PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

IF PROLOG IS THE ANSWER, WHAT IS THE QUESTION?

Daniel G. Bobrow
Intelligent Systems Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304, USA
November 1984

Abstract: Knowledge programming, the keystone of the fifth generation project, requires specialized tools to help people represent and manipulate knowledge in the computer. Prolog systems provide some of these tools. This paper raises questions which suggest that logic programming should be combined with paradigms of function, object, rule and access oriented programming to facilitate the knowledge programming task. A second theme pursued here is the integration of these paradigms with each other, and within a flexible user friendly computing environment. Such an environment must provide source level debugging and monitoring facilities, analysis and performance tuning tools, and an extended set of user communication programs.

1. Introduction

Prolog technology has been adopted by the Japanese Fifth Generation Project as the kernel of the "official" programming language for research in knowledge based systems. Although this paper contains *some* direct criticisms of Prolog, its major thrust is to describe the requirements for a programming environment for the kinds of problems faced in the knowledge programming task. It illustrates by examples the need for these requirements, and shows how these they have been satisfied in some environments, including Prolog, though no single environment contains all the facilities suggested. A particular environment may be missing some facilities, and be particularly good in others.

One theme of this paper is the need for multiple paradigms of programming. Yokoi [1982] states that Prolog is being used as the basis for fifth generation programming because it "gives new paradigms of programming ... [which] make it much easier to deal with programs and programming". Logic programming provides a non-procedural representation of knowledge, combined with a powerful database search facility. I argue that although this combination is very powerful, it is inappropriate for some problems. By having a number of other programming paradigms as well, one can build more understandable programs more quickly. No single paradigm is appropriate to all problems, and powerful systems must allow multiple styles. Just as there many tools in a carpenters toolbox, each specialized to its purpose, there must be many in the programmer's kit. One should not be forced (metaphorically) to pry up nails with a screwdriver. Integration of multiple paradigms is illustrated by our experience with Loops [Stefik83, BobrowStefik83] and with examples from other systems.

The second theme of this paper is the integration of these multiple paradigms within a powerful environment which supports incremental development of programs. In knowledge based programming, we are usually trying to develop a system for which the requirements are not known in advance. The user may not know the requirements, or not know what data is required, not know efficient ways to represent that data to obtain appropriate answers in reasonable time, or may need to explore fundamental algorithms. Thus, such systems evolve over time, with the definition of the problem being refined in concord with the development of the program to solve the problem. Sheil [1983] refers to this as "exploratory programming". It is also the case that programs once successful are also thought of as a basis for more extended programs. As new requirements for the extension, or interactions with other programs are developed, the original program must change. The questions asked here about the environment are the result of wanting to support that kind of incremental, ever changing exploratory programming.

The paper has two sections dealing with each of these themes. The questions presented in *italics* are suggested by the material of that section, and each question should be considered a partial answer to

the title question. However, it is not Prolog that alone that must answer such questions satisfactorily, but any system that is designed to support programming of knowledge based systems.

2. Programming Paradigms

A programming paradigm or style of programming supports the expression of a programers intent. Some common programming paradigms are the function oriented paradigm of Lisp, the object oriented paradigm of Smalltalk, and the logic oriented paradigm of Prolog. A language supports a paradigm if it provides the primitives of that paradigm, composition methods, and an appropriate user language to make programs written in the paradigm clear. A language must also allow effective execution of programs written in that style, for quantitative changes in running time make for qualitative changes in a system.

In the Turing machine sense, all common programming languages are universal. However, different techniques for expressing the knowledge may be more "natural", depending on the form of the problem, and the persons view the problem. There is a tradeoff between uniformity of a single methodology, and the closer fit of different methodologies to a problem. The costs to be considered include the cost of learning, the cost of debugging, the costs of change, and the cost of running the application. Because different paradigms organize and factor programs in different ways, for a particular part of an application, the various costs for using a particular paradigm can vary across parts of a single application. By allowing the user to have a choice, the total cost can be lowered.

Leverage from use of a paradigm arises primarily from two sources. The first is through the power of elision; different paradigms differ substantially in what can be concisely stated. Significant appeal of a paradigm arises from what does not have to be stated. By eliminating redundant or orthogonal verbiage, the intent of the code can be more easily understood. This, for example, is an important virtue of the separation of logic from control in logic programming.

Accomodation of program changes is the second source of power in a programming paradigm. Program change is facilitated by a number of facilities in the environment; however, an important component is the set of language features. To the extent the commonly occurring changes leave invariant appropriate properties of the program, the programmer need will have less to think about, and will avoid some change-induced bugs.

Different paradigms allow different things to be stated concisely, and provide different invariants under change. One must ask of a programming environment:

What paradigms of programming are available to the user in the system as it stands?

This is separate from the question of the underlying implementation environment; that is, a paradigm could be implemented in Lisp or Prolog or machine language. The questions about implementation revolve about whether the implementation environment provides easy facilities for building the semantics of the paradigm, whether it can easily maintain the user illusion, and whether it allows easy mixing of multiple paradigms. This leads to a second question:

What support does the environment provide for embedding new paradigms?

Does the system allow new syntactic forms embedded within current structures? Can the user create efficient implementations? Is it easy to save and restore linguistic entities written in new sublanguages? All these are necessary, if not sufficient conditions for embedding.

A third question is:

How are the paradigms integrated into the environment?

One can easily imagine using Lisp to implement Prolog. However, one would like to see, for example, smooth transitions between Prolog and Lisp in both directions, and coordination of environmental tools such as debuggers, editors, etc. In what follows, we describe several different paradigms, and bring up questions which occur most naturally in trying to implement each paradigm.

2.1 Object Oriented Programming

Object oriented programming has proven to be a valuable way of thinking about programs. In this paradigm, objects combine state and behavior. In general, objects are grouped into classes, all of which have the same structure and the same behavior. Behavior is invoked from an object by sending it a

message. A message consists of a name for a behavior (often called a selector) and some other parameters. The response to a message is determined by the class of the object. Thus, if one is programming a simulation of some vehicles, this "message passing" allows separation of the implementation of functionality for each class of object. If one has a collection of different types of vehicular objects, changing how one of them responds to the *MoveForward* message does not effect the other implementations. Adding a new type of object which responds to the *MoveForward* message can be done independently by different people. From the users point of view, one wants to add and view behavior factored by the class of objects, rather than by operation (the usual factoring in function oriented and logic oriented systems).

In building a representation of objects within a system, if one is going to build many interacting objects, one is forced to ask about the implementation:

Can one efficiently represent objects with class specific behavior and dynamic state?

A second powerful idea of object oriented programming which should be supported well is the notion of specialization. One class should be describable as being like one or more others with some additions. Behavior should be inherited from the more general "super" classes, and it should be easy for the user to state this fact. The question is:

Can classes easily be specialized from one or more super classes?

As a simple example, suppose the user defined a MovableObject class, with structure that included an xPos and yPos, and simple behavior that included responses messages to Move, CurrentPosition, and Home (return to 0,0). Then every subclass of MovableObject would include that structure and that behavior.

General behaviors should be dividable at a fine enough grain so that subclasses need only specify a portion of a new behavior specific to the specialization. For example, suppose the Move method for MoveableObject was:

```
[Move (self newX newY)   ;arguments of the
method are the object, and new position
   (← self Erase)        ;Call the object specific Erase
   (←@ xPos newX)
   (←@ yPos newY)        ;Update the local information
   (← self Draw)         ;Call the object specific Draw]
```

Now suppose a specialization *Square* inherits this method. Square needs to specialize the Erase and Draw messages. In the invocation of Move which is inherited from MovableObject, it is the method for *Erase* found on Square that will need to be called. One must ask:

What granularity of inheritance is allowed for an object hierarchy?

Explicit message delegation as in for example Concurrent Prolog (Shapiro and Takeuchi 83) doesn't allow such fine grained specialization. Once a "message is passed" to the delegated object, the object being processed is the delegate. Thus specialized calls from the inherited method must be found on the delegated object.

2.2 Access-Oriented Programming

Access oriented programming facilitates separation of monitoring program events from the processes that can cause those events. In particular, the events of concern here are storage and access to states of objects. In procedure and logic oriented programming, this is achieved by providing an interface procedure (relation) which is always called to make the changes. Then changes and access can be monitored simply by changed the access procedure (relation). The mechanism of annotated values used in Loops provides affirmative answers to each of the following questions which make it superior to interface functions for achieving separation of concerns:

Can programmers monitor arbitrary values without previous programatic anticipation?

Is the mechanism invisible to programs not using it?

Is there low computational overhead when the mechanism is not in use?

Is the mechanism efficient when in use?

Is it independent of the number of instances of the mechanism being used?

Does the mechanism allow specialization, and self embedding?

The mechanism in Loops for achieving this effect is called an annotated value. There are two kinds of annotated values: *property annotation* and *active values*. *Active values* can associate, with any value, methods to be invoked when a data store or fetch is requested. Active values can be inserted in any

object value, and need not have been anticipated. The implementation for active values makes use of a special data type, *valueWrapper*, which is checked for on any storage or retrieval operation. This data type check is extremely fast (supported in the microcode), and hence adds little overhead when not in use.

The *valueWrapper* contains a Loops object which will be sent a Get or Put message depending on the access required. Since the *valueWrapper* is local to this value, the efficiency is independent of the number of other active values in use. Because the wrapped object is an ordinary Loops object, the user can use one of the standard class of active values for monitoring, debugging, maintaining simple constraints, and can specialize these for his own application in the same way he can specialize any class of objects.

Property annotations can associate with any value an optional labelled property list. Such property lists can be used to annotate the value with such useful but subsidiary quantities. For example, to augment a reasoning system to use certainties (or probabilities) one can store the certainty of particular values on annotations for that value without having to change the structure of the represnted object. If one were to use a TMS style of reasoner, it would be possible to store justifications for particular values as annotations, and be able to do dependency directed backtracking. Annotations are provide a place for documentation for human readability of data structures.

2.3 Logic Programming

Logic programming advocates have been split in whether one should consider such programming as knowledge representation or higher level programming. Prolog has been the flagship of Logic Programming. As such it has attempted to answer in the affirmative the important question:

Is there a clear declarative reading to "program" statements?

However the exigencies of making Prolog into an efficient programming language have led to the use of control mechanisms which effect the declarative reading. As indicated by Robinson (1983), the CUT is the GOTO of logic programming, with well documented effects on the declarative semantics of the program statements. Alternative approaches to Logic Programming which make use of different control primitives and search strategies, such as SProlog (Smolka83), TABLOG (Malachi & Manna83), and ESP (Chikayama83) suffer less from this problem.

Another question with an unfortunate answer in current Prolog implementations is the following?

Are the answers independent of the control flow in the logic?

The negative answer in Prolog comes up in a number of guises. Since Prolog is incomplete, it may fail to get an answer which is implicit in the logic of the statements. Another kind of problem occurs when trying to use a straightforward program as a generator. Consider the following definitions:

```
append(List, [], List).
append([F|Z], [F|X], Y) :- append(Z, X, Y). ;Z is the result of appending X to Y

sublist(Sublist, ContainingList) :-
        append(ContainingList, FirstPart, Y)
        append(FirstPart, X, Sublist).
```

Given the problem:

```
sublist([a b], X).
```

Prolog goes into an infinite loop without giving any answers. This is a result of the order of the literals (goals) in the definition of sublist, and the depth first generation order for lists for append. Simply reversing the order of the clauses won't help, because then the problem:

```
sublist(X, [a])
```

will loop forever (rather than fail) after it generates the first two sublists of [a].

Another question which arises from the theme of multiple paradigms is:

How easy is it to embed other paradigms in this one?

There are several parts to this problem, some of which are easier in logic programming, and some harder. For example, to translate statements in relational form to other Prolog statements (where the natural semantics is logic) is easy in Prolog. Unification, a natural for combining of pattern matching and structure decompostion, is a good starting point for a compiler. The harder part to achieve is the incorporation of special syntax, and more the inclusion of non-logical semantics, especially non local control structures. ESP

(Chayama83) is an example of a Prolog extension which incorporates features to make such extensions easy. Languages such as LM-Prolog (CarlssonKahn83) use a Lisp base to extend the Prolog syntax and semantics.

With the trend towards decreasing cost for hardware, it is becoming clear that the individual user will have access to multiple machines. This leads naturally to the question:

How are parallel computations expressed within the paradigm?

Much work has been done in Prolog on identifying sources of parallelism, and allowing users to take advantage of it. Conery [1981] identifies four primary kinds: *and* parallelism, for each of the subgoals of a goal, *or* parallelism, for alternative sets of subgoals for a single goal; *stream* parallelism, where pipelining of results can allow one process to take partial results from another; and *search* parallelism, where a database is to be searched, and the assertions are divided between processors. Shapiro (1983) has suggested using read only variables as a primary synchronization mechanism for such parallel activity. Object oriented parallelism has focused on *data flow* to objects which compute when enough arguments are available, and independent actor models, with queues of messages for synchronization. This clearly is an important direction for future work.

2.4 Rule Based Programming

Rule based programming is often very much like logic programming. However, at least two issues are usually developed in rule systems which have not been addressed as much in the logic programming paradigm: dealing with uncertainty, and explanation systems.

Many knowledge based systems must deal with uncertain information, and inferences which are judgemental rather than logical. Extensive work has been done in the context of rule based on combining the evidence from a number of sources. This is at the heart of the Mycin system for medical diagnosis, and its various offshoots. One must ask of any system:

How is evidential reasoning handled?

The second point which is most strongly seen in rule based systems such as Mycin is the development of extended facilities for providing user explanations. By keeping track of the rules used to determine the answer to questions, the system can provide the user with a justification which consists of the trail of rules which provided positive evidence. A problem in this form of explanation is that it does not usually show how negative evidence was used to rule out alternative conclusions. On must ask of any system:

How does the system explain its results to the user?

There is a the tension in Prolog between providing an efficient programming language (for example, to write append), and writing a higher level reasoning system. For the former, explanations in terms of storing program traces are inappropriate; for the latter, it is worth the cost. A compromise which has been tried is to build an intepreter in Prolog which does the recording, at the price of an extra level of interpretation.

3. The Environment

Knowledge programming is exploratory programming. The specification of the system is developed with its implementation. As Kowalski (1984) puts it,

> *".. a Prolog programmer who is always correcting errors ... is a bad programmer. But for a person using Prolog ... as a language for analyzing ... knowledge,...trial and error is unavoidable."*

The question then arises for every knowledge programming environment:

What tools are available for understanding, debugging and improving programs incrementally?

3.1 Errors, Tracing, Breaking, Monitoring

In trying to understand a program, it is often useful to run it at slow speed, tracing its steps, and taking them one at a time. In addition, when when a program error occurs, the program should halt and allow the user to explore the dynamic environment of the computation. In embedded paradigms, and in systems where the dynamic state is coded for efficient computation, this imposes an often difficult criterion of translation of the computational environment into the source language of the user.

What view of the program state is available to the user?

In Interlisp-D (Sanella83), facilities are provided so that views of data values can be filtered through macros provided by the user. Loops uses these to allow objects being processed to be in their source form. In the Prolog implementations I know, tracing and stepping are shown in source form, but inspecting the environment of a computation requires knowing internal representations, if it is available at all.

One can ask more generally what appropriate abstractions of a complicated system can be seen by the user. Graphs of potential call trees, dependency of modules, definitions of interfaces are all potential candidates. Supporting growth of systems that are inevitably too big for one person requires many of these views.

Another important facility for incremental debugging is the ability to make a change in the program source, back up the stack to the last call to the place where the error occurred, and to resume the computation form that point. The reason this is necessary is that in large systems, developing the state in which the failure took place may be long and expensive, and/or may have required extensive human intervention which may be tiresome to recreate.

Can exception handling allow special processing, and backing up of the process?

Recognition of exception conditions is often a way of making main line coding straightforward, and easy to understand. Lisp machine lisp (WeinrebMoon81) for example, has a complex error handling system using objects that allows recognition and specialization of code to deal with error situations.

3.2 Analysis and Performance Tuning

For real applications, one needs to tune performance. This implies multiple representations, space-time tradeoffs, and the ability to change implementations while holding the higher level specifications invariant. It also requires timing analysis tools easily accessible to the user.

Examples of the latter tools are the Mastescope and Spy tools written by Larry Masinter for Interlisp-D. Masterscope provides a full analysis of the cross-calling behavior of procedures, and their use of variables, local and free. It was easily extendible to include message sending by procedures to be stored as part of its database. A graphical interface allows easy access to the source code from the network of relations. Since it is integrated with the file system, it keeps track of changes to pieces of the system, and can update its data base when necessary.

Spy is a program which allows the user to see how much time is being spent in any part of his program. It provides a graphical view of the incremental and integrated time spent in any subroutine of the sytem during a particular computation. The cost of using the Spy is small, since it runs by sampling the computation rather than by computing and storing intervals. Comparable measurement tools need to be developed for Logic programming systems to allow the development of production quality systems.

Various techniques are available for improving the performance of systems. In Lisp, a standard technique is Macro expansion which eliminates function calls, and can specialize the code for the arguments. A similar but more general technique is suggested by Kahn in Prolog by specializing interpreters using Partial Evaluation. Smolka has suggested in that Prolog programs be marked for data flow (which are the input, outputs), and functionality (only once answer expected) to obtain specialized compilations with much greater efficiency. These techniques get to be more important as we develop a library of programs on which many people depend, and which are in the inner loops of computations.

3.3 User Interface tools

Systems are for people, and we should be able to stay in a single computational environment as we switch between all the computational tasks of our everyday life. This leads to the question:

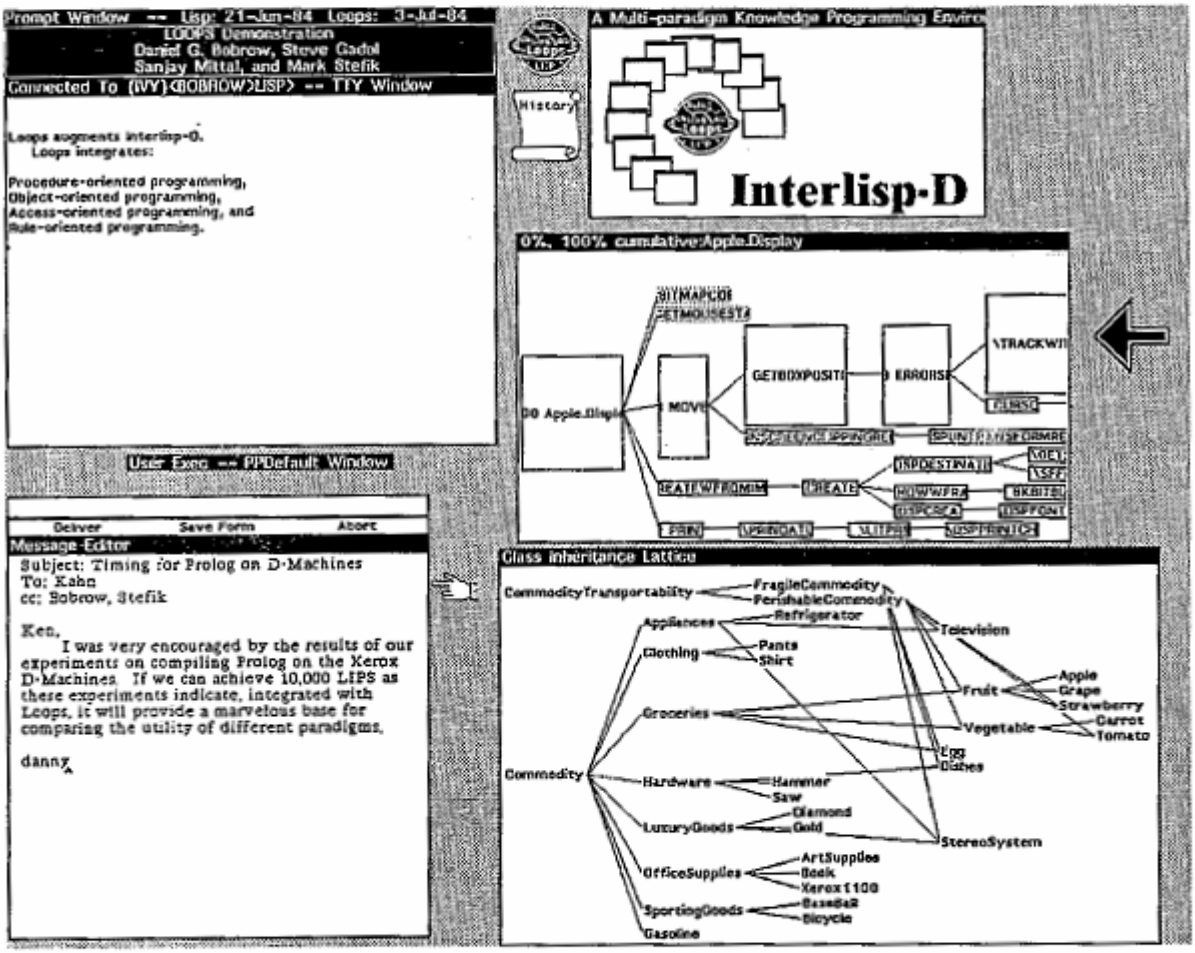
How rich is our computational environment?

People need to switch between tasks; hence multiple proceses are a necessary part of the environment. High quality displays are needed for high bandwidth communication. People communicate with each other, both on a rapid time scale (several times a day) and over a longer period to report their results. This implies the need for integrated message systems and editors for papers.

These facilities ought to be built in the same environment as the knowledge programming system, both to minimize switching time for the people, and to allow programmatic interaction with the facilities so that we can use the best tools to improve our own facilities. The figure below illustrates how a number of tools are available simultaneously in the Interlisp-D system.

4. Conclusions

The questions that have arisen in this document have been driven by the philosophy that people need to have a large set of tools for building the fifth generation systems that we have all set our sights on. Although it has criticized some of the current incarnations of Prolog and other systems, its real focus has been explore how computational environments should be enriched. The question is not which language or system is good and which bad, but rather what are the proper tools and paradigms for each problem, and how can we combine them to get maximum synergy.



This figure is taken from the screen of an Interlisp-D machine running Loops. In the lower left is a electronic mesage being sent. It was created using a text editor which is also available for production of formatted documents. To the right are two uses of graphical net browsers. The upper one shows a timing breakdown produced by the Spy system, where the size of the box is proportional to the proportion of time consumed by the particular function. The lower network shows the inheritance lattice for a Loops simulation of a Truckin world. It is an active network, allow users to access and change classes by pointing at the node in the display.

Acknowledgements: I wish to thank Mark Stefik, a close collaborator and friend, with whom many of these ideas were developed; Ken Kahn for teaching me much about Prolog and logic programming that is otherwise inaccessible to those outside the community; and John Seely Brown, the Intelligent Systems Laboratory and the Xerox Palo Alto Research Center for providing an environment which supports the exploration of ideas and systems.

Bibliography

Bobrow, D. G., Stefik, M. J. *The Loops Manual*, Intelligent Systems Laboratory, Xerox Corporation, 1983

Carlsson, M., Kahn, K. *The LM-Prolog Manual*, UPMAIL, Uppsala University, Uppsala, Sweden 1983

Chikayama, T. *ESP - Extended Self-contained PROLOG- as a Preliminary Kernel Language of Fifth Generation Computers*, New Generation Computing, Springer Verlag V. 1, No. 1 1983

Conery, J. S., Kibler, D. F. *Parallel interpretation of logic programs* Proc of the ACM Conference on Functional Programming Languages and Computer Architecture, October 1981

Goldberg, A. and Robson, D. *Smalltalk-80, the language and its implementation*, Addison Wesley, 1983

Kahn, K., Carlsson, M. *The Compilation of Prolog Programs without the Use of a Prolog Compiler*, FGCS '84 Tokyo 1984

Malachi, Y. Manna, Z. and Waldinger, R. *TABLOG: The Deductive-Tableau Programming Language*, ACM Symposium on Lisp and Functional Programming, 1984

Robinson, J. A., *Logic Programming -- Past Present and Future* New Generation Computing, Springer Verlag V. 1, No. 2 1983

Sanella, M. *Interlisp-D Reference Manual* Xerox Corporation 1983

Shapiro, E. and Takeuchi,, A. *Object Oriented Programming in Concurrent Prolog* New Generation Computing, Springer Verlag V. 1, No. 1 1983

Sheil, B. *Power Tools for Programmers*, Datamation 1983

Smolka, G. *Making Control and Data Flow in Logic Programs Explicit*, ACM Symposium on Lisp and Functional Programming, 1984

Stefik, M.J., Bobrow, D. G., Mittal, S., Conway, L. *Knowledge Programming in Loops, Report of an experimental Course*, AAAI Magazine Fall 1983

Weinreb, D. and Moon, D. *LISP Machine Manual*, MIT Cambridge Mass, (July 1981)

Yokoi, T. et al, *Logic Programming and a Dedicated High-Performance Personal Computer*, Fifth Generation Computer Systems (T. Moto-oka) North Holland Publishing Company 1982