHERSISTER(*X,*Y) -CHILD(*Z,*X) -CHILD(*Z,*Y) -DIF(*X,*Y);;
   AMEN

READ
   FACTS
   +CHILD(PAUL,MARIE);;
   +CHILD(PAUL,PIERRE);;
   +CHILD(PAUL,JEAN);;
   +CHILD(PIERRE,ALAIN);;
   +CHILD(PIERRE,PHILIPPE);;
   +CHILD(ALAIN,SOPHIE);;
   +CHILD(PHILIPPE,ROBERT);;
   AMEN

READ
 QUESTION
 -BROTHERSISTER(MARIE,*X) -DESC(*X,*Y) / +GREATNEPHEW(*Y) -MASC(*Y)..
 AMEN

CONCATENATE(FAMILYTIES,RULES,FACTS)

PROVE(FAMILYTIES,QUESTION,ANSWER)

WRITE(ANSWER)

AMEN
```

The output from this program was thus not a term but instead the following set of binary clauses:

```
+GREATNEPHEW(SOPHIE) -MASC(SOPHIE);.
+GREATNEPHEW(ROBERT) -MASC(ROBERT);.
```

The READ command read a set of clauses preceded by a name x and ending with AMEN and assigned it the name x. The command CONCATENATE(y,x_1,...,x_n) computed the union y of the sets of clauses

$x_1,...,x_n$. The command PROVE(x,y,z) was the most important one; it started a procedure where the program is x, the initial resolvent is y, and the set of answers is z. Finally, the command WRITE(x) printed the set of clauses x.

The man-machine communication system operated in four phases and made use of four programs, that is, four sets of clauses:

- C1 to analyze a text T0 and produce a deep structure T1,
- C2 to find in T1 the antecedents of the pronouns and produce a logical form T2,
- C3 to split the logical formula T2 into a set T3 of elementary information,
- C4 to carry out deductions from T3 and produce the answers in French T4.

The sequence of commands was therefore

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

where T0, the French text to process, and T4, the answers produced, were represented by elementary facts over lists of characters.

7.4.3 Implementation of the Preliminary Prolog

Because the Luminy computing center had been moved, the interpreter was implemented by Philippe in ALGOL-W, on the IBM 360-67 machine of the University of Grenoble computing center, equipped with the CP-CMS operating system based on the virtual machine concept. We were connected to this machine via a special telephone line. The machine had two unique characteristics almost unknown at this time, characteristics that were essential for our work: it could provide a programmer with a virtual memory of 1Mb if necessary (and it was), and it allowed us to write interactive programs. So it was, that, on a single console operating at 300 baud, we developed not only the interpreter but also the question-answering system itself. The choice of ALGOL-W was imposed on us, because it was the only high-level language we had available that enabled us to create structured objects dynamically, while also being equipped with garbage collection.

The basis for the implementation of the resolution was an encoding of the clauses into interpointing structures with anticipated copying of each rule used in a deduction. Nondeterminism was managed by a backtracking stack and substitutions were performed only by creating chains of pointers. This approach eliminated the copying of terms during unifications, and thus greatly improved the computing times and memory space used. The clause analyzer was also written in ALGOL-W, in which the atoms were managed by a standard "hash-code" technique. This analyzer constituted a not insignificant part of the system which strengthened Alain's desire to solve these syntax problems in Prolog itself. However, experience was still lacking on this topic, because the purpose of the first application was to reveal the very principles of syntactic analysis in logic programming.

7.5 PART IV. THE FINAL PROLOG

Now that we have described the two forerunners at length, it is time to lay out the fact sheet on the definitive Prolog of 1973. Our major preoccupation after the preliminary version was the reinforcement of Prolog's programming language aspects by minimizing concepts and improving its interactive

capabilities in program management. Prolog was becoming a language based on the resolution principle alone and on the provision of a set of built-in predicates (procedures) making it possible to do everything in the language itself. This set was conceived as a minimum set enabling the user to:

- create and modify programs in memory,
- read source programs, analyze them and load them in memory,
- interpret queries dynamically with a structure analogous to other elements of the language,
- have access dynamically to the structure and the elements of a deduction,
- control program execution as simply as possible.

7.5.1 Resolution Strategy

The experience gained from the first version led us to use a simplified version of its resolution strategy. The decision was based not only on suggestions from the first programmers but also on criteria of efficiency and on the choice of FORTRAN to program the interpreter which forced us to manage the memory space. The essential differences with the previous version were:

- no more delayed evaluation (`DIF`, `BOUM`),
- replacement of the `BOUM` predicate by the more general `UNIV` predicate,
- the operations assert and retract, at that time written `AJOUT` and `SUPP`, are used to replace the mechanism that generates clauses as the result of a deduction,
- a single operator for backtracking management, the search space cut operator "!" , at that time written "*/*",
- the meta-call concept to use a variable instead of a literal,
- use of the predicates `ANCESTOR` and `STATE`, which have disappeared in present Prologs, to access ancestor literals and the current resolvent (considered as a term), for programmers wishing to define their own resolution mechanism.

Backtracking and ordering the set of clauses defining a predicate were the basic elements retained as a technique for managing nondeterminism. The preliminary version of Prolog was quite satisfactory in this aspect. Alain's reduction of backtracking control management to a single primitive (the cut), replacing the too numerous concepts in the first version, produced an extraordinary simplification of the language. Not only could the programmer reduce the search space size according to purely pragmatic requirements but also he or she could process negation in a way, which, although simplified and reductive in its semantics, was extremely useful in most common types of programming.

In addition, after a visit to Edinburgh, Philippe had in mind the basis for an architecture that is extremely simple to implement from the point of view of memory management and much more efficient in terms of time and space, if we maintained the philosophy of managing nondeterminism by backtracking. In the end, all the early experiences of programming had shown that this technique allowed our users to incorporate nondeterminism fairly easily as an added dimension to the control of the execution of the predicates.

Concerning the processing of the implicit "or" between literals inside a clause, sequentiality imposed itself there again as the most natural interpretation of this operator because, in formal terms, the order of goal execution has no effect at all on the set of results (modulo the pruning of infinite branches), as Robert Kowalski had proved concerning SL-resolution.

To summarize, these two choices concerning the processing of "and" and "or", were fully justified by the required objectives:

- employ a simple and predictable strategy that the user can control, enabling any extra-logical predicate (such as input-output) to be given an operational definition,
- provide an interpreter capable of processing deductions with thousands or tens of thousands of steps (an impossible objective in the deductive systems existing at that time).

7.5.2 Syntax and Primitives

On the whole, the syntax retained was the same as the syntax of the preliminary version of the language. On the lexical level, the identifier syntax was the same as that of most languages and therefore lower-case letters could not be used (the keyboards and operating systems at that time did not systematically allow this). It should be noted that among the basic primitives for processing morphology problems, one single primitive `UNIV` was used to create dynamically an atom from a character sequence, to construct a structured object from its elements, and, conversely, to perform the inverse splitting operations. This primitive was one of the basic tools used to create programs dynamically and to manipulate objects whose structures were unknown prior to the execution of the program.

Enabling the user to define his own unary and binary operators by specifiying numeric precedences proved very useful and flexible although it complicated somewhat the clause analyzers. It still survives as such in the different current Prologs.

In the preliminary version of Prolog, it was possible to create clauses that were logically deducible from other clauses. Our experience with this version had showed us that sometimes it was necessary to manipulate clauses for purposes very far removed from first order logic: modeling of temporal type reasoning, management of information persisting for an uncertain lifetime, or simulation of exotic logics. We felt that much research would still be needed in the area of semantics in order to model the problems of updating sets of clauses. Hence, we made the extremely pragmatic decision to introduce extra-logical primitives acting by side effect to modify a program (`ADD`, `DELETE`). This choice seems to have been the right one because these functions have all been retained.

One of the missing features of the preliminary Prolog was a mechanism that could compute a term that could then be taken as a literal to be resolved. This is an essential function needed for metaprogramming such as a command interpreter; this feature is very easy to implement from a syntactic point of view. In any event, a variable denoting a term can play the role of a literal.

In the same spirit—and originally intended for specialists in computational logic—various functions giving access to the current deduction by considering it as a Prolog object appeared in the basic primitives (`STATE`, `ANCESTOR`). Similarly, the predicate " / " (pronounced "cut " by Edinburghers) became parametrizable in a very powerful manner by access to ancestors.

7.5.3 A Programming Example

To show the reader what an early Prolog program looked like, we introduce an old example dealing with flights between cities. From a base of facts that describes direct flights, the program can calculate routes which satisfy some scheduling constraints.

Direct flights are represented by unary clauses under the following format:

```
+FLIGHT(<departure city>,<arrival city>,
        <departure time>,<arrival time>,<flight identifier>)
```

where time schedules are represented by pairs of integers under the format <hours>:<minutes>. All flights are supposed to be completed in the same day.

The following predicate will be called by the user to plan a route.

```
PLAN(<departure city>,<arrival city>,<departure time>,<arrival time>,
     <departure min time>,<arrival max time>)
```

It enumerates (and outputs as results) all pairs of cities connected by a route that can be a direct flight or a sequence of flights. Except for circuits, the same city cannot be visited more than once. Parameters <departure min time> and <arrival max time> denote constraints given by the user about departure and arrival times. The first flight of the route should leave after <departure min time> and the last one should arrive before <arrival max time>.

In order to calculate PLAN, several predicates are defined. The predicate:

```
ROUTE(<departure city>,<arrival city>,<departure time>,<arrival time>,
    <plan>, <visits>, <departure mini time>,<arrival maxi time>)
```

is similar to the PLAN predicate, except for two additional parameters: the input parameter <visits> is given represents the list (Ck.Ck-1...C1.NIL) of already visited cities (in inverse order), the output parameter <plan> is the list (F1....Fk.NIL) of calculated flight names. The predicates BEFORE(<t1>,<t2>) and ADDTIMES(<t1>,<t2>,<t3>) deal with arithmetic aspects of time schedules. The predicate WRITEPLAN writes the sequence of flight names. Finally, NOT(<literal>) defines negation by failure, and ELEMENT(<element>,<elements) succeeds if <element> is among the list <elements>.

Here then is the complete program, including data and the saving and execution commands.

```
* INFIXED OPERATORS
--------------------.
-AJOP( ".", 1, "(X|X)|X")  -AJOP(":",  2, "(X|X)|X") !
*  USER PREDICATE
-----------------.
+PLAN(*DEPC, *ARRC, *DEPT, *ARRT, *DEPMINT, *ARRMAXT)
   -ROUTE(*DEPC, *ARRC, *DEPT, *ARRT, *PLAN, *DEPC.NIL, *DEPMINT,
                                                        *ARRMAXT)
   -SORM("-------------------------------------------------")
   -LIGNE
   -SORM("FLYING ROUTE BETWEEN:  ")
   -SORT(*DEPC)
   -SORM(" AND: ")
   -SORT(*ARRC)
   -LIGNE
   -SORM("-------------------------------------------------")
   -LIGNE
   -SORM(" DEPARTURE TIME:   ")
   -SORT(*DEPT)
   -LIGNE
   -SORM("ARRIVAL TIME:      ")
   -SORT(*ARRT)
   -LIGNE
   -SORM(" FLIGHTS:    ")
   -WRITEPLAN(*PLAN) -LIGNE -LIGNE.
* PRIVATE PREDICATES
```

```
--------------------.
+ROUTE(*DEPC, *ARRC, *DEPT, *ARRT, *FLIGHTID.NIL, *VISITS, *DEPMINT,
                                                          *ARRMAXT)R
-FLIGHT(*DEPC, *ARRC, *DEPT, *ARRT, *FLIGHTID)
   -BEFORE(*DEPMINT, *DEPT)
   -BEFORE(*ARRT, *ARRMAXT).
+ROUTE(*DEPC,*ARRC, *DEPT,*ARRT, *FLIGHTID.*PLAN, *VISITS,*DEPMINT,
                                                        *ARRMAXT)
   -FLIGHT(*DEPC, *INTC, *DEPT, *INTT, *FLIGHTID)
   -BEFORE(*DEPMINT, *DEPT)
   -ADDTIMES(*INTT, 00:15, *INTMINDEPT)
   -BEFORE(*INTMINDEPT, *ARRMAXT)
   -NOT( ELEMENT(*INTC, *VISITS)
   -ROUTE(*INTC,*ARR,*INTDEPT,*HARR, *PLAN, *INTC.*VISITS,
                                            *INTMINDEPT, *ARRMAXT).

+BEFORE(*H1:*M1, *H2:*M2) -INF(H1, H2).
+BEFORE(*H1:*M1, *H1:*M2) -INF(M1, M2).
+ADDTIMES(*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).
+WRITEPLAN( *X. NIL) -/ -SORT(*X).
+WRITEPLAN( *X.*Y) -SORT(*X) -ECRIT(-) -WRITEPLAN(*Y).
+ELEMENT(*X, *X.*Y).
+ELEMENT(*X, *Y.*Z) -ELEMENT(*X, *Z).
+NOT(*X) -*X -/ -FAIL.
+NOT(*X).
* LIST OF FLIGHTS
-----------------.
+FLIGHT(PARIS, LONDON, 06:50, 07:30, AF201).
+FLIGHT(PARIS, LONDON, 07:35, 08:20, AF210).
+FLIGHT(PARIS, LONDON, 09:10, 09:55, BA304).
+FLIGHT(PARIS, LONDON, 11:40, 12:20, AF410).
+FLIGHT(MARSEILLES, PARIS, 06:15, 07:00, IT100).
+FLIGHT(MARSEILLES, PARIS, 06:45, 07:30, IT110).
+FLIGHT(MARSEILLES, PARIS, 08:10, 08:55, IT308).
+FLIGHT(MARSEILLES, PARIS, 10:00, 10:45, IT500).
+FLIGHT(MARSEILLES, LONDON, 08:15, 09:45, BA560).
+FLIGHT(MARSEILLES, LYON, 07:45, 08:15, IT115).
+FLIGHT(LYON, LONDON, 08:30, 09:25, TAT263).
* SAVING THE PROGRAM
--------------------.
-SAUVE!
* QUERYING
----------.
-PLAN(MARSEILLES, LONDON, *HD, *HA, 00:00, 09:30)!
```

This is the ouput of the program:

```
---------------------------------------------
 FLYING ROUTE BETWEEN:   MARSEILLES  AND: LONDON
---------------------------------------------
DEPARTURE TIME:     06:15
ARRIVAL TIME:      08:20
FLIGHTS:    IT100-AF210

---------------------------------------------
 FLYING ROUTE BETWEEN:   MARSEILLES  AND: LONDON
---------------------------------------------
DEPARTURE TIME:     07:45
ARRIVAL TIME:      09:25
FLIGHTS:    IT115-TAT263
```

7.5.4 Implementation of the Interpreter

The resolution system, in which nondeterminism was managed by backtracking, was implemented using a very novel method for representing clauses, halfway between the technique based on structure sharing used by Robert Boyer and Jay Moore in their work on the proofs of programs [Boyer 1972; Moore 1974] and the backtracking technique used in the preliminary version of Prolog. Philippe came up with this solution during his stay in Edinburgh after many discussions with Robert Boyer. In this new approach, the clauses of a program were encoded in memory as a series of templates, which can be instantiated without copying, several times in the same deduction, by means of contexts containing the substitutions to be performed on the variables. This technique had many advantages compared to those normally used in automated theorem-proving:

- in all known systems, unification was performed in times that were, at best, linear in relation to the size of the terms unified. In our system, most of the unifications could be performed in constant time, determined not by the size of the data, but by that of the templates brought into action by the clauses of the called program. As a result, the concatenation of two lists was performed in a linear time corresponding to the size of the first and not in quadratic time as in all other systems based on copying techniques;
- in the same system, the memory space required for one step in a deduction is not a function of the data, but of the program clause used. Globally therefore, the concatenation of two lists used only a quantity of memory space proportional to the size of the first list;
- the implementation of nondeterminism did not require a sophisticated garbage collector in the first approach but simply the use of several stacks synchronized on the backtracking, thus facilitating rapid management and yet remaining economical in the use of interpreter memory.

Concerning the representation of the templates in memory, we decided to use prefix representation (the exact opposite of Polish notation). The system consisted of the actual interpreter (i.e., the inference machine equipped with a library of built-in predicates), a loader to read clauses in a restricted syntax, and a supervisor written in Prolog. Among other things, this supervisor contained a query evaluator, an analyzer accepting extended syntax, and the high level input-output predicates.

Alain, who like all of us disliked FORTRAN, succeeded nonetheless in persuading the team to program the interpreter in this language. This basic choice was based primarily on the fact that

FORTRAN was widely distributed on all machines and that the machine we had access to at that time supported no other languages adapted to our task. We hoped in that way to have a portable system, a prediction that proved to be quite correct.

Under Philippe's supervision, Gérard Battani [Battani 1973] and Henri Meloni developed the actual interpreter between June 1973 and October 1973 on a CII 10070 (variant of the SIGMA 7) while René Bazzoli, under Alain's direction, was given the task of writing the supervisor in the Prolog language itself. The program consisted of approximately 2000 instructions, of roughly the same size as the ALGOL-W program of the initial version.

The machine had a batch operating system with no possibility of interaction via a terminal. Hence, data and programs were entered by means of punched cards. That these young researchers could develop as complex a system as this under such conditions and in such short time is especially remarkable in light of the fact that none of them had ever written a line of FORTRAN before in their lives. The interpreter was finally completed in December 1973 by Gérard Battani and Henry Meloni after porting it onto the IBM 360-67 machine at Grenoble, thus providing somewhat more reasonable operating conditions. Philippe Roussel wrote the reference and user's manual for this new Prolog two years later [Roussel 1975].

7.6 CONCLUSION

After all these vicissitudes and all the technical details, it might be interesting to take a step back and to place the birth of Prolog in a wider perspective. The article published by Alan Robinson in January 1965, "A machine-oriented logic based on the resolution principle," contained the seeds of the Prolog language. This article was the source of an important stream of works on automated theorem-proving and there is no question that Prolog is essentially a theorem prover "à la Robinson."

Our contribution was to transform that theorem prover into a programming language. To that end, we did not hesitate to introduce purely computational mechanisms and restrictions that were heresies for the existing theoretical model. These modifications, so often criticized, assured the viability, and thus the success, of Prolog. Robert Kowalski's contribution was to single out the concept of the "Horn clause," which legitimized our principal heresy: a strategy of linear demonstration with backtracking and with unifications only at the heads of clauses.

Prolog is so simple that one has the sense that sooner or later someone had to discover it. Why did we discover it rather than anyone else? First of all, Alain had the right background for creating a programming language. He belonged to the first generation of Ph.D.s in computer science in France and his specialty was language theory. He had gained valuable experience in creating his first programming language, Q-systems, while on the staff of the machine translation project at the University of Montreal. Then, our meeting, Philippe's creativity, and the particular working conditions at Marseilles did the rest. We benefitted from freedom of action in a newly created scientific center and, having no outside pressures, we were able to devote ourselves fully to our project.

Undoubtedly, this is why that period of our lives remains one of the happiest in our memories. We have had the pleasure of recalling it for this paper over fresh almonds accompanied by a dry martini.

ACKNOWLEDGMENTS

We would like to thank all those who contributed to the English version of this paper: Andy Tom, Franz Günther, Mike Mahoney, and Pamela Morton.

BIBLIOGRAPHY

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

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

[Bergman, 1973a] Bergman, Marc and Henry Kanoui, Application of mechanical theorem proving to symbolic calculus, *Third International Colloquium on Advanced Computing Methods in Theoretical Physics*, Marseilles, France, June 1973.

[Bergman, 1973b] *Résolution par la démonstration automatique de quelques problèmes en intégration symbolique sur calculateur*, 3ème cyle thesis, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Oct. 1973.

[Bergman, 1975] Bergman, Marc and Henry Kanoui, *SYCOPHANTE, système de calcul formel sur ordinateur*, final report for a DRET contract (Direction des Recherches et Etudes Techniques), Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, 1975.

[Boyer, 1972] Boyer, Roger S. and Jay S. Moore, The sharing of structure in theorem proving programs, *Machine Intelligence 7*, edited by B. Melzer and D. Michie, New York: Edinburgh University Press, 1972, pp. 101–116.

[Chastellier, 1969] Chastellier (de), Guy and Alain Colmerauer, W-Grammar. *Proceedings of the ACM Congress*, San Francisco, Aug., New York: ACM, 1969, pp. 511–518.

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

[Cohen, 1988] Cohen, Jacques, A view of the origins and development of Prolog, *Commun. ACM 31*, 1, Jan. 1988, pp. 26–36.

[Colmerauer, 1970a] Colmerauer, Alain, Total precedence relations, *J. ACM 17, 1*, Jan. 1970, pp. 14–30.

[Colmerauer, 1970b] 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, Sept. 1970.

[Colmerauer, 1971] Colmerauer, Alain, Fernand Didier, Robert Pasero, Philippe Roussel, Jean Trudel, *Répondre à*, Internal publication, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, May 1971. This publication is a computer print-out with handwritten remarks.

[Colmerauer, 1972] Colmerauer, Alain, Henry Kanoui, Robert Pasero and 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, Oct. 1972.

[Colmerauer, 1975] Colmerauer, Alain, *Les grammaires de métamorphose GIA*, Internal publication, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Nov. 1975. English version, Metamorphosis grammars, *Natural Language Communication with Computers, Lectures Notes in Computer Science 63*, edited by L. Bolc, Berlin, Heidelberg, New York: Springer Verlag, 1978, pp. 133–189.

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

[Elcok, 1988] Elcok, E., W. Absys: the first logic programming language—A restrospective and a commentary. *The Journal of Logic Programming*.

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

[Floyd, 1967] Floyd, Robert W., Nondeterministic algorithms. *J. ACM 14, 4*, Oct. 1967, pp. 636–644.

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

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

[Joubert, 1974] Joubert, Michel, *Un système de résolution de problèmes à tendance naturelle*, 3ème cycle thesis, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Feb. 1974.

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

[Kanoui, 1976] Kanoui, Henry, Some aspects of symbolic integration via predicate logic programming. *ACM SIGSAM Bulletin*, 1976.

[Kowalski, 1971] Kowalski, Robert A. and D. Kuehner, *Linear resolution with selection function*, memo 78, University of Edinburgh, School of Artificial Intelligence, 1971. Also in *Artificial Intelligence 2, 3*, pp. 227–260.

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

[Kowalski, 1974] Kowalski, Robert A. and Maarten van Emden, *The semantic of predicate logic as programming language*, memo 78, University of Edinburgh, School of Artificial Intelligence, 1974. Also in *JACM 22*, 1976, pp. 733–742.

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

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

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

[Meloni, 1975] Meloni, Henry, *Mise en oeuvre 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, June 1975.

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

[Pasero, 1973] 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, May 1973.

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

[Robinson, 1965] Robinson, J.A., A machine-oriented logic based on the resoulution principle, *J. ACM 12, 1*, Jan. 1965, pp. 23–41.

[Roussel, 1972] 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, May 1972.

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

[Taum, 1971] *TAUM 71*, Annual report, Projet de Traduction Automatique de l'Université de Montréal, Jan. 1971.

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

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

[Wijngaarden, 1968] van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, and G. H. A. Koster, *Final Draft Report on the Algorithmic Language Algol 68*, Mathematish Centrum, Amsterdam, Dec. 1968.

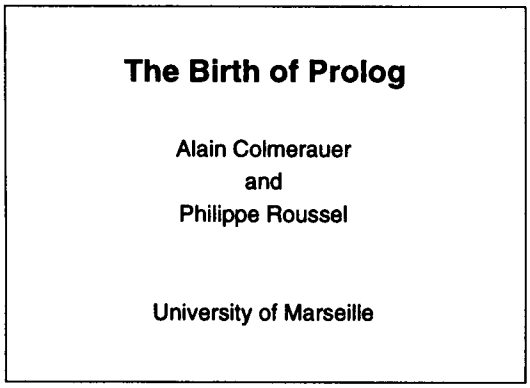
TRANSCRIPT OF PRESENTATION

ALAIN COLMERAUER: (SLIDE 1) I was supposed to give this talk with Philippe Roussel but he could not come. I won't speak so much about Prolog but about the circumstances which brought about the birth of the language. Unlike other programming languages, there was no institution which decided to create Prolog. Prolog was born in a spontaneous way, as a software tool for natural language processing.

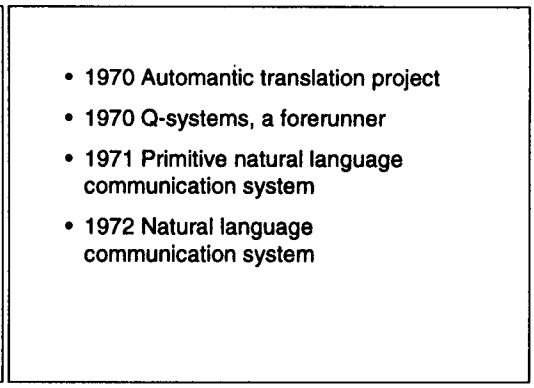
These are the succesive topics of my talk:

(SLIDE 2) After my PhD in Grenoble, I went, as Assistant Professor, to the University of Montreal where I got involved in an automatic translation project. It was within the framework of this project that I developed a software tool, called "Q-systems", which in fact turned out to be a forerunner of Prolog. The letter "Q" stood for Quebec. At the end of the 70s, I went back to France where I got a position as Professor in Marseilles. There I met Philippe Roussel and we started to develop a natural language communication system involving logical inferences.

After a first "primitive" system, we developed a more sophisticated one, which led us to design and implement a preliminary version of Prolog, which I will call, "Prolog 0." We then developed a full version of the language, which I will call "Prolog 1". This version had a wide distribution. After describing all this in detail, I will conclude by giving information about direct and indirect contributions of various people to the birth of Prolog.



SLIDE 1



SLIDE 2

(SLIDES 3 and 4) First let us talk about the automatic translation project which was given the name TAUM, for "Traduction Automatique Université de Montréal". We were supposed to translate texts from English into French, and here is a typical example of a sentence to be translated.

(SLIDES 5 and 6) Slide 5 shows the input and slide 6 shows the output. The input sentence was translated using Q-sytems rules which were essentially rewriting rules on sequences of complex symbols.

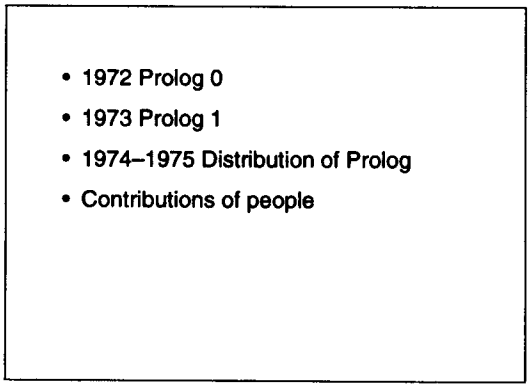
(SLIDE 7) You would take the input sentence and cut it into pieces separated by plus signs. The "–1–" indicated the beginning of the sentence and "–2–" the end. Then you would apply rules on subsequences of elements. You would apply a first set of rules and then you would obtain this.

(SLIDE 8) "The" has been transformed into a definite article, "expansion" is a noun and so on. This makes up the first step in morphology. Then you would add another phase and you would end up with just one complex symbol.

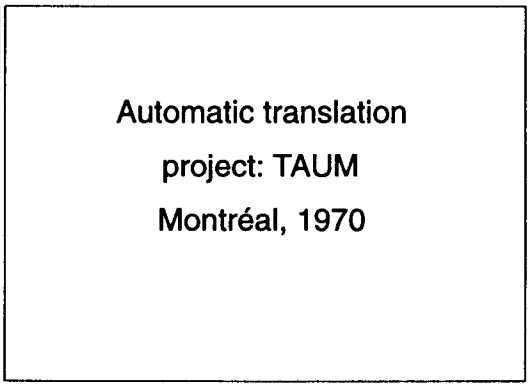
(SLIDE 9) What results is, in fact, a tree, in which you would have all the information contained in the sentence. Then you would have another set of Q-system rules which would take the tree and cut it into little pieces of complex symbols. While cutting it into little pieces you would start to produce some French.

(SLIDE 10) Then you would parse this output, obtain a French deep structure . . .

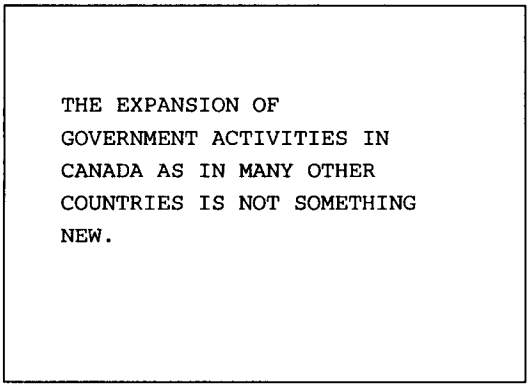
(SLIDE 11) . . . and then you would produce a French sentence.



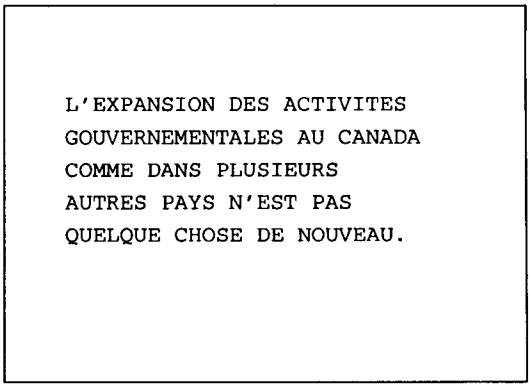
SLIDE 3



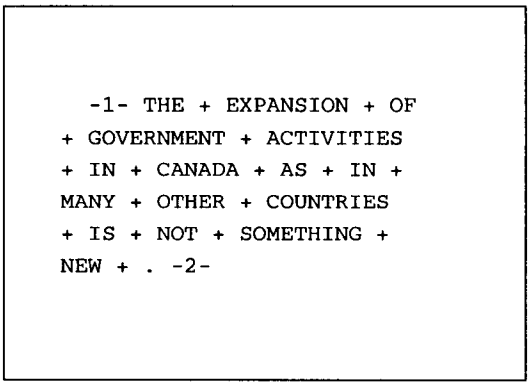
SLIDE 4



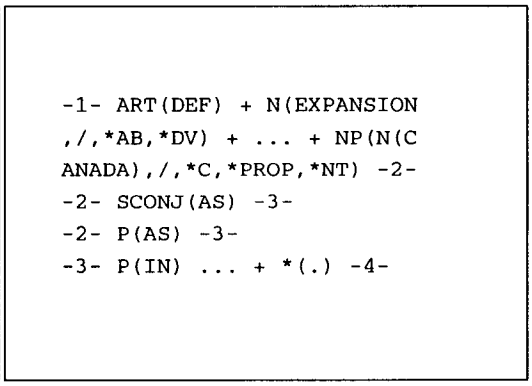
SLIDE 5



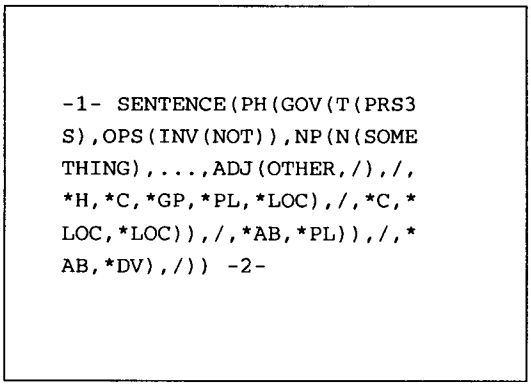
SLIDE 6



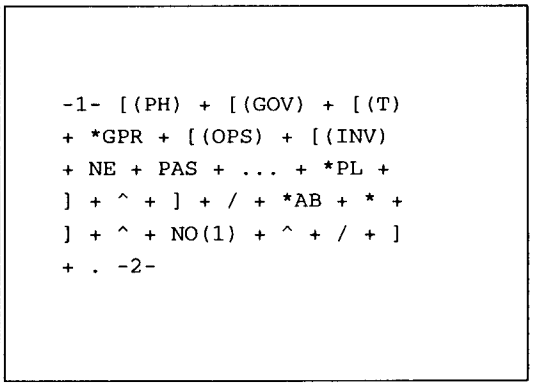
SLIDE 7



SLIDE 8



SLIDE 9



SLIDE 10

```
-1- PH(GOV(T(*IPR),OPS(INV
(NE,PAS)),SN(N(QUELQUE,CHO
SE),ADJ(GOUVERNEMENTAL,/).
GP(P(DANS),SN(CONJ(COMME),
SN(N(CANADA),...,)),/,/,*F
,*AB,*PL,3)),/,/*F,*AB,*S,3
,),/) + . -2-
```

SLIDE 11

```
-1- L + ' + EXPANSION + DE
S + ACTIVITES + GOUVERNEME
NTALES + AU + CANADA +COMM
E + DANS + PLUSIEURS + AUT
RES + PAYS + N + ' + EST +
 PAS + QUELQUE + CHOSE + D
E + NOUVEAU + . -2-
```

SLIDE 12

(SLIDE 12) This is just to give an idea of what we were doing.

(SLIDE 13) I would like to say a few words about the Q-systems. Essentially a Q-system is a set of rewriting rules.

(SLIDE 14) Here the third rule states that the subsequence **X + C** can be rewritten as the sequence **C + X**. When you apply the rewriting rules on an input sequence like **A + A + B + B + C + C**, you want to be very efficient and avoid a large search space. The way we did it, was to represent each symbol occurrence by an arrow labeled by that symbol and thus to represent the input sequence by a graph. Then the result of applying the different rewriting rules becomes the following graph.

(SLIDE 15) This graph representation leads to an efficient bottom-up parser which keeps most common parts together.

The important thing was that, within a rule, you were allowed to speak about complex symbols and complex symbols were just trees, like the ones we now have in Prolog. To denote complex symbols, that is, trees, we were using formulae, similar to Prolog terms, with the difference that there were three kinds of variables, each represented by a single letter followed by an asterisk. A variable with a letter from the beginning of the alphabet, like "A," was used to denote an unknown label of a node, a variable with a letter in the middle of the alphabet, like "I," was used to denote a whole unknown tree and a variable with a letter at the end of the alphabet, like "X" and "Y," was used to denote an unknown sequence of trees.

Q-Systems

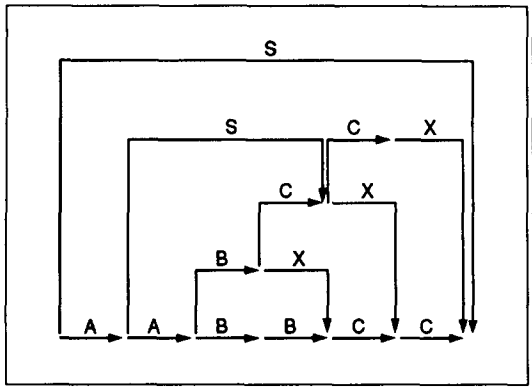
Montréal, 1970

SLIDE 13

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


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

SLIDE 14



SLIDE 15

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

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

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

SLIDE 16

(SLIDE 16) So, for example, if you apply the rule in the middle of this slide, to the sequence at the top of the slide you obtain the three different sequences at the bottom of the slide, according to the fact that the variable "I*" matches "R," "S," or "T." Here we have a nondeterministic unification, but it is not a true unification, it is more like a pattern matching because it just works in one direction. But, the status of the variables here was already the same as in Prolog. They were true unknowns.

(SLIDE 17) The Q-systems were very efficient and they are still used in a program which every day translates Canadian weather reports from English to French. This slide shows an example of such a translation made in 1978.

(SLIDE 18) During my stay at the University of Montréal, I was surrounded by a large team of linguists and I had to spend a lot of energy to make the computer scientists work together with them. Also, I was working more on syntactical aspects of natural languages than on semantical aspects, and my knowledge of English was, and is still, too poor to work on English syntax. So I wanted to stop working on automatic translation and do research in the processing of information contained in French sentences.

Just before I went to Marseilles I was introduced to the "Resolution Principle" paper by Alan Robinson about automatic theorem proving. I was very impressed and then saw the possibility of designing a system able to "understand" statements and questions written in French and able to do the right logical inferences for answering the questions. So, I arrived in Marseilles together with Jean Trudel, a Canadian doctoral student, in the winter of 1971. Both Jean and Philippe Roussel started to write different theorem provers à la Robinson. But, instead of proving mathematical theorems, Jean always tried to do inferences on little stories.

(SLIDES 19 and 20) This is the kind of logical axiom he was writing for this purpose: reflexivity (axiom 1) and transitivity (axiom 2) of the subset relation as shown: if the sets "x" and "y" are in the relation "r", and if "x" prime is a subset of "x" and "y" prime is the subset of y then "x" prime and "y" prime are also in the relation "r" (axiom 4). For example, if the boys love the girls, then if you take a boy and a girl then that boy will love that girl.

It's very naive but if you work on natural languages, you always have the problem of plural noun phrases which denote sets. Thus all your relations are on sets and you have to make a connection between sets, subsets, and your relations. We also had a lot of trouble with the subset relation because the addition of a communitivity axiom was producing a lot of unnecessary inferences. So we only allowed communitivity in the case where we were sure that every subset was a singleton (axiom 3).

TAUM METEO 1978

CLOUDY WITH A CHANCE OF
SHOWERS TODAY AND THURSDAY

NUAGEUX AVEC POSSIBILITE
D AVERSES AUJOURD HUI ET
JEUDI.

SLIDE 17

Primitive natural language
communication system

Marseille, 1971

SLIDE 18

1
$(\forall x)$[Subset(x,x)],

2
$(\forall x)(\forall y)(\forall z)$[Subset($x$,$y$)^
Subset(y,z) $\Rightarrow$ Subset(x,z)],

SLIDE 19

3
$(\forall a)(\forall b)$[Subset(The (a)], The (b))
$\Rightarrow$ Subset(The(b), The(a))],

4
$(\forall x)(\forall y)(\forall r)$ $(\forall x')$ $(\forall y')$
[True(r,x,y) ^ Subset(x',x) ^
Subset(y',y) $\Rightarrow$ True(r,x',y')].

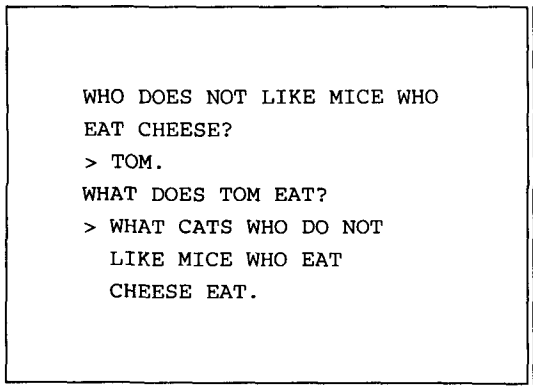
SLIDE 20

CATS KILL MICE.
TOM IS A CAT WHO DOES NOT
LIKE MICE WHO EAT CHEESE.
JERRY IS A MOUSE WHO EATS
CHEESE.
MAX IS NOT A MOUSE.
WHAT DOES TOM DO?

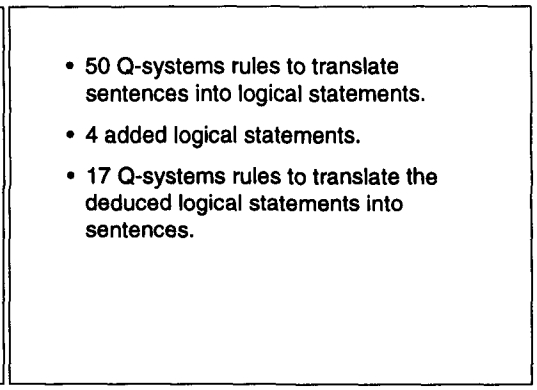
SLIDE 21

> TOM DOES NOT LIKE MICE
  WHO EAT CHEESE.
> TOM KILLS MICE.
WHO IS A CAT?
> TOM.
WHAT DOES JERRY EAT?
> CHEESE.

SLIDE 22



SLIDE 23



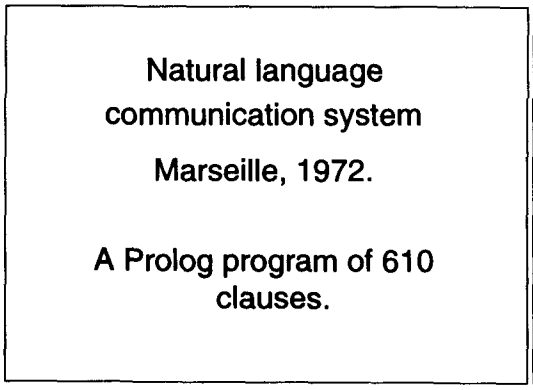
SLIDE 24

(SLIDES 21, 22, 23, and 24) By putting together the four axioms, a whole theorem prover, a little Q-system to parse sentences in French, another one to produce answers, we were able to have the conversation shown in the slides and in which sentences starting with a ">" prompt are answers generated by the computer. Our taste for embedded relative clauses must be noticed when, to the query "What does Tom eat?" the computer replies "What cats, who do not like mice who eat cheese, eat."

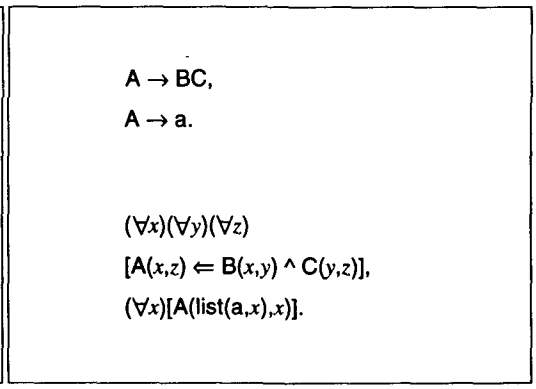
(SLIDE 25) We thought maybe we could use the theorem prover not only for making inferences, but also for the parsing and for the synthesis of the sentences. I spent a long time solving this problem.

(SLIDE 26) I think it was in the summer of 1971 that I found the trick for writing a context-free parser using logical statements. It is a trick that is now well known under the name "difference lists". If you consider the first context-free rule of slide 26, you would like to write roughly: if "x" is of type "B" and if "y" is of type "C", then "x" concatenated with "y" is of type A. You would then have to run a concatenation subprogram and the parser would be very inefficient. So you just write: if the list "x" minus "y" (that is, the list "x" from which you have suppressed the ending list "y") is of type B and if the list "y" minus "z" is of type "C" then the list "x" minus "z" is of type "A."

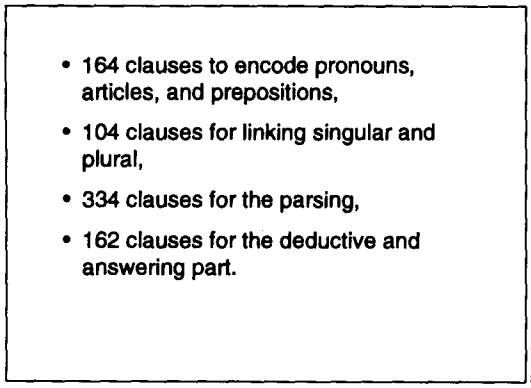
After this discovery, I decided that from now on we should program everything by just writing logical statements and use a kind of theorem prover to execute our statements. So we wrote a special theorem prover that I have called Prolog 0 and we wrote the first Prolog program which was already large.



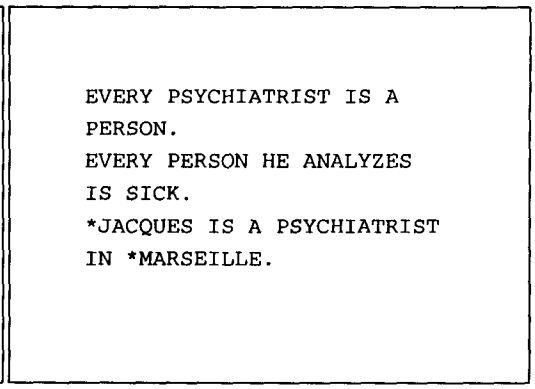
SLIDE 25



SLIDE 26



SLIDE 27



SLIDE 28

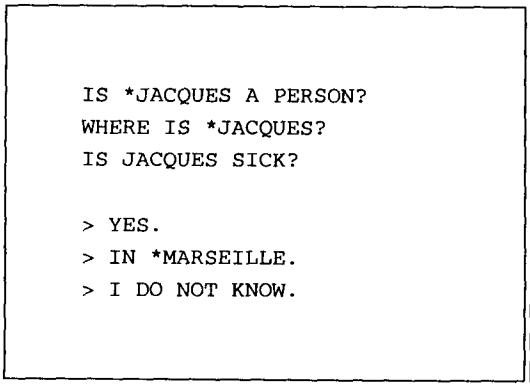
(SLIDE 27) This is the size of the different parts of the program: a dictionary of pronouns, articles and prepositions, a small morphology of French, a large parser of French sentences and finally a part for computing and generating answers.

(SLIDES 28 and 29) These are the kinds of sentences you were able to give as input to the computer. In these sentences the computer knows only the words "every," "is," "a," "he," "in" (that is, prepositions, articles and pronouns) and recognizes the proper nouns "Jacques" and "Marseille" because of the asterisk in front of them. With this input, you asked the questions : "Is Jack a person?", "Where is Jack?" "Is Jack sick?" and then you got the answers.

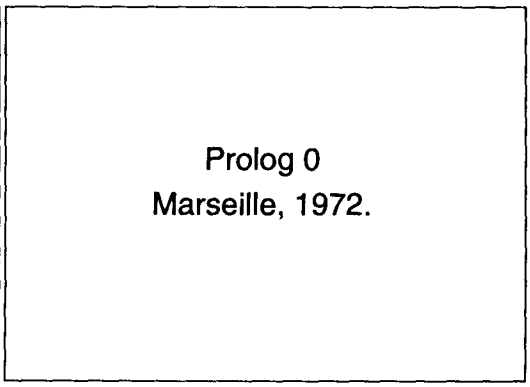
(SLIDE 30) This system was written using Prolog 0, the very first Prolog.

(SLIDE 31) For the syntax of Prolog 0 programs, this gives an idea. I was using an asterisk in front of the variables (in the Q-systems I used to put an asterisk after the variables). The minus sign denotes negation, the plus sign is used otherwise and all the atomic formula are implictly connected by "or" connectors. At the bottom of the slide you can see the way you would write the same program now if you use the normalized Prolog which I hope will come out soon.

There were some main differences between Prolog 0 and current versions of Prolog. There was not a simple cut operation but there were different double punctuation marks at the end of each clause to perform different types of search space reductions.



SLIDE 29



SLIDE 30

```
+APPEND(NIL,*Y,*Y).
+APPEND(*A.*X,*Y,*A.*Z)
        -APPEND(*X,*Y,*Z).

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

SLIDE 31

No cut operation but double punctuation marks at the end of the clause:

.. cuts after the clause head,
.; cuts after the whole clause,
;. gives only the first answer,
;; do not cut.

SLIDE 32

(SLIDE 32) For example you would put a double dot if you wanted to perform a cut after the head of the clause. There were very few predefined predicates.

(SLIDE 33) There was a "DIF" predicate to write nonequal constraints. You could write "X" is not equal to "Y" and it would delay the test until "X" and "Y" were enough known. So Prolog 0 already had features which we later put in Prolog II. There was a primitive called "BOUM" to split or create an identifier into, or from, a string of characters. And somebody wanted to have a strange thing which was to copy terms and rename the variables.

We didn't like it but we were forced to add the "COPY" predicate. There was no explict input and output instructions. For the ouput, what you were supposed to do was to put a slash in some clauses and then all the literals after the slash would be delayed and accumulated. At the end the Prolog 0 interpreter would print the sequence of all these literals. What you would input, would in fact be a set of clauses and the output would be another set of clauses which would be entailed by the first set of clauses. This was Prolog 0.

(SLIDE 34) Then we decided to make a full programming language which I have called Prolog 1. This language was very close to the actual Prolog programming language.

(SLIDE 35) With Prolog 1, there was the introduction of a unique cut operation and no more not-equal constraints. There were all the classical built-in predicates like input, output, assert, retract. The names of the last two ones were "AJOUT" and "SUPPRIMER". There was a metacall: you could

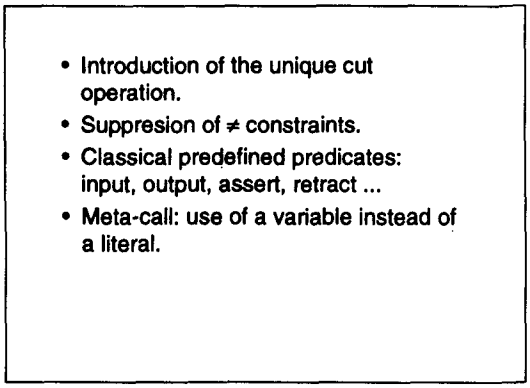
Predefined predicates:

- DIF for the ≠ constraint of Philippe's dissertation,
- BOUM to split or create an identifier,
- COPY to copy a term.

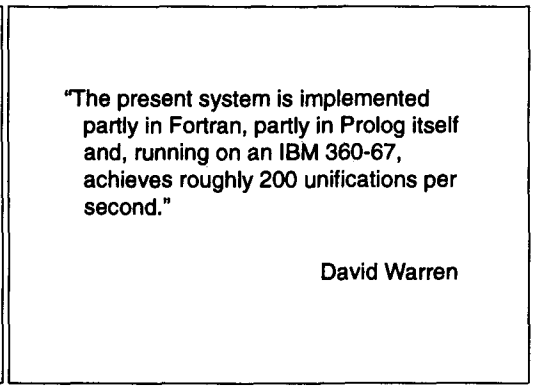
SLIDE 33

Prolog 1
Marseille, 1973

SLIDE 34



SLIDE 35



SLIDE 36

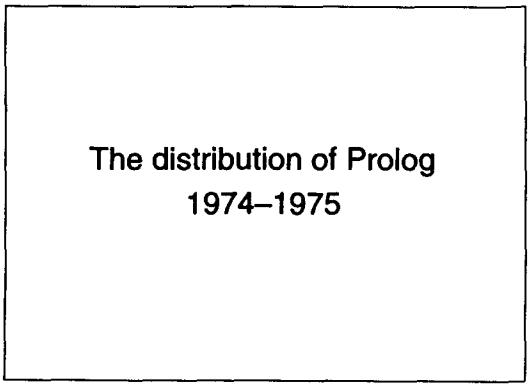
use a variable instead of a literal. This was because a large part of Prolog was written in Prolog itself and thus we needed some meta level.

(SLIDE 36) We had a visit from David Warren. This was in January 1973. After his visit, David wrote the sentence shown, which among others gives the speed of our implementation. In fact, an early version of Prolog 1 was running on a Sigma 7. This computer was sold in France by BULL, but under a different name.

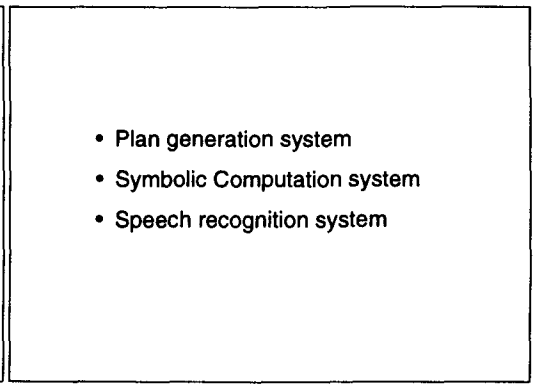
(SLIDE 37) The fact that the interpreter was written in FORTRAN was very helpful in the distribution of the language. We had a lot of visitors who came and took a copy away with them. First what happened was that different people started using Prolog to do something other than natural language processing.

(SLIDE 38) The first other application was a plan generation system made by David Warren who was visiting us. Then there were people like Marc Bergman and Henry Kanoui who started to write a symbolic computation system. Then Henri Meloni started to make a speech recognition system using Prolog.

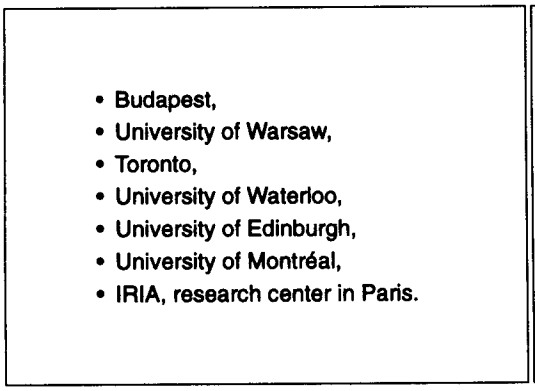
(SLIDE 39) At the same time people came and took Prolog away to Budapest, University of Warsaw, Toronto, University of Waterloo, University of Edinburgh, University of Montreal, and INRIA in Paris. I was later told that David Warren took a copy from Marseilles and brought it to Stanford. From there Koichi Furakawa took a copy to Tokyo. This story was related in an issue of *The Washington Post*.



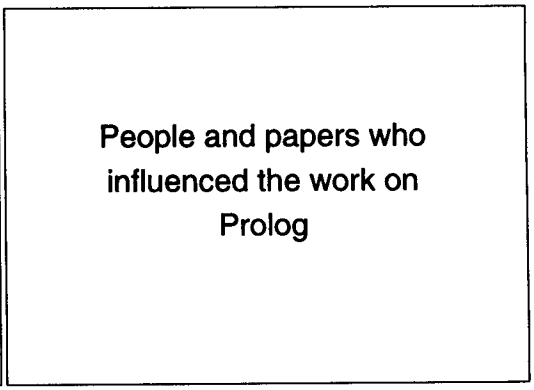
SLIDE 37



SLIDE 38



SLIDE 39



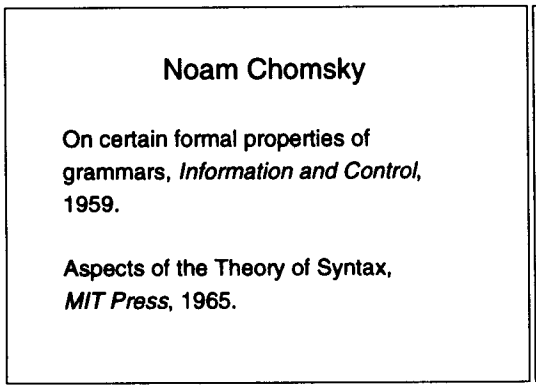
SLIDE 40

(SLIDE 40) I wanted to talk about the people and papers who influenced the work around Prolog. First, it sounds strange but Noam Chomsky had a strong influence on my work. When I did my thesis, I worked on context-free grammars and parsing and I spent a long time on this subject. At that time, Noam Chomsky was like a god for me because he had introduced me to the context-free grammars in his paper mentioned at the top of this next slide.

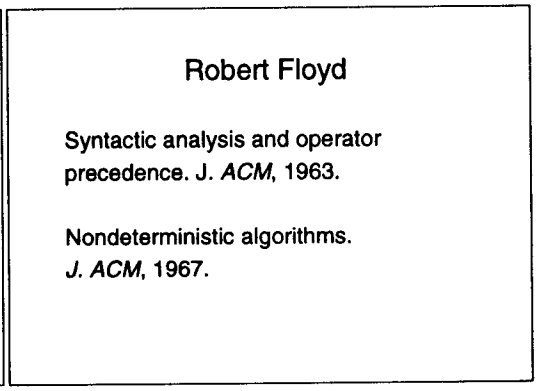
(SLIDE 41) Also, when I was in Montreal, I discussed with people who were working in linguistics and these people were very influenced by the second paper mentioned in the same slide. If you look at this second paper from the point of view of a computer scientist, you understand that you can do everything in a nice way if you just use trees and transform them. I think this is what we are all doing. We are doing this in Lisp; we are doing this in Prolog. Trees are really good data structures.

(SLIDE 42) Another person who is not a linguist but a computer scientist and who had a strong influence was Robert Floyd. He visited me when I was still a student in Grenoble and I was very impressed by him. I did my thesis starting from the first paper mentioned in the slide. I knew about the work described in the second paper before it was published. So I understood the way to implement a nondeterministic language, how to do the back-tracking, how to save information in a stack, and so on, and how to do it in an efficient way.

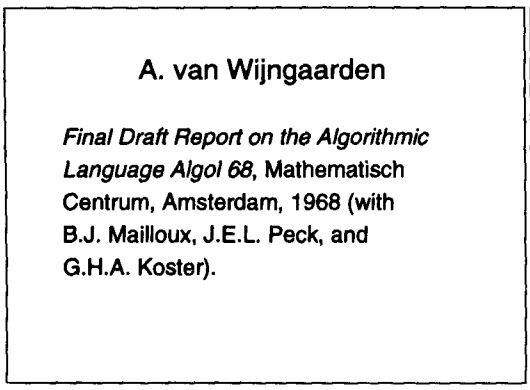
(SLIDE 43) Another person who influenced me was A. van Wijngaarden. I was influenced in two ways. First, as a student I participated in different meetings to design a new ALGOL 60. I was very



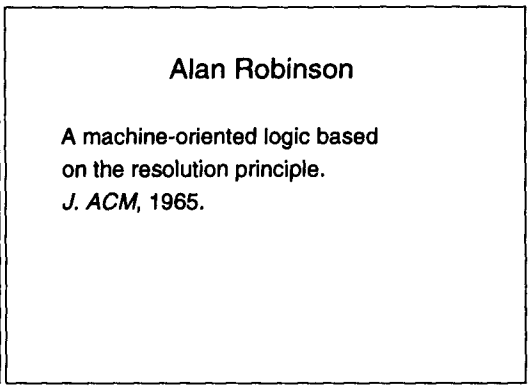
SLIDE 41



SLIDE 42



SLIDE 43

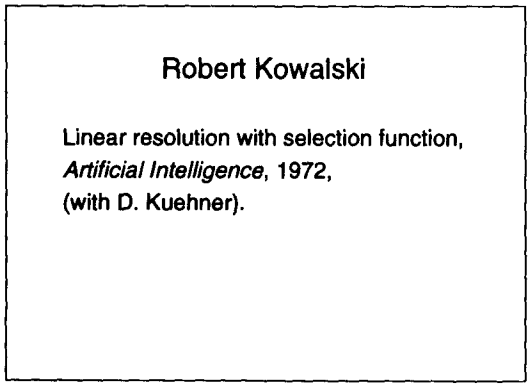


SLIDE 44

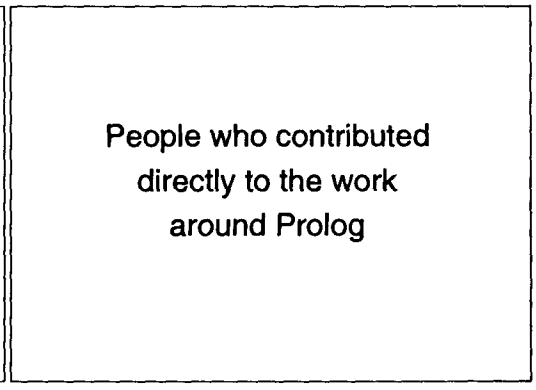
impressed to see that people, starting from nothing, were creating a whole language. It's hard to believe that you can just take a sheet of paper and design a new language which will be used by many people. Secondly, I was very strongly influenced by the W-grammars, which is my name for the formalism introduced by A. van Wijngaarden to describe ALGOL 68. And in fact, I wrote a complete parser for languages that are generated by W-grammars. I also wrote a complete generator and I used both for performing automatic translation before using the Q-systems.

(SLIDE 44) Of course, I think that Prolog would not have been born without Alan Robinson. I did not get the paper mentioned in the slide directly. I was still at the University of Montreal, and a student came, Jean Trudel, and showed me the paper. I looked at it and I didn't understand anything. I was not trained in logic at all. Jean Trudel spent a lot of time working on the paper and explaining it to us. This was a very important paper.

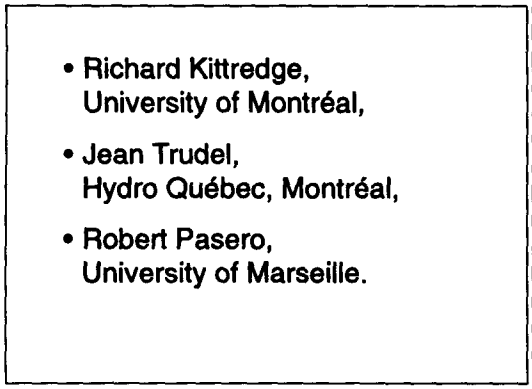
(SLIDE 45) There was another influence: Robert Kowalski. When we arrived in Marseilles, we were not trained in logic and we were looking at all the papers written on automatic theorem proving. But we didn't exactly have a feeling for why things were made in such a way. So we invited people who knew a lot about automatic theorem proving. Robert Kowalski came to visit us and of course something clicked. He talked about logic, we wanted to do inferences and he did not know a lot about computers. So we began a very close collaboration. He had a very nice prover which was essentially



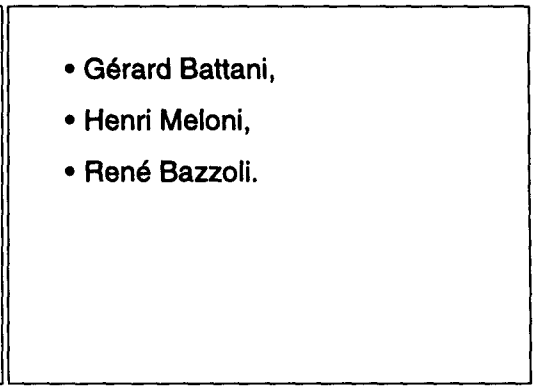
SLIDE 45



SLIDE 46



SLIDE 47



SLIDE 48

a resolution prover and the first Prolog was somehow an imitation of his complete SL-resolution method mentioned in the slide.

(SLIDE 46) These are people who were more directly involved with the implementation or discussion which were next door.

(SLIDE 47) First, I learned a lot from Richard Kittredge at the University of Montréal. He was a linguist who had received his Ph.D. from Zelleg Harris. I wanted to thank Jean Pierre Trudel because he was really the person who filled the gap between Robinson's paper and our own work. Robert Pasero is the person who was always involved on the natural language processing side of Prolog.

(SLIDE 48) And there are three other people who really were the implementors of Prolog 1 which was written in FORTRAN: they are Gerard Battani, Henri Meloni, and René Bazzoli. The first Prolog interpreter was their first FORTRAN program!

That is all I have to tell.

TRANSCRIPT OF DISCUSSANT'S REMARKS

SESSION CHAIR RANDY HUDSON: Jacques Cohen is a Professor in the Michtom School of Computer Science at Brandeis University. He has two Doctorates, one from the University of Illinois, the other from the University of Grenoble, France. He has held visiting positions at MIT, Brown, and at the University of Marseilles where the Prolog language originated. From 1963 to 1967 he was a colleague of Alain Colmerauer as a Doctoral student at the University of Grenoble. At that time, they had similar interests in syntax-directed compilation and nondeterminism. Jacques interacted only informally with Alain in the 1970s. In 1980, he was invited by Alain, now the head of the Artificial Intelligence group at the University of Marseilles, to teach a yearly course on compilers. Since the early 1980s, Jacques has been an aficionado of Prolog and has written several papers on logic programming including one on its early history. Jacques' current research interests are in the areas of compiler development using logic programming, parallelism, and constraint languages. He is currently Editor in Chief of Communications of the ACM. Jacques is a member of the Program Committee of this conference.

JACQUES COHEN: In the first HOPL Conference the discussant for the Lisp language was Paul Abrams who rightly pointed out that LISP was unusual in the sense that it had not been designed by a committee, but it was an invention or, if you want, a discovery of a single individual, John McCarthy. Abrams also mentioned that he knew only one other language with a similar origin. That was APL,

the brain child of Ken Iverson. One can state that Prolog also belongs to that category of languages. Prolog had a principal inventor, Alain Colmerauer, who contributed to most of the ideas underlying that language.

Robert Kowalski and Philippe Roussel also played key roles. The first one, as an expert of automatic theorem proving, the second one as a doctoral student of Alain's who implemented the first version of Prolog. Alain and Philippe admit in their paper that had Prolog not been discovered by them, it would have certainly appeared later under a form which would be recognizable as a variant of Prolog as it is known nowadays. I wish to emphasize that this in no way diminishes the amazing feat of having been the first to discover this beautiful language. Again, a parallel with Lisp is in order. Prolog, like Lisp, has a very simple syntax and embodies an elegant set of primitives and rules. In my view, both languages appear to be so simple and effective for expressing symbolic computation, that they would have come into existence sooner or later.

In the 1960s I was a colleague of Alain as a doctoral student at the University of Grenoble. I remember that one of his first research projects was to write a syntax-directed error detector for COBOL programs. Alain avoided reading many technical papers because at the time he had some difficulties with English. I am sure that when he located a key paper he read it with great care. I believe that two papers from Robert Floyd had a considerable impact on Prolog. The first was on parsing and the second was on backtracking.

Prolog rules can be viewed as grammar rules and parsing corresponds to the equivalent Prolog program. Furthermore, the subject of parsing is closely related to Alain's interest in natural language processing. It is relevant to point out that Alain became aware of Alan Robinson's seminal paper of resolution and unification five years after that paper had been published in the *Journal of the Association for Computing Machinery*. As happened in the case of the Floyd papers, Alain realized the importance of Robinson's work and that prompted him to contact experts in theorem proving like Robert Kowalski.

In the conclusion of their paper, Colmerauer and Roussel state that their first Prolog interpreter was essentially a theorem prover à la Robinson. The great merit of the inventors is to transform a theorem prover into an interpreter for a novel programming language and that in itself represents a formidable task. To illustrate that transformation, let us consider the following experiment: Submit a set of current Prolog programs to Robinson's theorem prover. (A parenthetic remark is in order. That set of Prolog examples actually took several years to develop. It would not be available without the experience gained in programming in the new language.) Robinson's prover would perhaps provide the needed answers to the current Prolog programs. But, it would be excruciatingly slow or it would run out of memory, even using present-day workstations.

The heart of Colmerauer's quest was how to make Robinson's prover work practically. Efficiency became of paramount importance and the Prolog designers were willing to use all sorts of compromises provided that they could solve the class of problems they had in mind. In Prolog's case the problem was natural language understanding using computers. The compromises that Alan Colmerauer was willing to make would stupefy even the most open-minded logician. First, they dropped completeness by performing only certain steps needed to prove a theorem. Therefore, they had no guarantee that if it was known that the theorem was provable, that the prover would inevitably find that proof. They also dropped an important test which is called "the occur-test," which insures that the unification works properly. In other words, they were willing to eliminate these costly tests even though certain programs would automatically result in an infinite loop. Finally, they introduced annotations (mainly to cut operations) that bypass certain computations in an effort to further increase the speed at the expense of missing certain solutions of a problem.

All this was done for the sake of increasing efficiency. But the gained efficiency was precisely what enabled Alain to experiment with the prover by concentrating on expressing problems in their embryonic language. That experience was crucial in making the transition from a theorem prover to an interpreter of a new programming language.

The role of Robert Kowalski in cleaning up the rudimentary language that preceded Prolog is considerable. Kowalski's main contribution was the discovery that most of the sample problems that he and Alain were submitting to the general theorem prover were of a particular kind. They simply did not need all the power of the general prover. I have had the opportunity to discuss that discovery with Robert Kowalski. According to Robert, it was a moment in which the main faults in Alain's implementation actually became a superior advantage. It avoided the combinatorial explosion of Robinson's approach by restricting the form of the theorem submitted to the prover to the so-called Horn clauses. In so doing, we would then regain the formal advantages of dealing with logic. One can also state that the limitation of the occur-test and the introduction of cut were premonitions of the designers of the new language and the fertile new areas of research extending Prolog.

In my view, the development of Prolog reflects Polya's advice for solving difficult problems. If you cannot solve a difficult problem, try to solve a particular version of it. In the case of Prolog, the particular version of the problem developed into a rich area of logic programming. Polya's approach seems to underline Prolog's historical development and it can be summarized by: first specialize then generalize.

A discussion about the history of Prolog would be incomplete without a few remarks about implementation. I believe that the lack of a fast mainframe in Colmerauer's actually slowed down the development of Prolog. As I recall, even in 1980, Alain and his colleagues implemented one version of the first Prologs on an Apple II. It is known that Prolog interpreters have a voracious appetite for memory. The designers had to implement a virtual memory on the Apple II, using a floppy disk as slow memory. According to Alain and his colleagues the hardest part of this implementation was temporarily stopping a program using Control C and making sure that the information in memory was not destroyed. Fortunately, the visit to Marseilles of David H. Warren from Edinburgh enabled him to implement the first fairly efficient compiler for the PDP 10. It is fair to state that it was that compiler that contributed to a wider acceptence of Prolog.

Finally, I would like to mention that in my view, the recent contributions that Alain has made in the area of constraint logic programming, equal or surpass his previous contributions as an inventor of Prolog. One can only hope that in HOPL III, he or one of his colleagues will present a paper on the evolution of Prolog into the new class of constraint logic programming languages.

TRANSCRIPT OF QUESTION AND ANSWER SESSION

GUY STEELE from Thinking Machines: Q-systems seem reminiscent of the COMIT language. Was there any influence of one language on the other?

COLMERAUER: I am not familiar with COMIT but I am familiar with SNOBOL. But in SNOBOL there is no parsing. Furthermore, in the Q-system the rules are not context free. They are general rules specifying the rewriting of trees. So, the answer is no.

BIOGRAPHY OF ALAIN COLMERAUER

Alain Colmerauer was born in 1941 in Carcassonne, France. After receiving a graduate degree in computer science from the Institut Polytechnique de Grenoble, he received a doctorat from the

University of Grenoble in 1967. The subject of his thesis was: precedences, syntactic analysis, and programming languages.

His interest then turned from computer languages to natural languages. As an assistant professor at the University of Montréal, he worked for three years on a project for automatic translation from English into French. Upon returning to France in 1970, he was appointed professor in computer science at the Faculty of Sciences of Luminy. There, together with Philippe Roussel, he designed the programming language Prolog which has become indispensable in the field of Artificial Intelligence.

His work centers on natural language processing, logic programming, and constraint programming. A good introduction to his recent work is the article: An Introduction to Prolog III, in *Communications of the ACM*, July 1990.

He has received these prizes and honors:

- Lauréat de la Pomme d'Or du Logiciel français 1982, an award from Apple France shared with Henry Kanoui and Michel Van Caneghem.
- Lauréat for 1984 of the Conseil Régional, Provence Alpes et Côte d'Azur.
- Prix Michel Monpetit 1985, awarded by the Académie des Sciences.
- Chevalier de la Légion d'Honneur in 1986.
- Fellow of the American Association for Artificial Intelligence in 1991.
- Correspondant de l'Académie des Sciences in mathematics.

BIOGRAPHY OF PHILIPPE ROUSSEL

Philippe Roussel is presently Associate Professor at the University of Nice, Sophia Antipolis, and member of the I3S-CNRS Laboratory of the same university.

He is the coauthor, with Alain Colmerauer, of the Prolog language. He implemented it for the first time in 1972, while he was a researcher at the University of Marseilles. He received his doctoral degree the same year, in the field of automatic theorem-proving. After spending six years in this university as an Assistant Professor in Computer Science, he joined the Simon Bolivar University in Caracas, Venezuela. He took charge of a cooperative program for developing a master's program in computer sciences, working on logic programming and deductive data bases. He returned to France in 1984, first as manager of the Artificial Intelligence Department of the BULL Company and later, as a scientific director of the Elsa Software Company. During this time, he conceived the LAP language, an object-oriented language based upon logic programming paradigms.

His major research interest is knowledge representation languages for information system modeling.