

A Consideration of Prolog

Jerry T. Ball
1989

ABSTRACT

Previous analyses of Prolog as a programming language have adopted both theoretical and empirical perspectives. Additionally, the uniqueness of the language has elicited many subjective comments. The result is an inconsistent collection of often contradictory statements. Within this muddle, this paper attempts to outline a more consistent statement of the usefulness of Prolog as a programming language. The analysis is mostly empirical and considers the features of Prolog which have or will prove useful for specific applications areas. A discussion of similarities and differences between Prolog and Lisp is also presented. Finally, some subjective statements about the future of Prolog are considered

A Consideration of Prolog

1. Levels of Analysis

An analysis of Prolog as a programming language can be conducted on several quite distinct levels. On one level, one can look at an assortment of problems that have been solved using other programming languages, attempt to solve them using Prolog, and compare the resulting solutions (or failures) with the other languages along such dimensions as efficiency of code, speed of execution, reliability, understandability (of the code), etc. Then based on this experience one can make generalizations about the relative usefulness of the respective languages. One might call this approach the empirical method. On a different level, one can look at the theoretical roots of the respective languages and draw conclusions about the elegance and usefulness of the languages based on the theories which led to their implementation. This approach can be called the theoretical method. Each of these two approaches provides a different perspective, but if the approaches are valid, then the results of the analyses should be consistent. Unfortunately, this is not the case. Prolog has proponents and detractors who support their positions with both theoretical and empirical arguments. Further, Prolog has also been examined on a quite different level which relates to the impact Prolog has had on the Artificial Intelligence and computing community. On this level, the subjective level, researchers have provided their subjective evaluations of Prolog's impact and potential for future impact. On this level inconsistencies abound and Prolog might be said to be "all things to all people".

A review of the literature reveals that most articles which analyze Prolog in isolation or in comparison with other languages (typically functional programming languages like Lisp) do so on one of these three levels with perhaps some mixing of theory into the empirical approach. For example, Carl Hewitt's (Hewitt, 1985) article, "The Challenge of Open Systems" is a theoretical analysis of logic programming methods. He argues that such methods are inadequate for modeling inconsistent worlds and further that intelligent systems must deal with inconsistency. Thus, from a theoretical perspective he argues against the usefulness of logic programming languages within AI. He makes the strong statement that "logic programming has some fundamental limitations that preclude its becoming a satisfactory programming methodology" (ibid 1985, p. 239). It is interesting to note that this is the position of the designer of Planner, a logic programming language and predecessor of Prolog.

From a more empirical perspective, Drew McDermott (1980) discusses various features of Prolog which make it a useful language for certain applications. He notes that Prolog's provision of an excellent pattern matcher and assertional database make it unnecessary for the user to implement them. He notes that the efficient implementations of Prolog achieved in the last few years make it competitive with Lisp for many applications. He provides several examples of Prolog code and discusses the elegance of the code. Not surprisingly, he notes that Prolog is not efficient for performing such operations as finding the roots of quadratic equations. On the other hand, the

implementation of the relation “append” is quite elegant reflecting the power of Prolog’s pattern matcher. He discusses techniques for getting around perceived weaknesses of Prolog’s localized variables and global backtracking. Global variables can be simulated by the use of “assert” and “retract”, and the “cut” is provided to control backtracking as needed. As further empirical evidence of the usefulness of Prolog he discusses an informal experiment at Edinburgh in which two groups of students were taught AI in POP-2 (a Lisp-like language) and Prolog, respectively. “After two months, the POP-2 group was writing pattern matchers and the PROLOG group was writing natural-language question answerers. Once they had learned PROLOG, the low-level stuff was already there” (ibid, p. 18). McDermott concludes that Prolog is a language that competes “pretty well” with Lisp.

On the subjective level Prolog is the center of considerable controversy. Some proponents have recklessly proclaimed Prolog’s virtues, thereby generating significant backwash from disbelievers. Robert Kowalski (1980, p. 44) has gone so far as to state

There is only one language suitable for representing information – whether declarative or procedural – and that is first-order predicate logic. There is only one intelligent way to process information – and that is by applying deductive inference methods. The AI community might have realized this sooner if it weren’t so insular. The database community, for example, learned its lesson several years earlier.

Researchers like Wendy Lehnert (1984) have responded to the smugness of such statements by replying

PROLOG appeals to our intellectual insecurities and desire for scientific respectability. It appeals to a rather seductive ‘blind faith’ mentality that takes hold in the face of extremely difficult problems. It allows us to avoid truly hard problems in AI at the same time that it lends dignity to narrower issues...with diversions like PROLOG around, it is difficult to keep a clear perspective on exactly what needs to be accomplished in the AI community.

Lehnert further contends that “it is not the case that PROLOG makes any real contribution to any of the [AI] problems: it merely makes it easy to program some very old ideas” (ibid, 1984). How then is one to sort out the usefulness of Prolog as a programming language in the face of such strong and contradictory claims?

2. Basis for Comparison with Other Languages.

It is a generally accepted principle that all programming languages are computational equivalent in power to a Turing Machine and therefore to each other. However, this computational equivalence has long been disregarded when comparing different languages. In the SIGART Newsletter Special Issue on Knowledge Representation, Brachman and Smith (1980) present a hypothesis for discussion that “such ‘Turing equivalence’ is a vacuous notion when dealing with representation languages” (ibid 1980, p. 127). The authors go on to suggest that the usual method of showing equivalence by

implementing language A in language B and vice versa does not preserve denotational correspondence (e.g. implementation does not in general preserve the identity of data structures), and therefore “does not constitute equivalence between representation languages” (ibid, p. 128). That is to say that in order for two representation languages to be equivalent they must be both behaviorally and structurally equivalent. Turing Machine equivalence is limited to behavioral equivalence. Thus, we can compare languages in terms of their structural differences. There seems to be some correspondence between the terms behavioral/structural used in the SIGART article and the control logic/data structures of actual programming languages. However, since different languages may have different control structures (e.g. recursion in Lisp, pattern matching/backtracking, in addition to recursion, in Prolog) as well as different data structures, the exact nature of this correspondence is clouded.

Some researchers have gotten round the inadequacy of computational power as a basis for comparing programming languages by adopting a less well-defined notion of expressive power. According to Reddy (???, p. 195), “Expressive power, unlike computational power, is not a simple and precisely defined quality. It refers to several traits such as length of programs, ease of writing programs, readability, etc.”. These researchers concentrate on the empirical analysis of language features in the absence of sound theoretical criteria for distinguishing between languages.

Perhaps, I can sum up this discussion by suggesting that since representation languages are theoretically equivalent, the only bases for comparison are empirical differences. By saying this I am suggesting that Hewitt’s theoretical arguments against the inadequacy of logic programming languages can be reduced to empirical arguments about the difficulty of using such languages to model open systems. That is, I believe it can be shown that logic programming languages are in fact capable of modeling open systems. The question is therefore not whether they can be used to model open systems, but whether or not they are suitable for doing so. Thus, this paper’s discussion of Prolog’s usefulness as a programming language and its comparison with Lisp consists mainly of an analysis of the empirical features of Prolog and its differences from Lisp.

3. Development of Prolog

I am apparently not alone in thinking that the development of Prolog to date resembles the early development of another controversial programming language – Lisp. According to Cohen (1985, p. 1311) “...since its conception, Prolog has been evolving in a manner not unlike the early evolution of LISP”. Cohen goes on to note that both Prolog and Lisp are in fact still evolving. I think this development process can be viewed as a struggle between two camps: the purists (theoretically speaking) and the empiricists. In the case of Lisp it is apparent that many empirically useful constructs have been added to the language which was and continues to be based on Church’s Lambda Calculus. Most Lisp programmers begin by learning to program in Lisp and only later develop an understanding of its theoretical underpinnings. As such they are not willing to restrict themselves to the use of theoretically pure constructs when other more immediately useful constructs are available. On the other hand, there is a small group of diehard purists who consider these added constructs to contaminate the original language and reduce its theoretical elegance. Prolog faces much the same dichotomy. On the one hand,

researchers like McDermott (1980, p. 18) contend that “the notion that programming in PROLOG is programming in logic is ridiculous”. And Barbuti et al. (???, p. 160) note that

A fully declarative language [e.g. Prolog]...does not seem to be adequate as a machine language. In fact, both at the system and at the application level there exist software components which are intrinsically procedural. Defining them declaratively would, on one side, be unnatural, and, on the other side, make them less efficient.

People who hold this view tend to look favorably upon extensions to Prolog to make it more generally useful as a programming language. On the other hand, Kowalski contends that while Prolog does have certain extralogical features (e.g. the cut, assert, retract), these features are not essential to the language. Thus, according to Kowalski (1985, p. 15), “IC PROLOG does away with the extralogical features of other PROLOG implementations. It incorporates directly a number of extensions, such as indexing, corouting, negation by failure, conditionals and the construction and manipulation of all solutions of a subgoal which are compatible with the semantics of logic”. Kowalski contends that programming in IC PROLOG is semantically consistent with logic. Nonetheless, Kowalski concedes that Prolog is an evolving language subject to improvement and change. He notes that “PROLOG is the first and most important logic-programming language, and it provides a tantalizing preview of the more powerful logic-programming languages of the future” (ibid, p. 176). Indeed, much of the current literature on Prolog is devoted to articles suggesting improvements to the existing implementation. Some of these recommendations are discussed later in this paper.

4. Prolog Features

Prolog is a distinctive programming language with numerous features which serve to distinguish it from other programming languages. In this section we will examine several of those features in terms of their usefulness as well as limitations.

4.1 Built-in Control Structure.

Unlike other programming languages, Prolog has a built-in control structure which drives all Prolog programs. This control structure is composed of backward reasoning applied to Horn Clauses of predicate logic through the use of unification-based pattern matching. If a conjunct on the right-hand side of a clause under consideration can be matched to the left-hand side of another clause, then this conjunct is replaced by the right-hand side of the matching clause. Since Horn Clauses are of the form:

Conclude A if B and C,

this process is an implementation of backward reasoning (i.e. in order to conclude A one must first prove B and C). This built-in control structure is extremely useful for certain applications, but highly constraining for others. It is extremely useful for rule-based

applications since the programmer need not code the control logic and can concentrate on developing the rule base to which the control logic will be applied. Thus, a major portion of the development of such a system is eliminated. The built-in control structure also changes the nature of the programmer's task. Instead of developing an entire system and considering trade-offs in the complexity of the control structure vs. the size of the knowledge base as part of the design process, the programmer is relegated designing a knowledge base which can be suitably accessed by the built-in control logic. While the programmer's task appears to be lessened, this will only be the case if a knowledge base can be designed which is both suitable for use with Prolog's control logic and adequate to model the domain of knowledge under consideration. If either of these requirements are difficult to satisfy, the programmer's task will be correspondingly difficult.

4.2 Incremental Development of Programs.

Prolog programs are composed of clauses which can be incrementally added to or deleted from the program. This is a typical quality of rule-based systems in general, which makes it easy to develop a basic program which can later be expanded to handle larger and larger sets of rules. Unfortunately, this incremental development capability is not as useful as it at first appears. Since the rules in a Prolog program can interact with each other in complex ways, it is not always possible to determine the impact of rule addition or deletion in large programs. While it may be easy to add a rule, the rule addition may require modification of many other rules before it is appropriately integrated into the system. It is this complex interaction which makes large rule-based programs difficult to modify and which will be the case for large Prolog programs as well.

4.3 Logic Basis of Prolog

Prolog's theoretical basis gives it credence as a programming language, but how does this basis effect its empirical usefulness? Many researchers content that the basis in logic makes it easier to verify the correctness of programs and should therefore reduce the number of program errors. According to Cohen (1985, p. 1323), "Having a foundation in logic, Prolog encourages the programmer to describe programs in a logical manner that facilitates checking for correctness and consequently reduces the debugging effort". However, Kluzniak and Szpakowicz (1983, p. 81) argue that "there is no particular reason why a sizable set of formulae [predicate calculus] should contain less bugs than a sizable set of program statements".

It can also be argued that logic programming is closer to human thinking than other programming languages. Here again though, many people (myself included) only feel they understand a program when they can visualize the actual processing which goes on during execution of that program. Prolog's declarative logic basis avoids the need to consider this processing, and it is a moot point as to whether it is more understandable as a result. Kluziak and Szpakowicz (ibid, p. 82) suggest that "the key to successful programming lies in the programmer's ability to rapidly alternate between the two viewpoints [operational vs. relational view] while developing the program".

4.4 Uniform Data Type

According to MacLennan (1983, p. 505), “The small number of built-in data types and operations in PROLOG is an example of the Simplicity Principle. The uniform treatment of all data types as predicates and terms is an example of the Regularity Principle”. Complying with such principles is desirable for programming languages since it makes them easier to use and reduces the number of syntax errors which are likely to occur. Prolog’s uniform representation goes beyond that of more traditional programming languages in that both data and program are uniformly represented. This makes it possible to treat data as program and vice versa. Thus, a program which generates data can later turn around and treat that data as program. According to Minsky (1984, p. 249) “...to make an intelligent system, you eventually want the system to be able to learn to improve itself. In order to do that, that system must write its own little programs”. Lisp and Prolog are the two notable languages with such a capability. This uniform treatment of data in Prolog also makes it possible for Prolog to maintain its own internal database and in general blurs the distinction between program and data.

4.5 Efficiency of Implementation.

The viability of any programming language is inextricably tied to the efficiency with which it can be implemented. Lisp has long been shunned outside the field of AI because of real and perceived inefficiencies in implementation. The survival of older languages like FORTRAN is in part due to efficiency of implementation which have outweighed shortcomings of the language itself. Early Prolog implementations were notably inefficient. In the last few years improved implementations of interpreters and compilers have brought the performance of Prolog in line with that of Lisp. While this may impress AI researchers, mainstream computer scientists are less likely to be impressed. And the implementation of efficient versions of Prolog is not without cost. According to McDermott (1980, p. 19)

PROLOG may be seen as an effort to simplify a theorem prover down to the point where it is as efficient as a programming language. I approve of this, but think its inventors may have gone a little bit too far. They concentrated on implementing one basic idea with more and more efficiency, and have turned their backs on ideas that may in the long run pay off.

Prolog’s suitability as a language for parallel processing (i.e. “the theory of logic programming assures us that the order of ‘evaluation’ of literals and clauses is immaterial” – attributed to Wise by Kluzniak and Szpakowicz, 1984, p. 17) offers hope for significant gains in execution speed as a result of implementation on parallel machines. Whether such implementations will attain or exceed the performance of more traditional languages remains an open question.

4.6 Bidirectionality of Clauses

Pure Prolog clauses (e.g. without cuts, I/O, assert, retract) impose no directionality on their execution. That is to say, it is not necessary to determine in advance which

parameters are input and which are output. At the time a clause is invoked, the directionality can be determined by the identification of which parameters are variable (output) and which are not (input). Thus, given the existence of an “append” relation we can invoke it with the following clauses:

- (1) `append([a,b],[c,d],X).`
- (2) `append(X,Y,[a,b,c,d]).`

In case (1) the input parameters, [a,b] and [c,d], will be combined to form the output [a,b,c,d]. In case (2), the input parameter [a,b,c,d] will be broken down into all possible combinations of lists which when combined form [a,b,c,d]. In procedurally oriented and functionally oriented languages (e.g. Lisp), separate procedures would be required to handle these two cases, while in Prolog this is unnecessary. Bidirectionality is thus a unique and potentially powerful feature of Prolog. Unfortunately, to maintain bidirectionality Prolog programs must avoid the extralogical features added to the language to make it more efficient and useful. Use of the “cut” in a bidirectional program is disallowed and the efficiencies gained by its careful use are lost. Additionally, bidirectionality of I/O is not possible in most cases. Prolog purists see the loss of bidirectionality in Prolog programs as a result of its contamination with extralogical features and use this as a basis for arguing against their inclusion in the language. Empiricists suggest that while bidirectionality is a nice idea, its use in practice is limited. According to McDermott (McDermott, 1980, p. 19).

It is often claimed that a PROLOG program can be used in more than one way, and simple ones can...everyone quickly learns how seldom a program works this way. It does tend to be true that a program used to generate a result can be used to check one as well, but any other use will often introduce gross inefficiencies or infinite loops.

4.7 Length of Prolog Programs

According to Cohen (1985, p. 1322), “Prolog programs are usually significantly shorter than programs written in other languages (typically 5-10 times shorter)”. He later adds, “The conciseness of Prolog programs, with the resulting decrease in development time, makes it an ideal language for prototyping” (ibid, p. 1323). But conciseness of code is not always a desirable feature. APL, an extremely concise language, is often criticized for the lack of understandability of its code.

4.8 Features that Prolog Lacks

A great deal of the Prolog literature is concerned with improvements to this evolving language. Apparently, researchers are impressed with the power and potential of Prolog as a programming language, but find in actual practice that it lacks certain features which are provided in other languages and which can be simply added to these languages. Unfortunately, it is more difficult to add such features to Prolog. Ennals, Briggs and Brough (1984, p 383) note that “particular problems remain, especially for users with a

background in conventional programming. The current limitations of Prolog provide some traps for the unwary". A survey of the IEEE 1984 Symposium on Logic Programming shows Nakashima (1984, p. 126) discussing Prolog/KR, "an extension of Prolog towards knowledge representation"; Kahn (1984, p. 242) proposing "a control predicate to augment the inadequate control primitives of Prolog"; Tamaki (1984, p. 259) introducing a new language which is a "usual definite clause language of first order logic but includes a special predicate called reducibility"; Zaniolo (1984, p. 265) introducing primitives into Prolog to support object-oriented programming; etc. While it is true that the incorporation of all recommended additions to Prolog would make it the PL/1 of logic programming languages, it is also true that many of these features are highly desirable. For example, Warren (1982) notes that Prolog is in need of a set expression predicate (Kahn's proposed "collect" predicate is similar) to allow the language to handle queries of the form:

How many people are known to drink milk?

At present, Prolog has no means of answering this query since it has no capability for collecting solutions as part of its backtracking algorithm. Such a feature can easily be coded into another programming language (e.g. Lisp), however, in Prolog it requires revision of the language itself (if it is to be at all efficient).

Another proposed extension to Prolog is the allowance of variable predicates. Variable predicates would extend the range of information retrievable from a given database (see the section on database applications). Warren (1982, p. 449) notes that "the introduction of predicate variables...seems to have a certain elegance...however, if predicate variables are used in more than small doses, the program becomes excessively abstract and therefore hard to understand". Nonetheless, this capability is already being introduced into Prolog implementations.

There are many features that will need to be added to the language to support parallel computations. The result will be a logic programming language with many extralogical features to control parallel processing.

5. Comparison of Lisp and Prolog

The comparison of Lisp and Prolog reveals an interesting theoretical vs. empirical dichotomy. Prolog with its basis in logic and relations is theoretically more general than Lisp with its basis in functional notation. However, in terms of their use as programming languages, Lisp provides more generality in the selection and use of control structures making it a more general programming language than Prolog. The lack of generality of Prolog is the result of its built-in control logic. One is tempted to respond that this cannot be so since Lisp can be implemented in Prolog and therefore Prolog is at least as general as Lisp. However, as argued earlier in this paper, the ability to implement one language in another does not demonstrate both structural and behavioral equivalence, nor does it consider expressive power. By implementing Lisp in Prolog we increase the generality of the Prolog implementation to encompass Lisp, but in so doing we no longer have a pure Prolog implementation. In fact, many researchers have suggested that the combined implementation is more expressive and therefore more useful than either separate Prolog

or Lisp implementations. Of the existing combined implementations, logic programming has been implemented in a functional programming language and not vice versa (e.g. Robinson and Sibert, 1982).

In terms of features, Prolog and Lisp are closer to each other than they are to mainstream computer languages. It can be argued that Prolog's relations are a generalization of Lisp's functional notation. Lisp's "append" function and Prolog's "append" relation are frequently compared in the literature and serve to reveal this similarity of notation and resulting programming style. Both Prolog and Lisp include features for list processing. The actual operation of Prolog's control logic looks very much like recursion. Prolog clauses typically include a base case for terminating the program just as Lisp uses a statement to terminate the recursive execution of a function. Both Prolog and Lisp are able to treat program as data and vice versa (this may well be the main reason these languages are used for AI applications). How then do Prolog and Lisp differ? According to Reddy (???, p. 195):

It may be seen that the fundamental difference between logic and functional languages is that logic programs are transparent to the 'directionality' of their computations, whereas functional programs explicitly incorporate this directional information. One may argue that this directional transparency makes logic programming easier to write or understand.

While directionality is an important characteristic which stems from the mathematical differences between relations and functions, I do not see this as the most important difference. Rather, in terms of its impact on programming, I consider the built-in control logic of Prolog to be paramount. Prolog programmers have a completely different perspective on the programming problem. The Lisp programmer is responsible for designing and coding the algorithm to solve a particular program. The Prolog programmer is concerned with writing declarative statements which are manipulated by a built-in generalized search algorithm. Lisp programmers may consider many different control strategies before adopting a particular one to solve a given problem. As part of this design process, Lisp programmers must consider the trade-offs between using simplified searching algorithms which increase the importance of knowledge representation and more complex algorithms which increase the importance of the control structure. Lisp programmers are free to consider alternative formalisms for knowledge representation. On the other hand, Prolog programmers are restricted in terms of both the control structure available and the knowledge representation scheme allowed. Whether the freedom of Lisp turns out to be a burden or an advantage will depend largely on the nature of the problem under consideration.

Lisp has been shown to be amenable to structured programming techniques including modularization, hiding of variables and structures, etc. According to Cohen (1985, p. 1323), "The nonexistence of block structure, scoping, and type checking of variables may deter potential users from writing very large Prolog programs". Lisp programs, and AI programs in general, tend to be large, as the problems they attempt to solve are complex and difficult. If Prolog is to be a useful AI language, researchers must be capable of coding large programs in it.

6. Applications of Prolog

Prolog's built-in features make it a useful language for particular applications. Prolog's strong unification-based pattern-matching capability has natural applications in database retrieval applications. The fact that Prolog maintains its own internal database is also useful in this realm. Prolog clauses can be directly interpreted as if/then rules making Prolog useful as a language for implementing rule-based systems like expert systems. Prolog's clauses can also be interpreted as grammar rewrite rules making it useful for the implementation of syntactic analyzers. To the extent that semantic analysis can be formalized as rewrite or if/then rules, Prolog will also prove useful for semantic analysis. Each of these application areas is discussed further below.

While Prolog has been shown to be useful for the applications mentioned above, Kluzniak and Szpakowicz (1984, p. 79) caution that "...it is only prudent to suspect that even Prolog might prove almost useless when applied to work it was not meant to do".

6.1 Linguistic Analysis.

Since the Horn Clauses of Prolog can be interpreted as rewrite rules, Prolog's use as a language for linguistic analysis has long been realized. In fact, Colmenaurer, the creator of Prolog, originally applied Prolog to the solution of Natural Language Processing problems. Despite Prolog's obvious applicability in this application area there are some problems. Prolog programs are incapable of analyzing language fragments which violate the grammar used to analyze the fragment. If an error occurs at the beginning of a text fragment, analysis of the rest of the fragment is not possible. While it may be possible to modify the Prolog interpreter to accept some forms of invalid input, such revisions would require extensive changes to the interpreter and one is left wondering if the use of a more general programming language might be preferable.

Of late, Prolog is beginning to be used in the development of semantic analyzers. Xiuming Huang (1985) and Dan Fass (1988) of the Computing Research Laboratory at New Mexico State University are both researching this potential. Their original efforts began with the conversion into Prolog of a semantic analyzer based on Wilks' Preference Semantics (e.g Wilks and Fass, 1992) and previously coded in Lisp. The results of the research show that Prolog can in fact be used in the development of semantic analyzers and may be preferable to Lisp as a development language for this application. The results of their analyses might seem surprising. Preference Semantic is based on the selection of preferred combinations of word senses and larger units of meaning. There is, in general, no right or wrong reading of a chunk of text. Yet Fass and Huang have been able to implement Preference Semantics in a language which can only distinguish between statements which are true or false, valid or invalid. In order to accomplish this, Fass and Huang have built up a level of representation on top of Prolog which can in fact deal with questions of preference.

The bidirectionality of Prolog offers hope for the development of systems capable of both parsing and generating linguistic text with a single grammar. The efficiency and elegance obtainable by such a program is intriguing. However, in order to create bi-directional programs one must accept greater inefficiency in the backtracking mechanism of Prolog due to the required elimination of "cuts" and look-ahead checks. Xiuming

Huang (1984) is attempting to get around this problem as part of a bidirectional implementation of his XTRA system. Huang is using a grammar to analyze English which contains the cut symbol for efficiency, but is developing an identical grammar without cuts to be used to generate English. The need for two separate grammars could be eliminated if the Prolog interpreter were modified to allow two modes of operation: one in which the cut is executed and the other in which it is simply ignored. Whether bidirectionality is attainable, and if so, without too high an expense is an open question for research.

6.2 Database Applications.

Prolog's pattern matching capability makes it a useful language for database retrieval operations. The uniform representation of data and programs is also a useful feature. However, much of the current research in database systems deals with the development of database languages capable of accessing existing databases. The necessity of converting these databases into Prolog clauses has limited the usefulness of Prolog for such applications. The large size of databases is also a problem for Prolog. These problems may only be temporary in nature as programmers begin to use Prolog and create databases within it, and as larger Prolog systems become available.

On a lower level, Prolog does not allow the use of variable predicates. Thus, while queries of the form

loves(X,tom)

are allowed, queries of the form

VAR(mary,tom)

are not. The disallowance of the second form has to do with Prolog's logical basis. However, from the point of view of someone trying to access information from a database, the inability to access information using variable predicates is a shortcoming of the language. While Warren (1982) has shown that it is possible to get around this limitation and does not feel that variable predicates are necessary, their usefulness in data retrieval is obvious and future implementations will probably make them possible.

The use of Prolog for database applications overlooks many of the main topics of database systems research. According to Kluzniak and Szpakowicz (1984, p. 80)

There are numerous topics in database research (both theoretical and practical) that do not belong in the 'logic and database' area: recovery of information 'lost' due to hardware and software malfunction; locking protocols and the avoidance of deadlock; assuring adequate performance in the presence of large numbers of users that query or modify the database at the same time, etc.

On the other hand, these same researchers consider Prolog to be an excellent language for the development of prototypes (ibid):

The importance of Prolog as a tool for the construction of experimental database systems can hardly be overestimated. Many subtle and abstract problems of database design can be investigated in a practical setting, logic providing a welcome vehicle for expressing both theoretical considerations and tentative implementations

Prolog's usefulness as a language for developing both natural language parsers and database systems makes it especially useful for the development of natural language front ends to database systems. A Natural Language Interface (NLI) can be coded in Prolog which converts a natural language query into a Prolog query and then executes that query on a Prolog database. Wallace's (1984) QPROC is an example of such a system. At the Computing Research Laboratory, Ted Dunning has demonstrated a simple Prolog implementation of a menu-based natural language front end similar to Tennant's NLMENU system (Tennant et al. 1983). At present, Prolog's usefulness in this combined domain remains largely untested. Nonetheless, its potential is obvious and this may well prove to be a major area of application of Prolog in the coming years.

6.3 Expert Systems.

Prolog is an excellent language for the development of rule-based systems. I have argued elsewhere (Ball, 1985) that the most distinguishing characteristic of expert systems is that they are rule-based. However, other researchers have defined additional features which frequently occur in such systems:

- (1) model expertise in a limited domain
- (2) use probabilistic reasoning
- (3) are able to explain behavior
- (4) use domain specific problem solving strategies

To demonstrate the usefulness of Prolog for the general development of Expert Systems, one must show that these features can be modeled in Prolog. It is a basic assumption of most Expert Systems developers that expertise can be modeled in the form of rules. If this assumption is valid, feature (1) follows directly. Clark and McCabe (1982) have shown that (2) and (3) can be implemented in Prolog. Thus only (4) remains problematic. Prolog has a fixed control logic. The development of Expert Systems using Prolog will require programmers to adapt the system to the built-in logic. For those applications in which the control logic is suitable, Prolog will be a useful programming language. For other applications, the cost of adaptation to Prolog's control structure may be too great to be practical and other implementation languages will be more appropriate. This is much the same problem faced by the developers of Expert System shells—tool for the development of Expert Systems. If they provide specific features in the shell, the shell will prove useful for applications which can make use of those features. In providing specific features, they narrow the range of applications for which the shell is useful. Prolog, as an Expert Systems development language, falls somewhere in between general purpose programming languages like Lisp and specific Expert System shells like OPS5 or EMYCIN. It remains to be seen if the trade-off between specific features and general

programming constructs provided by Prolog will make it a useful language for the development of expert systems.

6.4 Parallel Programming.

Many researchers have expounded on Prolog's potential as a parallel programming language. Parallel implementations of Prolog are being and have already been developed (e.g. Parlog). The Japanese are hard at work developing a parallel machine designed to support Prolog as the major programming language. The parallel implementation of Prolog will greatly increase the speed of Prolog programs. However, parallel implementation alone will not overcome the limitations of combinatorial explosion for difficult problems. According to Kluzniak and Szpakowicz (1984, p. 73), "It is only after we learn to effectively decrease the size of the search space by conventional means that the potential of additional [parallel] hardware can make a qualitative difference in processing power". Additionally, the implementation of parallelism conflicts with some of the other desirable features of Prolog. According to Reddy (???, p. 196)

Parallel evaluation models for logic programming normally impose some kind of directionality on the programs they evaluate, because when two literals are resolved in parallel both of them should not be allowed to bind a shared variable...such a strong directionality constraint takes away the expressive power...and essentially makes them equivalent to...functional language[s].

7. Conclusions.

Prolog is a programming language which is here to stay (in one form or another). This in spite of comments by Marvin Minsky (1984, p. 249) to the contrary, "I think the Japanese will program in PROLOG for two or three years, write some very sophisticated programs, and then give up and go back to Lisp". The evidence in favor of Prolog's continued success is strong. Both Western Europe and Japan have invested heavily in developing efficient implementations which will compete with other programming languages. Researchers are continually expanding the realm of applications for which Prolog is a useful language. Indeed, if it survives nowhere else, it will survive as a component of a combined Lisp/Prolog system.

Prolog is what I would call a pseudo-general programming language. It provides many features which will prove useful for a large class of problems. However, the built-in control logic of the system will limit its usefulness to problems for which that logic is suitable. No doubt researchers will attempt to show that Prolog can solve any problem any other language can solve, but this is not at issue. At issue are the features provided by various programming languages, and how these features can be effectively used in the solution of different problems. While the class of problems for which Prolog is a useful tool will continue to expand, that expansion is not without limit, and researchers will have to learn how to best use this tool along with the other tools in the expanding toolbox of programming technologies.

While I have stated that Prolog is not without limitations, I must retract somewhat and suggest that those limitations are not so strict as would be supposed by those who take Prolog at its face value. The logical basis of Prolog is extensible to model real world problems. The implementation of Preference Semantics in Prolog is an example of this. The process of building higher level representations on top of lower level representations is ubiquitous in computer science and AI. The question is not whether or not this can be done using Prolog, but whether or not Prolog should be the language of choice for a specific application. Minsky (1984, p. 253) has argued that “the reason why LISP has retained its popularity in Artificial Intelligence is that it is not a language so much as a language that you write you own language in”. Prolog has a temporary advantage over Lisp in that it has several built in features which are of direct use in the development of certain applications, however, its broader use as a programming language will depend on the ease with which it can be extended to model problems for which the built in features are not directly applicable.

Bibliography

- Ball, Jerry (1985). *Natural Language Processing and Expert Systems*. Report No. MCCS-84-41. Memoranda in Computer and Cognitive Science. Computing Research Laboratory, New Mexico State University, Las Cruces, NM 88003.
- Brachman, R. J. and Smith, Brian (1980). Guest editors of SIGART Newsletter Special Issue on Knowledge Representation.
- Clark, K. L. and McCabe, F. G. (1982). "PROLOG: a language for implementing expert systems." In *Machine Intelligence*. Edited by J. E. Hayes and Donald Michie. New York: John Wiley.
- Cohen, Jacques (1985). "Describing Prolog by its interpretation and compilation." In *Communications of the ACM*, Dec 85, pp. 1311-1324.
- Ennals, R., Briggs J., and Brough, D. (1984). "What a naïve user wants from Prolog." In *Implementations of Prolog*. Edited by J. Campbell. New York: John Wiley.
- Fass, D. (1988). *Collative Semantics: a Semantics for Natural Language Processing*. Report No. MCCS-88-118. Memoranda in Computer and Cognitive Science. Computing Research Laboratory, New Mexico State University, Las Cruces, NM 88003.
- Genesereth, Michael, and Ginsberg, Matthew (1985). "Logic Programming." In *Communications of the ACM*, Sep 85, pp. 933-941.
- Huang, Xiuming (1985). *Machine Translation in the Semantic Definite Clause Grammars Formalism*. Report No. MCCS-85-7. Memoranda in Computer and Cognitive Science. Computing Research Laboratory, New Mexico State University, Las Cruces, NM 88003.
- Kahn, K. (1984). "A primitive for the control of logic programs." In *IEEE 1984 International Symposium on Logic Programming*. Silver Spring, MD: IEEE Computer Society Press, pp. 242-251.
- Kowalski, R. (1980). The SIGART Newsletter Special Issue on Knowledge Representation.
- Kowalski, R. (1982). "Logic as a computer language." In *Logic Programming*. Edited by K. Clark and S. Tarnlund. New York: Academic Press.
- Kowalski, R. (1985). "Logic Programmind." In *Byte*, Aug 85, pp. 161-176.

- Kluzniak, F. and Szpakowicz, S. (1984). "Prolog – a panacea?" In *Implementations of Prolog*. Edited by J. Campbell. New York: John Wiley, 1984.
- Lehnert, W. (1985). *The Prolog problem*. Report, Dept of CIS, University of Massachusetts, Oct 85.
- MacLennan, Bruce (1983). *Principles of Programming Languages: Design, Evaluation, and Implementation*. New York: Holt, Rinehart and Winston.
- McDermott, Drew (1980). "The Prolog Phenomenon". SIGART Newsletter N.72 July 1980 pp. 16-20.
- Minsky, M. (1984). "The problem and the promise." In *The AI Business, Commercial Uses of Artificial Intelligence*. Edited by P. Winston and K. Pendergast. Cambridge, MA: The MIT Press.
- Nakashima, H. (1984). "Knowledge Representation in Prolog/KR." In *IEEE 1984 International Symposium on Logic Programming*. Silver Spring, MD: IEEE Computer Science Press, pp. 126-130.
- Robinson, J. and Sibert, E. (1982). "LOGLISP: an alternative to PROLOG." In *Machine Intelligence 10*. Edited by J. Hayes and D. Mitchie. New York: John Wiley.
- Tamaki, H. (1984). "Semantics of a logic programming language with a reducibility predicate." In *IEEE 1984 International Symposium on Logic Programming*. Silver Spring, MD: IEEE Computer Science Press, pp. 259-264.
- Tennant, H. Ross, K. Saenz, R. Thompson, C. and Miller, J. (1983). "Menu-Based Natural Language Understanding." In *Proceedings of the 21st Annual Meeting of the ACM*.
- Wallace, M. (1984). *Communicating with Databases in Natural Language*. New York: John Wiley.
- Warren, D. and Pereira, L. (1977). "PROLOG – the language and its implementation compared with Lisp." *SIGPLAN Notices 12*, Volume 8, pp. 109-115.
- Wilks, Y. and Fass, D. (1992). "The Preference Semantics Family". In *Computers and Mathematical Applications*, Vol 23, pp. 205-221).
- Zaniolo, C. (1984). "Object-Oriented Programming in Prolog." In *IEEE 1984 International Symposium on Logic Programming*. Silver Spring, MD: IEEE Computer Science Press, pp. 265-270.