UPMAIL Technical Report No. 15B

Unique Features of Lisp Machine Prolog

Kenneth M. Kahn
revised by Mats Carlsson

February 2 1985

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, Uppsala University
P.O. Box 2059
S-750 02 Uppsala Sweden
Domestic 018 155400 ext. 1860
International +46 18 111925
Telex 76024 univups s

The research reported herein was supported by the National Swedish Board for Technical Development (STU).

Keywords: Prolog, Logic Programming, Unification, Embedded Languages, Lisp, Partial Evaluation, Virtual Objects.

Table of Contents

1. An Overview of the Lisp Machine Prolog Project 1
2. Design Strategies 2
3. Syntax and the Representation of Terms 2
3.1 Handling of Cyclic Structures 4
4. Evaluable Predicates 4
4.1 Control Primitives 4
4.2 Collection Primitives 7
4.3 Database Primitives 8
4.4 Interface between Lisp and Prolog 9
4.5 Recursive Prologs 9
4.6 Mutable Arrays 10
5. Directions for Future Research 10
6. Appendix — Benchmarks 11
7. References 11

Abstract

The unique features of a Prolog implemented in ZetaLisp are presented. The language features variable arity of predicates, user definable objects, mutable arrays, unification with or without occur check or handling cyclic structures, multiple databases, user-controllable database indexing, efficient conditionals, declarative determinacy, virtual sets and bags, mechanisms for delaying goals, the ability to re-enter Prolog recursively, turtle graphics, several debugging aids, and various interfaces between Lisp and Prolog.

The implementation is fully integrated in the Lisp Machine programming environment and features a translator from Edinburgh Prolog programs, an interpreter, an optimizing compiler, and microcode support.

This report is intended to describe the aspects of LM-Prolog that differ from the Prolog implementations in wide spread use and assumes that the reader is familiar with Prolog.

A user manual [Carlsson and Kahn, 1983] and an implementation description [Carlsson, 1984b] are available.

1. An Overview of the Lisp Machine Prolog Project

LM-Prolog (Lisp Machine Prolog) was initially developed with one purpose in mind, namely to support a partial evaluator capable of partial evaluating the Prolog. A partial evaluator is a program which takes in programs and generates more efficient, less general, versions of them ([Haraldsson, 1977] [Emanuelson, 1980]). The theorem prover inside the Prolog interpreter can be specialized with respect to a particular database of assertions thereby compiling the Prolog to Lisp. In fact, LM-Prolog's compiler translates Prolog predicates to Lisp procedures, which are further compiled to machine code by the Lisp compiler.

The partial evaluator for Lisp programs is written in LM-Prolog and is described in [Kahn, 1982b].

That the implementation is to be partial evaluated constrains it to be as simple, modular, and free of side-effects as possible. In addition, in order to partial evaluate it most effectively it must use the Lisp stack for control. That the implementation should support a partial evaluator constrains it to be fast enough to run a large program on difficult problems. In addition, all the *evaluable predicates* need to be implemented in a clean way for the partial evaluator to partial evaluate their use.

Another aspect of this project is to develop a partial evaluator for Prolog (written in Prolog). This program transforms and specializes Prolog programs. The plan is that the output of this program will be feed into the partial evaluator for Lisp programs together with the Prolog interpreter to produce very high quality Lisp code which can be further compiled down to machine language.

Satisfying these constraints would be impossible without very rich computational resources. The Lisp Machine, a single-user, large address space computer, which supports a powerful modern Lisp dialect has been instrumental in this research [Moon *et al.* 1983]. Running a slow interpreter in another interpreter on a time-shared machine with a limited address space would be too painful to attempt.

An important aspect of this project is of course to have a production quality Prolog available on Lisp machines. A high-performance implementation of a logic programming language is for the first time available in such a powerful programming environment.

2. Design Strategies

LM-Prolog uses copying as the means for constructing complex terms, as this greatly simplifies the interfaces between Lisp and Prolog and otherwise seems advantageous over structure sharing on the present machine architecture. The database contains coded representations of program clauses. The codes are designed to be maximally suitable for the microprograms that are responsible for unifying with and constructing complex terms.

Unification contains an escape mechanism for user-defined objects, upon which a mechanism for delaying goals, lazy and eager collections, and mutable arrays are based. This design facilitates user experimentation and extensions with new data types. Unification capitalizes on *invisible pointers* for variable to variable links and for unifying cyclic structures.

Continuation passing is used as the means for having Lisp procedures compute relations rather than functions. A predicate is translated to a procedure taking one extra argument — a continuation corresponding to the remainder of the current computation. Similarly, the interpreter accepts an extra argument. See [Carlsson, 1984a] for details.

Interpreted and compiled predicates may call each other freely.

3. Syntax and the Representation of Terms

The syntax of LM-Prolog resembles the syntax of Lisp rather than logic or other Prologs. A predicate is defined as follows:

(define-predicate *predicate-name options*
clause$_1$
...
clause$_n$)

Options are optional and discussed later. A clause is a list of predications. A predication is a cons of a symbol or variable and a list of arguments. A clause is interpreted as its *head* (the first element) is implied by its *body* (the rest).

Variables are distinguished from symbols by beginning with '?'. For example, the definition of append in LM-Prolog is:

```
(define-predicate append
  ((append ())) ;;zero lists concatenate to the empty list
  ((append ?total () . ?back) ;;an empty list can be simplified away
   (append ?total . ?back))
  ((append (?first . ?rest-of-total) (?first . ?rest-of-front) . ?back)
   ;;a non-empty list contributes one element to the concatenation
   (append ?rest-of-total ?rest-of-front . ?back)))
```

LM-Prolog allows only one functor symbol '.', i.e. Lisp's *cons*. This may seem a strong restriction, but it is, in fact, a generalization of the concept of terms. Predicates can accept an arbitrary number of arguments as a consequence. This convention facilitates the application of a predicate to a list. For example, append may be used with in many different ways:

(append ?list ?x ?difference)
: subtracts the list ?x from the beginning of ?list to obtain ?difference.

(append ?list ?difference ?x)
: subtracts the list ?x from the end of ?list to obtain ?difference.

(append ?list ?x ?y)
: generates pairs of lists ?x ?y which append to ?list.

(append ?list ? (?term) ?)
: decides whether ?term is a member of ?list.

(append ?list ? (?x) ?)
: generates elements of ?list called ?x.

(append ?list ? (?x))
: finds ?x which is the last element of ?list.

(append ?list ? (?term-1 ?term-2) ?)
: asks if ?term-1 and ?term-2 are successive elements of ?list.

(append ?a-list ? ((?key ?value)) ?)
: asks if ?value is associated with ?key in ?a-list.

(append ?list ? ?sub ?)
: asks if ?sub is a sub-list of ?list.

(append ?list ? (?term) ? (?term) ?)
: asks if ?term occurs twice in ?list.

(and (append ?list ?before (?term) ?after) (append ?remaining ?before ?after))
: removes ?term from ?list to obtain ?remaining.

(append ?list ?a ?b)
: where ?list, ?a, and ?b are bound will answer the question of whether ?a and ?b append to ?list.

And so on.

In order to exploit the variable-arity of LM-Prolog predicates the convention in all system provided predicates is, contrary to other Prologs, that typically the first argument is the output and the rest of the arguments the input.

Another consequence of the use of the '.' functor symbol is LM-Prolog's second-order "predicate call" feature. For example, a valid way of appending two lists is (?predicate ?x ?list-1 ?list-2) provided ?predicate was previously unified with append.

3.1 Handling of Cyclic Structures

The unification algorithm computes most general unifiers, i.e. minimal substitutions that make its arguments equal. When the only bindings which could make them equal describe two cyclic structures, we have a problem. We can have the unification procedure perform an *occur check* so that it detects such situations and fails. Sometimes this is the desired behavior. Other times the cyclic structure is desired. Yet other times, the programmer is confident that they will not appear and does not want the system to waste time looking for circularities. LM-Prolog allows the user to control whether the occur check is done or not, or whether cyclic structures should be anticipated. In the latter case, LM-Prolog succeeds in printing, instantiating (when making bags and sets) and unifying such structures. The user specifies whether circularity should be prevented, handled, or ignored. It is also possible to bind the mode over the scope of some computation.

4. Evaluable Predicates

These correspond to the primitive functions of other languages. The current list of them are

- Control primitives for conditionals, delaying goals, guaranteeing actions upon failure, cutting out non-determinism, sound and unsound versions of negation by failure.
- Collection primitives for generating lazy and eager sets and bags of items satisfying given conditions.
- Database primitives for asserting, retracting, and assuming clauses, defining and tracing predicates, and naming databases.
- An interface to Lisp which is used by the system to define i/o, arithmetic, graphics, and the like.
- Metering facilities to time the execution of goals.
- Debugging tools for tracing and stepping, like in Edinburgh Prolog.

4.1 Control Primitives

Negation is a problem in logic programming, because a logic program cannot contain rules for deducing negative information. However, it is often sensible to regard the predicate definitions of a logic programs as **complete** definitions, i.e. to regard them as giving not only sufficient but also necessary conditions for the predicates to be true. If one is prepared to do so, the *negation as failure* [Clark, 1978] rule becomes valid. This rule consists in deducing $\neg P$ from a failure to prove P, and is the usual treatment of negation in Prolog. In LM-Prolog, this rule exists as the evaluable predicate cannot-prove.

Another problem with negation is that of free variables. A predication $P(X)$ with a free variable X is in Prolog procedurally understood as a goal "find an X such that $P(X)$ is true". However, the negation as failure rule is not faithful to the intended use of free variables. A negative predication $\neg P(X)$ procedurally becomes a goal "there is no X such that $P(X)$ is true". Thus, negated predications containing free variables violate the semantics of the language and can cause bizarre results. Cannot-prove does not check for such situations. One solution for this problem is to delay the predication in hope that its free variables will eventually become bound, and execute it only then. In LM-Prolog, this rule exists as the evaluable predicate not, and is recommended (except for efficiency reasons) over cannot-prove.

The conditional of LM-Prolog is called Cases and resembles Lisp's cond. It is equivalent to the disjunction of the conjunction of the elements of a clause and the negation of the first element of each of the previous clauses. For example, (cases (a1 a2 a3) (a4 a5) (a6)) is logically equivalent to (or (and a1 a2 a3) (and (cannot-prove a1) a4 a5) (and (cannot-prove a1) (cannot-prove a4) a6)). This predicate cannot be implemented within Prolog itself without either unnecessary re-computation when tests fail or restricting the test to its first solution. (recomputing a1 and a4 in the example above).

IC-Prolog [Clark *et al.* 1982] has a conditional similar to LM-Prolog's which can be defined in terms of Cases as follows:

```
(define-predicate if
  ((if ?test ?then ?else)
   (cases (?test ?then) (?else))))
```

The controversial cut or slash of Prolog is also provided, it is implemented as a predicate called cut. There is another means of cutting out unwanted backtracking from LM-Prolog programs, which is generally recommended over cut: Predicates can be declared to be deterministic, i.e. to never backtrack to find more solutions. This is done by adding a declaration to the options part of the predicate.

LM-Prolog provides a primitive for guaranteeing that something is executed upon backtracking: (unwind-protect ?goal ?cleanup...) is equivalent to just ?goal but also guarantees that ?cleanup... is executed upon backtracking.

LM-Prolog provides primitives for delaying computations. One is called constrain and is similar to geler of Prolog II [Colmerauer 1982]. A predication (constrain ?x (p ?x)) succeeds immediately but binds ?x to a delayed computation of (p ?x), so that if ?x is later unified with a non-variable, the delayed goal is executed. A limitation of the implementation is the fact that the delayed goal becomes deterministic. Another primitive is freeze, which delays the execution of its arguments until it is ground. For example, Not is defined as

```
(define-predicate not
  ((not ?predication)
   (freeze (cannot-prove ?predication))))
```

To illustrate constrain with a classical example, we implement the *Sieve of Eratosthenes* for generating the list of prime numbers (compare with [Hansson *et al.* 1982] and [Kahn, 1982a]). The algorithm repeatedly takes the first element of a list, adds it to the list of primes, and deletes all multiples of it from the original list. The list is initially the integers beginning with 2.

```
(define-predicate integers-beginning
  ((integers-beginning ?n (?n . ?rest))
   (sum ?n+1 ?n 1)
   (constrain ?rest (integers-beginning ?n+1 ?rest))))

(define-predicate primes
  ((primes ?primes)
   (integers-beginning 2 ?integers)
   (sift ?primes ?integers)))

(define-predicate sift
  ((sift (?first . ?rest-sifted) (?first . ?rest-integers))
   (no-multiples ?rest-left ?first ?rest-integers)
   (constrain ?rest-sifted (sift ?rest-sifted ?rest-left))))

(define-predicate no-multiples
  ((no-multiples ?filtered ?n (?int0 . ?int))
   (cases ((lisp-predicate (zerop ( '?int0 '?n)))
           (no-multiples ?filtered ?n ?int))
          ((= ?filtered (?int0 . ?rest-filtered))
           (constrain ?rest-filtered (no-multiples ?rest-filtered ?n ?int))))))
```

Running (primes ?p) binds ?p to (2 . <delayed object # 77>) which in all ways behaves like the list of primes. (If one maps print down ?p the primes will be printed out one by one until the machine is stopped.)

Clearly, the constrain facility has lots of applications in logic programming, e.g. to implement sound versions of inequality and negation, to implement I/O streams, to write "generate and test" problems in a clear yet efficient way, to solve equations, and to compute with infinite lists.

An output stream that prints a list of characters on an output device as the list becomes instantiated is easily programmed:

```
(define-predicate output-stream
  ((output-stream () ?)
   (output-stream (?elt . ?rest) ?device)
   (lisp-command (send '?device ':tyo '?elt))
   (constrain ?rest (output-stream ?rest ?device))))
;;Initial goal:
(and (constrain ?list (output-stream ?list ?device) (compute ?list))
```

A more comprehensive stream interface exists between LM-Prolog and a Turtle graphics program [Lieberman, 1980]. This interface also insures that drawn parts are undrawn, should the program backtrack.

4.2 Collection Primitives

Sets and bags in LM-Prolog are made using the predicates set-of and bag-of. For example, given the predicate grandparent we can construct the list of grandchildren of Cele as (set-of ?grandchildren ?gc (grandparent cele ?gc)). A unique feature of LM-Prolog is the lazy versions of sets and bags. Lazy sets and bags create lists, whose elements are computed only to the extent necessary. This is very useful when the sets involved are large and only some aspect of them is of interest. For example, in many Prolog solutions to the problem of Mr. S and Mr. P [Warren 1981] there is code which uses Edinburgh Prolog's setof predicate to determine whether there is exactly one solution. Prolog creates the set of solutions and then unifies it with a term describing the set. In LM-Prolog, the set is only created to the extent to which it is "looked at" in the process of unification. In the case where one desires to know if the solution is unique, LM-Prolog fails after the second solution is found. In the case where one desires to know if there are more than one solution, LM-Prolog succeeds after finding the second one. The difference in efficiency when using lazy sets is greatest, of course, if the sets are infinite. There is also an experimental version of bags and sets which are "eager" and are computed in parallel processes. See [Kahn, 1983] for a detailed discussion of lazy and eager collections.

The following is a version of the *Sieve of Eratosthenes* implemented with lazy bags:

```
(define-predicate primes
  ((primes ?primes)
   (lazy-bag-of ?primes ?prime (prime ?prime))))

(define-predicate prime
  ((prime ?prime)
   (lazy-bag-of ?ints ?int (integer-from ?int 2))
   (prime ?prime ?ints))
  ((prime ?prime (?prime . ?)))
  ((prime ?prime (?seed . ?rest))
   (lazy-bag-of ?sifted ?not-multiple (not-multiple ?not-multiple ?seed ?rest))
   (prime ?prime ?sifted)))
```

```
(define-predicate integer-from
  ((integer-from ?i ?i))
  ((integer-from ?i+ ?i)
   (sum ?i+1 ?i 1)
   (integer-from ?i+ ?i+1)))

(define-predicate not-multiple
  ((not-multiple ?number ?seed (?number . ?))
   (quotient ?q ?number ?seed) ;;fixnum arithmetics
   (cannot-prove (product ?number ?q ?seed)))
  ((not-multiple ?number ?seed (? . ?numbers))
   (not-multiple ?number ?seed ?numbers)))
```

4.3 Database Primitives

There are two types of Prolog predicates that can be defined. The default is :static which means that the list of clauses in the body of the define-predicate are all that exist or ever will exist. If there is an attempt to add or delete any new clauses, an error is signaled. Various optimizations apply to static predicates and they are the default. Another type is :dynamic which allows subsequent addition or deletion of clauses. The predicate assert adds a new clause to a dynamic predicate. The predicate retract deletes a clause from a dynamic predicate.

Since this is a side-effect operation without a logical meaning, it is good programming practice to use it at top-level and in files and to minimize its use in programs.

A more acceptable way for programs to change the database is by using the Assume predicate. Assume inserts a clause in the database but upon backtracking restores the original predicate definition. This guarantees that the predicate is restored even in the case of abnormal return due to the use of control primitives or if the user aborts the computation.

The clauses in a database are automatically indexed by their predicate name. The options list of define-predicate can be used to specify additional indexing to speed retrieval. The system creates and uses hash-tables the keep access time down to a small constant regardless of the number of clauses associated with a predicate. The option :indexing-patterns is followed by a list of patterns. Each pattern should have exactly one symbol beginning with '+' in them. IC-Prolog has a similar indexing facility, but it cannot index on parts of the predicate's arguments.

For example, consider the following definition of father:

```
(define-predicate father
  (:options (:type :dynamic)
            (:indexing-patterns (father ?father ?) (father ? ?child)))
  ((father Jack Ken))
  ((father Jack Karen))
  ((father Isa Jack)))
```

This declares that father is a predicate which can be extended, and that indexing should be done both on father and child.

LM-Prolog supports multiple worlds. A world is a list of predicate definitions. The user declares to the system the *universe*, a list of worlds, in which computation should proceed. These worlds are searched linearly until a predicate definition is found. The implementation has a cache mechanism for avoiding this search in most situations. Predicates are available for *binding* the universe over some computation. The trace package, for example, traces a predicate by defining a predicate of the same name in the traced-predicates world which prints some message, then runs the original predicate, then prints some more messages.

This is analogous to the advice facility in some modern Lisps. Worlds are an ideal vehicle for hypothetical reasoning and structuring a complex knowledge base. ZetaLisp's *packages* or namespaces are inherited by LM-Prolog and provide good support for more static kinds of modularity, like preventing name clashes and accidental access from one program module into another.

4.4 Interface between Lisp and Prolog

For almost all purposes one of the three LM-Prolog predicates Lisp-value, Lisp-predicate or Lisp-command are adequate. Lisp-value is a two place predicate where the first argument is unified with the result of evaluating the second argument. Lisp-predicate is a one place predicate and it fails only if its argument evaluates to nil. Lisp-command is a one place predicate which always succeeds. The argument is executed.

In the other direction, the following Lisp function executes a Prolog query: (query-once *predication* ':lisp-term *pattern*). The *pattern* defaults to nil. If successful it will multiple value return an instantiation of *pattern* and t. If it fails it returns nil. A more general interface exists which sets up a stream of solutions. It is called make-query.

4.5 Recursive Prologs

LM-Prolog can be *entered recursively.* After printing out the results of the first answer to the user's goal, the user can either ask for more possibilities (by hitting the hand-down button), type in a new problem, or accept the current environment (by hitting the hand-up button). If the later option is taken, then Prolog is run in the environment of the last results shown. This provides essentially global variables during interactions with the system. These global variables cause no semantic difficulties for the language because they exist only in the user interface and cannot be used within programs.

This may be convenient at top-level as the following sample dialog illustrates:

```
(= ?i (integers-beginning ?integers 1))
OK
?i = (integers-beginning ?integers 1)
```
;;Hitting the hand-up *button indicates acceptance.*

```
;;The system displays "Prolog level 1".
?1 ;;meta-logical variable
OK
?integers = (1 .  <delayed object # 23>)
;; hand-up again — The system displays "Prolog level 2".
(= ?integers (?  ?  ?three .  ?rest))
OK
?three = 3
?rest = <delayed object # 25>
;; control-hand-down and we are back to level 1.
;; hand-down indicates that more possibilities are desired.
No more answers.
```

It is also useful to run Prolog recursively after an error or break to inspect the environment, add clauses, trace predicates, etc..

4.6 Mutable Arrays

LM-Prolog provides a completely logical interface to the arrays of ZetaLisp. A mutable array is, just like a ZetaLisp array, a (possibly very large) data structure with direct access to its elements. However, mutable arrays are logical objects and so are not amenable to destructive updates. LM-Prolog instead provides updating predicates which are not destructive, but creates a new virtual array while keeping the old. As much storage as possible is shared between the old and the new virtual array, in particular they both point to the same ZetaLisp array which normally reflects the new virtual array. The old virtual array normally keeps a record of the difference between it and the new virtual array.

The implementation is optimized so that in the best case array references and updates are nearly as efficient as using them directly in ZetaLisp. More details on mutable arrays can be found in [Eriksson and Rayner, 1984].

5. Directions for Future Research

One avenue of future research is to partial evaluate the Prolog interpreter with respect to the partial evaluator in Prolog to produce a Lisp version of the partial evaluator. This program can then be applied to itself and the Prolog interpreter to produce a Prolog compiler (i.e. a Lisp program for compiling Prolog programs into Lisp). This is described further in [Kahn, 1982b].

Uniform [Kahn, 1981] and Intermission [Kahn, 1982a] are logic programming languages which have various abilities lacking in Prolog. A second generation Intermission is being contemplated where every actor is represented by a Prolog database. There is some hope that an implementation of them in Prolog could be both compiled down to Lisp and compilers for programs in those languages could be generated. One difficulty expected in implementing Uniform in Prolog is the need for controlling search. A more complete version of Intermission with concurrency and explicit continuations would also demand more control over Prolog's search.

6. Appendix — Benchmarks

Performance benchmarks for Prolog were published in [Warren 1977]. The programs are *reverse-30*, a naive $O(n^2)$ reverse of a list 30 long; *qsort-50*, a quicksort of a list 50 long; four differentiations; a palindrome of a list 25 long; and a database query similar to the one described in section 2.3. The table below shows the timings in milliseconds for compiled and interpreted predicates on a CADR. (LM-C and LM-I, respectively), and for DECsystem-10 Prolog on a KI10 (10-C and 10-I). The LM-Prolog times are followed in parentheses by the timings without micro-code support. Note that the hardware of a KI10 is roughly twice the speed of a CADR Lisp machine.

The first benchmark suggest a LIPS rate of around 10000 for compiled code and 1800 for interpreted.

	LM-C	LM-I	10-C	10-I	
reverse-30	48(134)	274(1170)	53.7	1160	*;;498 inferences*
qsort-50	60(197)	671(2380)	75.0	1344	
d-times-10	7(16)	35(110)	3.0	76.2	
d-divide-10	8(18)	36(126)	2.9	84.4	
d-log-10	5(12)	22(77)	1.9	49.2	
d-op-10	6(12)	28(83)	2.2	63.7	
palin-25	42(167)	305(1250)	40.2	602	
query	422(1746)	1990(5858)	185.0	8888	

7. References

[Carlsson and Kahn, 1983] Carlsson M., Kahn K.M., "LM-Prolog User Manual", UPMAIL, Uppsala University, Technical Report No. 24.

[Carlsson 1984a] Carlsson M., "On Implementing Prolog in Functional Programming", UPMAIL Technical Report No. 5B, Computing Science Department, Uppsala University, and in *Proc. 1984 International Symposium on Logic Programming*, Atlantic City NJ, and in *New Generation Computing*, vol. 2 no. 4 pp. 347–360, Ohmsha, Ltd. and Springer-Verlag, Tokyo.

[Carlsson 1984b] Carlsson M., "LM-Prolog — the language and its implementation", Ph. L. thesis, UPMAIL Technical Report No. 30, Computing Science Department, Uppsala University.

[Clark, 1978] Clark K.L., "Negation as failure", in *Logic and Databases*, (eds. Gallaire H., Minker J.), Plenum Press, New York, pp. 293–322.

[Clark *et al.*, 1982] Clark K.L., McCabe F.G., Gregory S., "IC-Prolog Language Features", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Colmerauer, 1982] Colmerauer A., "PROLOG II Manuel de Réference et Modèle Théorique", *Proc. Prolog Programming Environments Workshop*, Linköping University.

[Emanuelson, 1980] Emanuelson, P., "Performance enhancement in a well-structured pattern matcher through partial evaluation", Linköping Studies in Science and Technology Dissertations, No 55, Software Systems Research Center, Linköping University.

[Eriksson and Rayner, 1984] Eriksson L.-H., Rayner M., "Incorporating Mutable Arrays Into Logic Programming", *Proc. Second International Logic Programming Conference*, Uppsala.

[Hansson *et al.* 1982] Hansson Å., Haridi S., and Tärnlund S.-Å., "Properties of a Logic Programming Language", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Haraldsson, 1977] Haraldsson, A., "A Program Manipulation System based on Partial Evaluation", Linköping Studies in Science and Technology Dissertations, No 14, Department of Mathematics, Linköping University.

[Kahn, 1981] Kahn, K. , "Uniform — A Language based upon Unification which unifies (much of) Lisp, Prolog, and Act 1", *Proc. IJCAI-81*, Vancouver BC.

[Kahn, 1982a] Kahn, K., "Intermission — Actors in Prolog", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Kahn 1982b] Kahn, K., "A Partial Evaluator of Lisp written in Prolog", *Proc. First International Logic Programming Conference*, Marseille.

[Kahn, 1983] Kahn, K., "A Primitive for the Control of Logic Programs", UPMAIL Technical Report No. 16, Computing Science Department, Uppsala University, and in *Proc. 1984 International Symposium on Logic Programming*, Atlantic City NJ.

[Lieberman, 1980] Lieberman H., "Logo Turtle Graphics for the Lisp Machine", MIT AI Laboratory Working Paper 214, May 1981.

[Moon *et al.* 1983] Moon D., Stallman R. M., Weinreb D., "Lisp Machine Manual", MIT AI Laboratory, Cambridge MA.

[Warren, 1977] Warren, D., "Implementing Prolog — compiling predicate logic programs", Department of Artificial Intelligence, University of Edinburgh, D.A.I. Research Report No. 39.

[Warren, 1981] Warren, D., "The 'S-P Problem' Revisted", Logic Programming Newsletter 2, Universidade Nova de Lisboa.