A VIEW OF THE ORIGINS AND DEVELOPMENT OF PROLOG

Dealing with failure is easy:
Work hard to improve.
Success is also easy to handle:
You've solved the wrong problem.
Work hard to improve.

(UNIX "fortune" message aptly describing Prolog's sequential search mechanism in finding all solutions to a query)

JACQUES COHEN

The birth of logic programming can be viewed as the confluence of two different research endeavors: one in artificial or natural language processing, and the other in automatic theorem proving. It is fair to say that both these endeavors contributed to the genesis of Prolog. Alain Colmerauer's contribution stemmed mainly from his interest in language processing, whereas Robert Kowalski's originated in his expertise in logic and theorem proving. (See [26] and the following article.)

This paper explores the origins of Prolog based on views rising mainly from the language processing perspective. With this intent we first describe the related research efforts and their significant computer literature in the mid-1960s. We then show that those existing circumstances would very naturally lead to the development of a language like Prolog.

In this paper I present a review of the origins and development of Prolog based on a long-term association, both academic and personal, with Colmerauer, the computer scientist who led the Marseilles team in helping to develop that language.

A description of the evolution of logic programming presented by Robinson covers over a century of events stemming from the work of Frege [37]. Loveland's review of the related area of automated theorem proving spans a quarter of a century of developments in that field [29].

Both Robinson's and Loveland's papers contain narratives of the significant work done in the described areas. This review contrasts and complements these two references by providing the background and motivation that led to the development of Prolog as a programming language.

The underlying thesis is that even seemingly abstract and original computer languages are *discovered* rather than *invented*. This by no means diminishes the formidable feat involved in discovering a language. It is tempting to paraphrase the path to discovery in terms of Prolog's own search mechanism: One has to combine the foresight needed to avoid blind alleys with the aesthetic sense required to achieve the simplest and most elegant solution to a given problem. In our view the various research topics studied by Colmerauer and his colleagues, including myself, (almost) deterministically led to the development of Prolog.

AN EARLY INVOLVEMENT WITH SYNTAX ANALYSIS

Both Colmerauer and I started doing research on compilers in the fall of 1963. Our research group at the Institute of Applied Mathematics, attached to the University of Grenoble, was led by Louis Bolliet. It was an exciting period for French informatics. First, there was a national pride involved in building up a computer industry. Although not entirely successful in its practical performance, the Bull Gamma 60 computer was recognized as having a novel architecture and showed the talent and promise of its hardware designers. Second, there was an effort to decentralize research from Paris by providing additional funds to the regional universities. The University of Grenoble was at the time one of

This work was partly supported by the National Science Foundation under Grant DCR 85-00881.

the most active in software development and in numerical analysis; considerable funds were made available to Bolliet's group.

One of the projects undertaken by the group was the development of an Algol 60 compiler for an IBM 7044, considered a fairly large mainframe at the time. J.C. Boussard was responsible for that project, and all members of the group collaborated in testing the compiler. We familiarized ourselves with the Algol 60 Report [32] and with the existing compiling techniques. Several of us marveled at the pioneering and highly intuitional approach of Rutishauser in the multiple pass compilation of arithmetic expressions (see [1]). The next major work read by most of us was that of Dijkstra [14], in which a stack and empirically derived weights were used to perform a single pass compilation of Algol 60 programs. Bauer, Samelson, and Paul had also done work along similar lines, this time trying to establish a relationship between a given BNF grammar and the weights used by Dijkstra in his empirical approach [33, 40]. However, in our view it was Floyd [15] who first succeeded in automatically determining that relationship for certain classes of grammars.

Before analyzing the effect of Floyd's work on the members of our group, I would like to digress briefly to describe a few related papers that raise the issue of *determinism* versus *nondeterminism*, which in turn is central to the theme of this article. Brooker and Morris in England and Irons in the United States had by this time suggested innovative approaches to compilation called syntax-directed translations. Brooker and Morris showed how recursive procedures obtained automatically from the grammar rules could parse a string in the language generated by the grammar [3, 4]. Actions were then triggered whenever certain syntactic constructs were encountered while parsing.

Iron's compiler also used a parser defined by recursive procedures that, in contrast to Brooker's, was guided by data representing the grammar rules [21]. A notable characteristic of the parsers proposed by these authors is that they could operate in a nondeterministic manner by trying to apply a given grammar rule and, in case of failure, backtrack to try another rule. The nondeterministic parsers were more general (i.e., they could process other classes of grammars), but less efficient than their deterministic counterparts.

It was an exciting period for French informatics. . . . there was a national pride involved in building up a computer industry.

With these previous works available, the Grenoble compiler's group began to have a clearer idea of the type of research that could yield significant results. Among the pertinent questions were,

- How could one reduce the degree of nondeterminism necessary for parsing?
- Could Dijkstra's empirically derived weights be determined formally for various classes of grammars?

Two remarkable papers addressed these questions and were avidly read by members of the group. The first, by Griffiths and Petrick, described an attempt to quantify backtracking in parsing [18]. Their approach was to select a two-stack nondeterministic Turing machine and use its instructions to write various parsers for different grammars. Nondeterminism could be controlled by using a selectivity matrix that allowed the parser to avoid certain blind alleys. By simulating the Turing machine on a computer, the authors were able to measure the relative efficiencies of the various parsers.

The second paper, by Floyd [15], described automatic means for generating a matrix from a given grammar. This matrix, called a precedence matrix, was then used by the parser to perform a (deterministic) syntactic analysis of input strings. Floyd also showed that information contained in the matrix could, in certain cases, automatically yield the weights that had been empirically determined by Dijkstra.

The precedence matrix could always be generated from grammars not containing two adjacent nonterminals in the right-hand side of a rule. It was also known that any BNF grammar could be "massaged" into another one whose rules had that special type of right-hand sides. Determinism could only be attained if the generated matrix contained single entries. One could imagine a Floyd-type nondeterministic parser that, when confronted with multiple entries in the matrix, would successively try each one of them and backtrack whenever a blind alley was reached.

Nevertheless, for compiler writing purposes, nondeterminism was to be avoided, and it became desirable to extend Floyd's work to other classes of grammars. Wirth and Weber's paper [45] eliminated the restrictions imposed by Floyd as to the type of right-hand sides of rules. Essentially, Floyd's precedence matrix was constructed only for the elements of the terminal vocabulary. Wirth and Weber extended the matrix construction and parsing to cover nonterminals as well as terminals. That, however, introduced an asymmetry that was not aesthetically pleasing since parsing had to proceed sequentially from left to right.

Colmerauer's project was to write an error detection and recovery program for Algol 60. After studying the Griffiths–Petrick and Floyd papers, he first thought of means to generalize Floyd's ideas in order to make them applicable to wider classes of grammars. In his 1967 dissertation on total precedence relations [7], Colmerauer restored the parsing symmetry by allowing nonterminals to appear as elements of a (pseudo) input string behaving as a stack. The similarity to Griffiths and Petrick's two-stack Turing machine then becomes obvious.

Finally, it should be mentioned that, although Knuth's work on *LR* (*k*) grammars was known to us at

the time, his mainly theoretical emphasis offered little hope of usage in actual compilers [22]. (Later developments, however, proved this was not the case for small k.)

NONDETERMINISTIC ALGORITHMS AND W-GRAMMARS

It is interesting that Floyd, who had made significant contributions in helping remove nondeterminism in parsing, was the first person to suggest the introduction of certain primitives allowing the full usage of nondeterminism in programming languages [16]. When his paper was made available to our group (prior to its formal publication), it generated great excitement. I recall that, having finished his dissertation, Colmerauer spent some time with other colleagues in the group incorporating Floyd's nondeterministic primitives into Algol 60. These primitives allowed a user to write very short programs to perform fairly complicated tasks. The examples considered by Floyd were the eight queens problem and determining minimum paths in graphs. The notion of a tree of choices was clearly described in that paper, and details for an implementation were provided. Three stacks called M, W, and R were used: The main stack, M, was needed to store the information that had to be restored while backtracking (e.g., values of variables prior to an assignment, labels from where `goto`s originated); the other stacks were used for input (R) and output (W). Floyd also detailed the implementation of nondeterministic (i.e., backtrackable) procedures. All these features would play a significant role in the actual implementation of Prolog.

The existence of parameters in Prolog rules reinforces the similarity to W-grammars since the parameters may specify a potentially infinite number of rules.

By 1966–1967, the Grenoble compiler group had become interested in Algol 68. Colmerauer then turned his interests to the two-level grammars that had been proposed by van Wijngaarden [42]. An interesting characteristic of these grammars is their ability to specify an infinite number of context-free rules that would be generated by using an auxiliary context-free grammar. Prolog rules bear a remarkable similarity to context-free grammar rules: They both have one element in the left-hand side and several or no elements (i.e., the empty string) in the right-hand side. The existence of parameters in Prolog rules reinforces the similarity to W-grammars since the parameters may specify a potentially infinite number of rules. Also striking is the similarity of Prolog rules to Knuth's attribute grammars [23]. These similarities actually confirm the existence of common themes in computer science.

Colmerauer's involvement with W-grammars represented another step (perhaps unconscious at the time) toward designing the future language. In fact, Colmerauer implemented an analyzer for strings generated by W-grammars, as well as a sentence generator operating on given grammar rules. This early effort bears a relationship to his later work on natural languages where parsing and generation are accomplished by the same Prolog program.

NATURAL LANGUAGE PROCESSING AND THEOREM PROVING

In fall 1967, both Colmerauer and I left Grenoble, he for the University of Montreal, where he stayed until 1970, and I for Boston, where I became associated first with MIT and later with Brandeis University. At this time Colmerauer became primarily interested in artificial intelligence and natural language processing. Although my own research activities were in the areas of compilers and artificial languages, I kept my interest in nondeterministic algorithms and acquainted myself with theorem-proving techniques utilized in program correctness. Despite the occasional visits to Montreal, and later on to Marseilles, it was not until 1981 that I had the opportunity to reestablish a close contact with Colmerauer and his colleagues of the GIA (Artificial Intelligence Group) at Marseilles.

In a paper that appeared in 1969, de Chastellier and Colmerauer showed how W-grammars can be used to specify syntax-directed translations [13]. One of the examples considered in that paper was the translation of arithmetic expressions into their postfix polish counterparts. Basically, postfix "patterns" are first generated by metalevel rules and then substituted into the main grammar rules as special nonterminals.[1] These rules are then used to generate (or analyze) infix arithmetic expressions. A more complex example considered in the de Chastellier–Colmerauer paper is the translation of (simple) sentences from French into English and vice versa. The main idea is to design two W-grammars, one describing the generation of English sentences and the other, the generation of the French counterparts. The metalevel rules for each of these grammars describe the so-called deep structure of a phrase. A source string in one of the languages is parsed to produce its deep structure, which appears as a (special) nonterminal in the metalevel grammar. This deep structure is then used to generate the corresponding target string in the other language.

Colmerauer's work on W-grammars showed that they could be used to carry out the type of syntax-directed translation used in natural language processing. Nevertheless, he started working on a relatively simple formalism for expressing transformations of directed graphs. This formalism incorporated the combined use of rewriting grammar rules and pattern matching, and

[1] Note that the substitution of patterns into the main grammar rules bears a striking similarity with the substitution of variables by their values into a Prolog rule: *Each* occurrence of a variable is replaced by its corresponding value.

could be efficiently interpreted. This became known as System Q, which he considers the ancestor of Prolog [8]. A remarkable (Prolog-like) characteristic of programs written using this formalism is their ability to work in both directions; that is, not only could input data produce output results, but also, given a possible output, the program could determine the corresponding input data. The pattern-matching algorithm used to implement System Q was nondeterministic, and the rewriting rules were reminiscent of Chomsky's type O grammars stating that a sequence of trees of a given shape are rewritten into a sequence of trees having another specified shape. The notions of terms (trees) and variables as they are known in present-day Prolog had their counterparts in System Q. Actually, the main basic differences between the two formalisms are (1) the absence in System Q of a bidirectional pattern-matching mechanism equivalent to Prolog's unification, and (2) the restriction to the use of context-free Prolog-like rules instead of the more general rules used in System Q. Nevertheless. considerable experience was gained by implementing and experimenting with a System Q processor. The system is still used to translate Canadian weather reports from English into French.

It was just before his return to France in 1970 that Colmerauer became acquainted with Robinson's key paper on resolution and unification [36]. At that time he had accepted a professorship at the University of Aix-Marseilles. With Jean Trudel, a Canadian student who was already well versed in logic, Philipe Roussel, and Robert Pasero, Colmerauer became interested in text understanding using logic deduction. That interest led the Marseilles group to establish contact with Kowalski, then at the University of Edinburgh, whose work with Donald Kuehner on SL-resolution [27] served as an initial theoretical model for the then embryonic language. The main concern of the Marseilles group in 1972 still remained the development of a man–machine interactive system capable of making deductions from a set of natural language sentences. An

Considerable experience was gained by implementing and experimenting with a System Q processor. The system is still used to translate Canadian weather reports from English into French.

often mentioned reference describing the research of the group at the time is [12]. This work, implemented by Roussel in 1972, embodies a few of the features that are still current in most Prolog interpreters. Among them were (1) a method for redefining priorities and associativities of operators, and (2) the use of annotations (actually the precursors of the cut) to let the user reduce the search space during execution. A valuable contribution of Colmerauer in this work was the use of two lists represented by the variables U and V to indicate parts of the input string that parse to a given nonterminal N. Basically, list U contains list V as a tail, and N generates the elements between the beginning of U and the beginning of V. This representation proved to be the most efficient for parsing using Prolog and later became known as difference lists. A most remarkable feature of the 1972–1973 paper is the presentation of a quite complex program, comprising hundreds of (fairly general) clauses to implement a man–machine interactive system using natural language. The authors end the report by stating that the system was admittedly slow, but they were confident that a 25-fold speedup could be attained by using an improved version that was under development.

Roussel's first Prolog interpreter was written in Algol-W and employed what is now known as the clause-copying technique. During a visit to Edinburgh later that year, Roussel learned of the structure-sharing approach of Boyer and Moore [2]. Upon his return, two of his students, H. Meloni and G. Battani, implemented a Fortran version of the prototype using structure sharing (see [39]).

The use of `cuts` (similar in spirit to `gotos` in Algol) became indispensable in helping to reduce the size of the search space constructed by programs. Although Colmerauer today insists that programmers should minimize the use of `cuts`, he assumes full responsibility for introducing it in the language at that early stage. Negation as failure was also utilized informally by the Marseilles group at that time; however, a definition of what it accomplishes only appeared in 1978 [5].

It was the availability of the Fortran interpreter of Prolog that helped disseminate the language. In particular, D. H. D. Warren from the University of Edinburgh spent some time in Marseilles to acquaint himself with the interpreter and language. It is fair to say that the subsequent interpreters and compilers developed by Warren played a major role in the acceptance of Prolog. The reason is simple: The previous interpreters were slow and ran on relatively small computers. The availability of the compiler developed by Warren on a PDP-10 showed the potential of Prolog programs being executed with a speed comparable to those of Lisp programs [43].

An important (although basically ancillary) contribution was also made by Colmerauer in the mid-1970s. This is his concept of metamorphosis grammars [9]. In that work he directly mapped the rules of Chomsky's grammar to Prolog programs capable of recognizing strings generated by the grammar. By adding parameters to the grammar rules, one could easily perform syntax-directed translations. Colmerauer demonstrated the usefulness of metamorphosis grammars by consid-

ering two applications. He first showed how a compiler for a minilanguage could be succinctly described and rapidly implemented. The second application, one that admits his predilection, considers the use of metamorphosis grammars to develop an interactive system capable of carrying out a dialogue in natural language. The work is based on a detailed study of the meaning of articles in French. The system is capable of reasoning about the contents of input sentences by providing logically derived answers to questions, or by finding inconsistencies within the sentences. In both applications, the underlying grammars and semantic actions are automatically translated into Prolog programs.

Unfortunately, the work on metamorphosis grammars remained little known until 1980, when Warren pointed out its importance as a powerful tool for writing compilers [44]. The usage of metamorphosis grammars in natural language processing has also been demonstrated by Pereira and Warren [34].

In a recent conversation with Kowalski, I was provided with further information about the origins of the language. He introduced me to his unpublished manuscript on this subject, which had been circulated informally among members of the Prolog community [26]. In the early 1970s, Kowalski's research effort was spent on theorem proving. In their collaboration, Kowalski and Colmerauer became interested in problem solving and automated reasoning using resolution theorem proving. Green's work in this area was known to both researchers [17]. Kowalski, then at Edinburgh, concentrated his research on attempting to reduce the search space in resolution-based theorem proving. With this purpose, he developed with Kuehner a variant of the linear resolution algorithm called SL resolution (for linear resolution with selection function), based on Loveland's model elimination [27, 28]. Kowalski's view is that, from the automatic theorem-proving perspective, this work paved the way for the development of Prolog. Having this more efficient (but still general) predicate calculus theorem prover available to them, the Marseilles and Edinburgh groups started using it to experiment with problem-solving tasks. Several formulations for solving a given problem were attempted. Almost invariably, the formulations that happened to be written in Horn clause form turned out to be much more natural than those that used non-Horn clauses. According to Kowalski a typical example was that of addition that can be represented either by using Horn clauses, analogous to both a recursive function definition and a Peano axiomatization, or by non-Horn clauses whose meaning is not as easily explained. (See [25, p. 162] and examples in Figures 1 and 2). Another case in which the Horn clause formulation was particularly elegant occurred in parsing strings defined by given grammar rules.

Kowalski, interested in logic, was amazed at the capabilities of the Horn clause formulation when he found that recursive programs (such as factorial) could easily be expressed in that formalism. In contrast, Colmerauer, interested in language processing, saw the great potential of the then embryonic Prolog when he could express the list processing procedure `append` using the same formalism.

There were two important theoretical developments to Prolog made in the mid-1970s. The first was the Horn clause basis for logic programming presented by Kowalski [24] and the proof of completeness of the theorem-proving method in which Prolog was based [19]. The second was the establishment of a formal semantics for the language: van Emden and Kowalski defined a fixed-point semantics for Horn clause programs and showed that it was equivalent both to the minimal model and to the operational semantics [41].

Almost invariably, the formulations that happened to be written in Horn clause form turned out to be much more natural than those that used non-Horn clauses.

DESCRIBING THE TWO PERSPECTIVES USING METALEVEL INTERPRETERS

Using Prolog itself we will now describe how the components (1) parsing and natural language processing, and (2) theorem proving actually resulted in the same inference mechanism currently utilized in the interpretation of Prolog programs.

In the "Nondeterministic Algorithms and W-Grammars" section, I referred to the striking similarity between nondeterministic parsing and the Prolog inference mechanism. This similarity is easily made apparent by expressing a predictive parser in Prolog and comparing it with the language's metalevel interpreter.

A predictive (top-down) parser successively replaces a nonterminal *N* on the top of the stack by the right-hand side (*RHS*) of a grammar rule defining *N*. The Prolog counterpart of grammar rules are clauses, and a nonterminal corresponds to a Prolog procedure. Unit clauses represent nonterminals that rewrite into the empty symbol ϵ (indicated below by the empty list `nil`). Therefore, when the nonterminal *N* on the top of the stack rewrites into ϵ, *N* is simply popped. Grammar rules are stored using the unit clauses `rule(`*N*, *RHS*`)`, where variables appear in italic. The parser then becomes

```
parse(nil)
parse(N.Rest) ←
  rule(N, RHS),
  parse(RHS),
  parse(Rest).
```

This program is identical to the classic metalevel Prolog interpreter in which the predicate `clause(`*Goal*, *Tail*`)` is the counterpart of `rule(`*N*, *RHS*`)` and `true` replaces `nil`. (See, e.g., [6]). A query represents a given sequence of nonterminals; success is achieved if that sequence can be parsed into the empty string.

```
This is an example of the non-Horn clause addition using SL resolution
X.Y denotes cons(X, Y)
input_clause(+p(X, Y, Z).+a(0, Y).nil)
input_clause(+p(X, Y, Z).-a(X, Z).nil)
input_clause(+a(s(X), s(Y)).-a(X, Y).nil)
input_clause(-p(X, Y, s(s(s(0)))).nil)

example ←
  prove(-p(X, Y, s(s(s(0)))).nil, nil)

An empty clause is a contradiction
prove(nil, Ancestors)

Else resolve against an ancestor or an input clause
prove(Literal.Clause, Ancestors) ←
  get_resolvent(Literal, Ancestors, Resolvent),
  prove(Resolvent, Literal.Ancestors),
  prove(Clause, Ancestors)

Obtain a resolvent from the ancestor list
get_resolvent(Literal, Ancestors, nil) ←
  complement(Literal, Literal'),
  remove(Literal', Ancestors, Ancestors')

Or obtain it from an input clause
get_resolvent(Literal, Ancestors, Resolvent) ←
  input_clause(Clause),
  complement(Literal, Literal'),
  remove(Literal', Clause, Resolvent)

complement(-Literal, +Literal)
complement(+Literal, -Literal)

remove(Element, Element.List, List)
remove(Element, Element'.List, Element'.List') ←
  remove(Element, List, List')
```

FIGURE 1. A Simplified SL Prover

```
Horn clause addition using SLD resolution
input_clause(+p(0, X, X).nil)
input_clause(+p(s(X), Y, s(Z))
  .-p(X, Y, Z).nil)

An empty clause is a contradiction
prove(nil)

Else resolve against an input clause
prove(Literal.Clause) ←
  get_resolvent(Literal, Resolvent),
  prove(Resolvent)
  prove(Clause)
```

FIGURE 2. An SLD Theorem Prover

What is implicit in this program as well as in the metalevel interpreter is the crucial role played by unification when the predicate `clause(Goal, Tail)` is invoked. The pioneering work of Robinson using unification in theorem proving is, therefore, of capital significance in the development of Prolog.

As mentioned earlier, the original quest of the Grenoble compiler group was to reduce the amount of nondeterminism involved in parsing. This same problem, when transposed to the case of Prolog execution, still remains of paramount importance. Indeed, several current research papers in the Prolog literature are dedicated to the study of means to avoid run-time blind alleys by careful program examination at compile time. As in the parsing of context-free grammars, it is also important to detect programs that are strictly deterministic. Since a large number of practical Prolog programs satisfy this requirement, it has become important, in this context, to detect and optimize Prolog programs that do not require backtracking. There are many other similarities between grammar properties and properties of Prolog programs. Some examples are ambiguous grammars and Prolog programs exhibiting multiple solutions, and grammar transformations corresponding to program transformations. However, the prevalent use of unification in Prolog programs renders the study of program properties a considerably more difficult task than that of grammar properties.

A relevant parallel between parsing and theorem proving is the correspondence relating bottom-up parsers to forward-chaining theorem provers, and top-down parsers to backward-chaining theorem provers. Loveland, through his pioneering work in model elimination [28], advocated backward chaining, which is by far the preferred approach presently used in logic programming. Also note that a combined forward- and backward-chaining scheme based on Earley's parsing algorithm has been proposed as a model for Horn clause deduction [35].

Although it may seem anachronistic, it is enlightening to describe the evolution of Prolog from the theorem-proving point of view by presenting the successive methods that led to the presently used inference mechanism: Selective Linear Definite or SLD clause resolution. The word *definite* refers to Horn clauses with exactly one positive literal, whereas general Horn clauses may contain entirely negative clauses.

Let I be a set of input clauses. The set of clauses N representing the negation of the theorem to be proved is initially placed in the list I. Robinson's original resolution algorithm can be described as a nondeterministic algorithm. The initialization stage consists of (nondeterministically) selecting one of the clauses in N and placing it in a list R. Then, pairs of resolvable clauses from $I \cup R$ are (nondeterministically) selected, and their resolvent is placed in R. This process is repeated until the empty resolvent is generated. This algorithm can be implemented by any number of search techniques including depth-first and breadth-first search. Moreover, SL and SLD resolution can be viewed as special cases in which restrictions are placed on the clauses that can be selected for resolution. To simplify the presentation of the SL and SLD algorithms, we will henceforth assume the negation N of the theorem consists of a single clause, thereby eliminating the first choice in the algorithm. In linear resolution, the next resolvent is generated by considering the clause most recently placed in R and any other clause taken from either I or R.

SL resolution places further restrictions on the choice of the two clauses that can be used to generate the next resolvent. As in linear resolution, the first clause must be the one most recently added to *R*. Unlike linear resolution, restrictions are also placed on the second clause, and on the complementary literals selected in the two clauses [27]. Most of the subtleties of this choice are captured in the simplified SL prover presented in Figure 1. The main purpose for presenting this program is to show that, when its input is restricted to Horn clauses, the resulting SLD prover becomes the Prolog metalevel interpreter. Therefore, only the principal features of SL resolution are considered in the program.

In Figure 1, input clauses and the negation of the theorem to be proved are asserted in the database using the unit clause `input_clause` having as a parameter the list whose elements represent the clause. The clauses in Figure 1 are non-Horn clauses specifying the addition of natural numbers. Essentially, the literal `p` states that Z represents $X + Y$, and the literal `a(X, Y)` is true if the difference $X - Y$ remains constant.

Initially, the first parameter of `prove` contains, in clause form, the negation of the theorem to be proved, and the second parameter, the ancestor list, is the empty list. The ancestor list will contain the literals that have been previously processed. The procedure will succeed if the resolvent, the literals in the ancestor list, and the input clauses are not simultaneously satisfiable.

The resolvent is unsatisfiable if and only if *each* of its literals are unsatisfiable. To prove that a literal *L*, a set of input clauses *I*, and the conjunction of ancestor literals *A* is unsatisfiable, one can

(1) obtain an ancestor that unifies with the complement of the literal;[2] or, if this is not possible,
(2) find an input clause that resolves against the literal to form a new resolvent *N*, and then show that *N*, $I, L \wedge A$ are unsatisfiable.

The procedure `prove` has two levels of recursion, specified by the calls

`prove(`*Resolvent, Literal.Ancestors*`)`,
`prove(`*Clause, Ancestors*`)`.

The second call is used to prove that each of the literals in a clause is unsatisfiable, while the first is used to prove that the current literal is unsatisfiable (assuming the satisfiability of *I* and *A*).

The call to `remove` in the first clause of `get_resolvent` simply attempts to unify the complement of the literal with a member of the ancestor list, whereas the call to `remove` in the second clause actually produces a resolvent. The resolvent produced by the first clause of `get_resolvent` is always empty. It is assumed the Prolog processor interpreting the program in Figure 1 incorporates the *occur* check so that logical soundness is preserved.

[2] We have omitted a membership test that prevents the selection of a literal already present in the ancestor list; this test can be used to avoid (certain instances of) infinite loops [27].

An alternate informal presentation of the SL resolution algorithm can be done by elaborating further the correspondences between parsing and theorem proving. (We will initially restrict our attention to the propositional case.) Table I summarizes the relevant correspondences.

A clause containing *n* literals, $A_1, A_2, \ldots, A_n$, will represent the *n* context-free rewriting rules

$$\begin{aligned} A_1' &\rightarrow A_2 A_3 \cdots A_n \\ A_2' &\rightarrow A_1 A_3 \cdots A_n \\ &\vdots \\ A_n' &\rightarrow A_1 A_2 \cdots A_{n-1}, \end{aligned} \tag{1}$$

where A_i' is the complement of A_i. The rewriting rule $A \rightarrow BC$ should be interpreted logically as "if *A* then $(B \vee C)$."

The parsing counterpart of proving the validity of a query is showing that the sequence of nonterminals representing the query can be rewritten into the empty string ϵ. It follows that the *order* in which one shows that each nonterminal in the sequence rewrites into ϵ is irrelevant.

The counterpart of resolution in theorem proving is the rewriting of a nonterminal in a sentential form using a grammar rule to produce a new sentential form [20]. More specifically, if $N\alpha$ is a sequence of nonterminals whose first element is *N* and $N \rightarrow \beta$ is a rule, then $\beta\alpha$ is the resulting new sequence; that is,

$$N\alpha \Rightarrow \beta\alpha.$$

It should be noted that, from an intuitional point of view, a sequence of nonterminals also corresponds to a clause that, in turn, can be viewed as a grammar rule. For example, the sequence $N\alpha$ corresponds to the grammar rule $N' \rightarrow \alpha$. Therefore, the derivation $N' \stackrel{*}{\Rightarrow} N\alpha$ can be replaced by the simpler derivation $N' \stackrel{*}{\Rightarrow} \alpha$. This operation is performed by the program in Figure 1 when it checks whether the complement of a literal is in the ancestor list (first clause of `get_resolvent`). Therefore, the parsing counterpart of `get_resolvent` has the role of checking if the given nonterminal is in the ancestors' list, in which case a success is assumed (the procedure `remove` is used to test for membership); otherwise, the nonterminal is (later) incorporated to the ancestors' list, and the procedure `remove` simply generates the proper right-hand side of the rule by removing the appropriate nonterminal to simulate the application of one of the rules (eq. (1)). The procedure `prove` then proceeds to determine if the sequence of nonterminals that constitute the right-hand side can be rewritten into ϵ.

It should be noted that, when generalizing the previous explanation to the predicate case, each clause containing variables corresponds to an infinite number of grammar rules. Therefore, unless the literals in the query are all ground, it becomes necessary to include the query as an additional input clause.

It is straightforward to transform the (simplified)

TABLE I. Correspondences between Parsing and Theorem Proving

Nonterminal	Literal
Grammar rule	Clause
ε-rule	Unit clause
Concatenation	Disjunction
Rewriting	Resolution
Parsing tree	Proof tree

SL prover in Figure 1 into an SLD[3] program capable of handling Horn clauses only. In this case it can be assumed that the clause to be proved is a list of negative literals. Furthermore, each input clause contains exactly one positive literal: the first. Thus, the ancestor list will consist only of negative literals, and hence no ancestor can resolve with the current literal since both are negative. Consequently, the second parameter in the procedures `prove` and `get_resolvent` of Figure 1 can be eliminated without sacrificing soundness. The pertinent part of the transformed program appears in Figure 2, which also shows a program to perform natural number additions using Horn clauses.

The procedure `get_resolvent` in Figure 2 will now only resolve against an input clause. A symbolic execution yields

```
get_resolvent(−Literal, Resolvent) ←
  input_clause(+Literal.Resolvent).
```

Finally, the replacement of the above `get_resolvent` in Figure 2 yields the classical metalevel interpreter:

```
prove(nil)
prove(−Literal.Clause) ←
  input_clause(+Literal.Resolvent),
  prove(Resolvent),
  prove(Clause).
```

A COMPARISON WITH THE DEVELOPMENT OF LISP

List processing in Prolog is done using terms that simulate the classical Lisp primitives, *cons*, *car*, and *cdr*, that allow for the construction of lists and the determination of their head and tail components. Recursion plays a primary role in both languages.

It is fair to say that most list processing in Prolog is done à la Lisp. Prolog, however, offers the additional features of nondeterminism and logical variables that allow the automatic determination of multiple solutions to a problem and, in certain cases, the performance of inverse computations (e.g., sorting-permutation, parsing-string generation, differentiation-integration).

We will now draw parallels between the development of the two languages and venture an explanation as to possible reasons for the longer time taken by Prolog to establish itself as a useful language.

[3] The word *selective* in SLD means that any negative literal in the current resolvent can be selected. *Selective* in SL means that any most recent literal can be selected. Therefore, SLD is not strictly a special case of SL.

First, it should be pointed out that the theoretical foundations of both languages resulted from the efforts of persons who had a training in mathematical logic. Coincidentally, both McCarthy and Robinson obtained their doctorates from Princeton University. Not surprisingly, McCarthy chose a logician's "language," Church's lambda calculus, as the theoretical foundation for Lisp. His presence at MIT in the early 1960s played a significant role in the success of the language [31]. He was surrounded by brilliant "hackers" and had ample access to up-to-date equipment.

The success of a language depends greatly on the development of a handful of interesting (and preferably short) programs that show its expressive power. McCarthy was certainly on the right track when he chose to present, among others, a differentiation program and *evalquote*, the universal interpreter [30]. Today these programs are assigned as homework in undergraduate computer science courses. Yet, one can imagine what a tour de force this would have been a quarter of a century ago, using much smaller and slower computers, disposing of relatively meager software.

The availability of Lisp in the early 1960s at MIT was a bonanza for its AI group and provided an unusual environment in which to test and perfect the language. Dozens of significant doctoral dissertations published at the time were made possible because of the existence of Lisp running on adequate equipment.

In contrast to Lisp, the development of Prolog proceeded at a relatively slower pace, especially considering that Robinson's ground-breaking paper [36] had been published in 1965. It remains for us to speculate as to the reason for this difference.

The first point has to do with the resemblance of Church's lambda calculus to an actual programming language, even a more primitive one. The notion of subroutines was well known at the time of Lisp's development, as was the concept of recursion. In my view, McCarthy's greatest contributions were (1) to have chosen a solid theoretical foundation for his language, (2) to show that it could be implemented with the existing available computers, and (3) to present a sample of useful examples demonstrating the language's capabilities.

Although Robinson's paper [36] provided the solid theoretical foundation for a language like Prolog, resolution was originally intended to be used to prove theorems in the predicate calculus. Theorem provers using the original resolution method had to contend with rampant nondeterministic situations, the redundancy in obtaining solutions, and the lack of goal-oriented searches. The knowledge on how to face these problems would require a considerably greater effort than did rendering lambda calculus usable as a computer language.

At least three other factors may be credited with responsibility for slowing down the development of Prolog: (1) the absence of a corpus of interesting examples demonstrating the novel usage of the language; (2) the unavailability of fast computers with substantial

main memories, as well as optimizing compilers enabling the practical use of Prolog in writing larger programs; and (3) the existence of better compilers and environments for the then more mature language Lisp, which had already proved its value in symbolic processing.

Most knowledgeable Prolog programmers would agree that just writing the procedure `append` in Prolog and exploring its use in writing other programs would be a substantial step toward establishing the set of convincing examples of the language's usefulness. The set of examples using `append` was developed at Marseilles and then expanded with the ongoing interactions with Edinburgh. The lack of adequate equipment must have hindered the Marseilles group. Edinburgh was more fortunate, and in my view, that accounted in part for the very successful work of Warren who, around 1977, was able to show that Prolog programs could achieve efficiencies comparable to those of Lisp.

The success of a language depends greatly on the development of a handful of interesting (and preferably short) programs that show its expressive power.

UNIFICATION AND CONTROL

The original unification algorithm proposed by Robinson included a special test called the `occur` test that prevented circular structures from being constructed by the algorithm.

The elimination of the `occur` test, originally suggested by the Marseilles group, was both a daring step and a pragmatic move toward decreasing the time taken by unification. This was risky, since the elimination could result in logically unsound results and, unless the necessary precautions were taken, the programs could enter infinite loops during unification or printing. An analogous situation occurs in Pascal-like programs in which run-time checks are introduced by compilers to test if an array index remains within its declared bounds. C. A. R. Hoare compared the elimination of such tests after debugging to "wearing life jackets during near-shore drills, but disregarding them in high-seas." In the case of the `occur` test, however, its elimination enabled the solution of problems using a reasonable amount of computation.

Nonetheless, Colmerauer was dissatisfied with the elimination of the `occur` test. By the late 1970s, he was engaged in developing a modified unification algorithm in which circular structures could be unified. His claim was that these structures play an important role in representing graphs (such as those describing flowcharts, and transition diagrams for finite state automata) and should not be avoided. His new algorithm is described in [10] and [11] and has now been incorporated in several interpreters. In the algorithm the notion of the most general unifier [36] is replaced by that of *solvable constraints.*

The `cut` is another feature that was originally introduced by the Colmerauer group, but for which, as mentioned, Colmerauer recommends spare usage. The `cut` is indispensable since it provides the ability to define negation by failure and the optimization of programs by supplying information to the interpreter that parts of the search space need not be inspected. As Kowalski aptly puts it, *Algorithm = Logic + Control*, and the `cut` is one of the few existing resources in Prolog to control *Control.*

We now return to the subject of unification and its extensions. Besides incorporating the capabilities of handling circular structures, Colmerauer also added to his unification algorithm the predicate `diff(A, B)`, which is based on Roussel's work on equalities in theorem proving [38]. This predicate requires a nontrivial implementation. Most available interpreters adopt a simplistic version of `diff` that fails if either one or both of its arguments are unbound, or bound to different ground terms. In Colmerauer's version of the algorithm, sets of equations and inequations are kept by the interpreter, and failure only occurs when the sets are unsatisfiable [11]. This manner of handling unification using equations is not unlike that proposed by Colmerauer's fellow countryman, the famous logician J. Herbrand. For the latter, unification (finding an element of the Herbrand universe) entailed finding a solution to a system of equations.

The availability of predicates expressing constraints (like `diff`) allows a programmer to increase the "purity" of Prolog programs by avoiding the `cut` and possibly increasing the potential of performing inverse computations. For example, if a predicate `p` is defined by

$$\begin{aligned} &p \leftarrow c_1,\ p_1',\ \ldots \\ &\vdots \\ &p \leftarrow c_n,\ p_n',\ \ldots \end{aligned}$$

and the conditions c_i are all mutually exclusive, then `cuts` become unnecessary.

It should be stressed that in logic programming unification and control are closely related. Normally, backtracking occurs when unification fails; otherwise, computation proceeds in the forward mode. Therefore, by extending the capabilities of unification one also extends the control features of the language.

Colmerauer's present goal is to extend the unification algorithm to cover linear inequations over the rationals, Boolean equations, and equations on special (linear-like) types of strings. In his forthcoming version of Prolog III, backtracking only occurs when the set of inequations and equations becomes unsatisfiable. In this respect Prolog's notion of variables becomes the

same as that of a variable as used in mathematics. Besides opening new horizons for the language, investing research efforts in extending unification is also wise: Even if other subsets of first-order predicate calculus are found to be more powerful than Horn clauses, it is very likely that the proposed extensions for unification would be applicable to those subsets as well.

CONCLUSIONS

This paper has presented a historical review of the origins and development of Prolog as a programming language. It is worth noting that during their research effort both Colmerauer and Kowalski recall rather precise moments in which they had the sudden insight of having discovered a fascinating novel way of expressing computer programs. Therefore, *discovery* in language design is not unlike discoveries occurring in fields such as physics or mathematics. This reinforces the thesis that discovery, rather than invention, lies at the heart of creative computer language design. I hope to have provided the sequence of closely related research efforts and events so that the reader can infer that indeed Prolog would have had its place in computer science regardless of its discoverers. In a similar vein, I have had the opportunity to ask McCarthy the analogous question vis-à-vis Lisp. His answer was unequivocal: Yes, Lisp would have existed anyway.

Such a determinism says a great deal about the nature of computer science. It indicates there is a basic set of concepts in this science that keep recurring in a variety of forms (e.g., grammars, nondeterminism, pattern matching). It also manifests a great affinity between computer science and mathematics, whose practitioners are almost unanimous in their praise for the beauty of certain theorems. The elegance and simplicity of Prolog will undoubtedly be judged in a similar way by computer scientists.

Acknowledgments. I wish to express my wholehearted thanks to Alain Colmerauer for helping me make precise some of the factual material described in this paper and for providing an improved version of the SL prover. I also had the opportunity to discuss candidly with Robert Kowalski his own views about the origins of Prolog; the conversations I had with him provided me with further insight into this matter. Both Alain and Bob read the original version of this paper at least twice and proposed numerous significant and detailed suggestions that were gladly incorporated into the revised version. The collaboration of my colleague Tim Hickey was also invaluable: His careful reading of the manuscript and thoughtful comments consistently resulted in a better presentation and increased accuracy. My thanks are also extended to Jean Louis Lassez for his helpful remarks.

A final note: This paper survived numerous revisions, several of them with the help of refinement operators provided by Alain, Bob, and Tim. Hopefully, we have at last reached a fixed point.

REFERENCES

Note: Reference [12] is not cited in text.

1. Bauer, F.L. *Historical remarks on compiler construction.* In *Advances in Compiler Construction.* Lecture Notes in Computer Science, vol. 21. Springer-Verlag, New York, 1974, pp. 603–621.
2. Boyer, R.S., and Moore, J.S. The sharing of structure in theorem proving programs. In *Machine Intelligence,* vol. 7. M. Melzer and D. Michie, Eds. Edinburgh University Press, Edinburgh, U.K., 1972, pp. 101–116.
3. Brooker, R.A., and Morris, D. A general translation program for phrase structure languages. *J. ACM 9,* 1 (Jan. 1962), 1–10.
4. Brooker, R.A., MacCallum, I.R., Morris, D., and Rohl, J.S. The compiler-compiler. *Annu. Rev. Autom. Program. 3* (1963), 229–275.
5. Clark, K. Negation as failure. In *Logic and Data Bases,* H. Gallaire and J. Minker, Eds. Plenum, New York, 1978, pp. 293–322.
6. Cohen, J. *Describing Prolog by its interpretation and compilation. Commun. ACM 28,* 12 (Dec. 1985), 1311–1324.
7. Colmerauer, A. *Total precedence relations. J. ACM 17,* 1 (Jan. 1970), 14–30.
8. Colmerauer, A. Les Systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur, Rep. 43, Dept. of Computer Science, Univ. of Montreal, Quebec, 1970.
9. Colmerauer, A. Les grammaires de métamorphose. Groupe d'Intelligence Artificielle, Univ. of Marseilles-Luminy, France, 1975. (Also: Metamorphosis grammars. In *Natural Language Communication with Computers,* L. Balc, Ed. Springer-Verlag, New York, 1978), pp. 133–189.
10. Colmerauer, A. Prolog and infinite trees. In *Logic Programming,* K. Clark and S. A. Tarnlund, Eds. Academic Press, New York, 1982, pp. 231–251.
11. Colmerauer, A. Equations and inequations on finite and infinite trees. In *Proceedings on the International Conference on Fifth Generation Computer Systems* (Tokyo, Japan, Nov.). ICOT, Tokyo, 1984.
12. Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. Un système De communication homme-machine en Français. Res. Rep., Groupe Intelligence Artificielle, Univ. Aix-Marseille II, France, 1973.
13. De Chastellier, G. and Colmerauer, A. W-grammar. In *Proceedings of the ACM Congress* (San Francisco, Calif., Aug.). ACM, New York, 1969, pp. 511–518.
14. Dijkstra, E.W. ALGOL 60 translation. *ALGOL Bull.* (supplement) *10* (1960). (Also: Recursive programming. *Numer. Math. 2* (1960), 312–318.)
15. Floyd, R.W. Syntactic analysis and operator precedence. *J. ACM 10* (1963), 316–333.
16. Floyd, R.W. Nondeterministic algorithms. *J. ACM 14,* 4 (Oct. 1967), 636–644.
17. Green, C.C. Theorem proving by resolution as a basis for question-answering systems. In *Machine Intelligence,* vol. 4. B. Melzer and D. Michie, Eds. Edinburgh University Press, Edinburgh, U.K., 1969, pp. 183–205.
18. Griffith, T.V., and Petrick, S.R. On the relative efficiencies of context-free grammar recognizers. *Commun. ACM 8,* 5 (May 1965), 289–300.
19. Hill, R. LUSH-resolution and its completeness. DCL Memo 78, Dept. of Artificial Intelligence, Univ. of Edinburgh, U.K., 1974.
20. Hopcroft, J.E., and Ullman, J.D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Mass., 1979.
21. Irons, E.T. The structure and use of the syntax directed compiler. *Annu. Rev. Autom. Program. 3* (1963), 207–227.
22. Knuth, D.E. On the translation of languages from left to right. *Inf. Control 8,* 6 (1965), 607–639.
23. Knuth, D.E. Semantics of context-free languages. *Math. Syst. Theory 2,* 2 (1968), 127–145.
24. Kowalski, R. Predicate logic as programming language. In *Proceedings of IFIP,* 74 (1974). North Holland Publishing Co., Amsterdam, 1974, 569–574.
25. Kowalski, R. *Logic for Problem Solving.* North-Holland, Amsterdam, 1979.
26. Kowalski, R. The early history of logic programming. Dept. of Computing, Imperial College, London, Oct. 1984.
27. Kowalski, R., and Kuehner, D. Resolution with selection function. *Artif. Intell. 2,* 3 (1970), 227–260.
28. Loveland, D.W. A simplified format for the model elimination theorem-proving procedure. *J. ACM 16,* 3 (July 1969), 349–363.
29. Loveland, D.W. Automated theorem proving: A quarter-century review. *Am. Math. Soc. 29* (1984), 1–42.
30. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (Apr. 1960), 185–195.
31. McCarthy, J. History of Lisp. *SIGPLAN Not.* (ACM) *13,* 8 (1978), 217–222.

32. Naur, P. Ed. Revised report on the algorithmic language ALGOL 60. *Commun. ACM 6*, 1 (Jan. 1963), 1–17.
33. Paul, M. A general processor for certain formal languages. In *Proceedings of the Symposium on Symbolic Languages in Data Processing* (Rome, Italy). Gordon and Breach, New York, 1962, pp. 65–74.
34. Pereira, F.C., and Warren, D.H.D. Definite clause grammars for language analysis. *Artif. Intell. 13* (1980), 231–278.
35. Pereira, F.C., and Warren, D.H.D. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics* (Cambridge, Mass.). Association for Computational Linguistics, 1983, pp. 137–144.
36. Robinson, J.A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.
37. Robinson, J.A. Logic programming—Past, present and future. *New Generation Comput. 1* (1983), 107–124.
38. Roussel, P. Definition et traitement de l'egalité formelle en démonstration automatique. Thesis, Faculté des Sciences, Univ. d'Aix-Marseille, Luminy, France, 1972.
39. Roussel, P. Prolog: Manuel de référence et d'utilisation. Groupe d'Intelligence Artificielle, Univ. d'Aix-Marseille, Luminy, France, 1975.
40. Samelson, K., and Bauer, F.L. Sequential formula translation. *Commun. ACM 3*, 2 (Feb. 1960), 76–83.
41. Van Emden, M.H., and Kowalski, R. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (Oct. 1976), 733–742.
42. Van Wijngaarden, A., Mailloux, B.J., Peck, J.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., and Fisker, R.G. Revised report on the algorithmic language ALGOL 68. *Acta Info. 5*, 1–3 (1975), 1–236.
43. Warren, D.H.D. Applied logic—Its use and implementation as a programming tool. Ph.D. dissertation. Dept. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, U.K., 1977. Also: Tech. Note 290, SRI International, Menlo Park, Calif., 1983.
44. Warren, D.H.D. Logic programming and compiler writing. *Softw. Pract. Exper. 10* (Feb. 1980), 97–125.
45. Wirth, N., and Weber, H. EULER: A generalization of ALGOL, and its formal definition: Part I. *Commun. ACM 9*, 1 (Jan. 1966), 13–23.

CR Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; F 4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic–*computational logic*; K.2 [**Computing Milieux**]: History of Computing—*people*; *software*

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Language design, list processing, logic programming, nondeterministic algorithms, SL resolution, theorem proving, unification

Author's Present Address: Jacques Cohen, Computer Science Department, 141 Ford Hall, Brandeis University, Waltham, MA 02254.