Use of a Forth-Based Prolog for Real-Time Expert Systems

II. A Full Prolog Interpreter Embedded in Forth

L. L. Odette

Applied Expert Systems, Inc.
5 Cambridge Center
Cambridge, MA 02142

W. H. Paloski

KRUG International
Technology Life Sciences Division
17625 El Camino Real, Suite 311
Houston, TX 77058

Abstract

In this article we outline the design of a Prolog interpreter embedded in Forth. The interpreter is the basis of the expert system component of an astronaut interface for a series of Spacelab experiments. The expert system is described in Part I of this article [PAL87]. Here we describe our approach to the representational issues in designing the programming machinery needed to interpret Prolog programs: (1) the internal representation of Prolog objects and (2) the representation of the state of a Prolog computation. We also describe the Forth-Prolog interface we use to support the mixed language programming that is necessary to handle the real-time data acquisition and control tasks involved in the application.

Our goal is to combine the advantages of Forth for real-time programming and the advantages of Prolog for symbolic reasoning. To take advantage of the large body of Prolog code we have developed for previous applications, we implemented the "core" Prolog system described in [CLO81] that is compatible with the widely available implementations.

Introduction

The text that follows briefly describes the implementation of the Prolog interpreter used in our application [PAL87]. The interpreter is fully Clocksin and Mellish compatible, using the standard Edinburgh syntax and providing the majority of the built-in predicates described in [CLO81] (some file I/O predicates are not implemented); however, it's a "tiny" Prolog in that it can fit in the 64K of the small model Forth. It is particularly suitable for Prolog applications that can leverage off the underlying Forth system, such as the knowledge-based control system described in [PAL87]. The full source code for the interpreter (about 100 screens) is available from the Forth Interest Group as volume 5 of the Forth Model Library.

The intent here is to cover briefly the main issues involved in implementing a Prolog interpreter in the context of one particular implementation in Forth. For this reason familiarity with Prolog is assumed. The first section describes the Forth data structures used to represent Prolog terms. The next three sections address memory allocation, how variables are bound and how Prolog procedures are invoked. The fourth section describes the implementation of built-in predicates and the interface between Prolog and Forth. The final section is a list of the built-in predicates that have been implemented.

Representation of Terms

Syntactically, there are two types of objects in Prolog—simple and complex objects. Constants and variables have no syntactic structure and are examples of simple (unstructured) objects. On the other hand, lists and predications have structure, and these are complex objects. Semantically, constants and variables are quite different from each other, and they need to have different internal representations. Moreover, for the sake of efficiency, internal representations of Prolog objects may incorporate other type distinctions (e.g., names and numbers may have separate internal representations even though they are both of "constant" type).

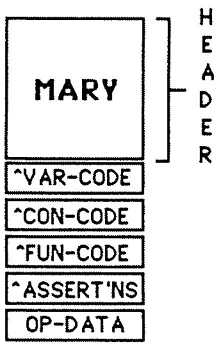
The Prolog interpreter needs to be able to identify the type of a Prolog object. Common schemes for typing include using separate memory areas for different object types or incorporating a type (tag) field into an object pointer. The former is probably not a good choice for a straightforward Forth implementation where it is natural to have the names of objects and the objects themselves intermingle in the dictionary. The latter was not compatible with the use of 16-bit pointers in this implementation. Instead, the typing scheme used here has the interpreter get the type of an object not from the value of the pointer or the pointer itself, but from the object pointed to. The following text describes this scheme in detail.

Primitive Terms: Constants, Variables and Numbers

Constants and variables have name and link fields combined to form a header just like any Forth word, and these are the only Prolog objects with name fields (referred to as named objects). Immediately following the header are an additional 5 fields comprising 3 vectors and 2 data fields. In order of increasing memory address the fields are:

Name	**Contents**	**Use (object pointer/goal)**
variable-code field	0	object = variable/(not used).
constant-code field	The cfa of the Forth word `RESOLVE.SINGLE`.	object = constant/ call function (no args).
function-code field	The cfa of the Forth word `RESOLVE.FUN`.	object = functor/ call function.
assertion field	A pointer to the list of assertions for this functor.	
functor data field	Functor data on precedence, position, associativity.	

For example, the representation of the constant term **mary** looks like:

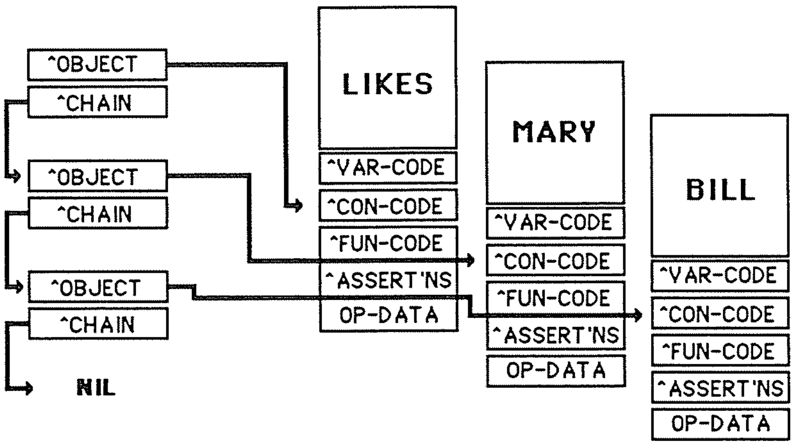


The vector fields of a named object have a dual use: they are used to transfer control at run time when a named object is to be executed as a goal or is the main functor of a goal, and they also are used by the unifier to identify the type of an object. Typing works as follows. If a pointer to a named object points to the variable-code field, then the object is treated as a logical variable. If the pointer points to the constant-code field, the object is treated as a constant. If the first element of a complex term points to the function-code field of some named object, then the complex term is interpreted as a function (predication or procedure) whose functor is the first element and whose arguments are the remaining elements of the complex term.

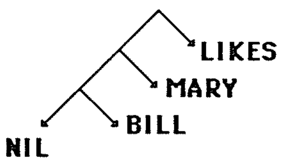
Numbers are the one exception to the typing scheme. The fact that an object is a number is determined either directly from the pointer to it or from the memory location the pointer references. In the first case, if an object pointer points to an address less than 256, then the pointer is interpreted as the object, that is, a number. This approach allows a more compact representation of small numbers at the expense of run-time checking. In the second case, if the first memory word of the object points to the next memory word, then the contents of the next word is interpreted as the number.

Complex Terms

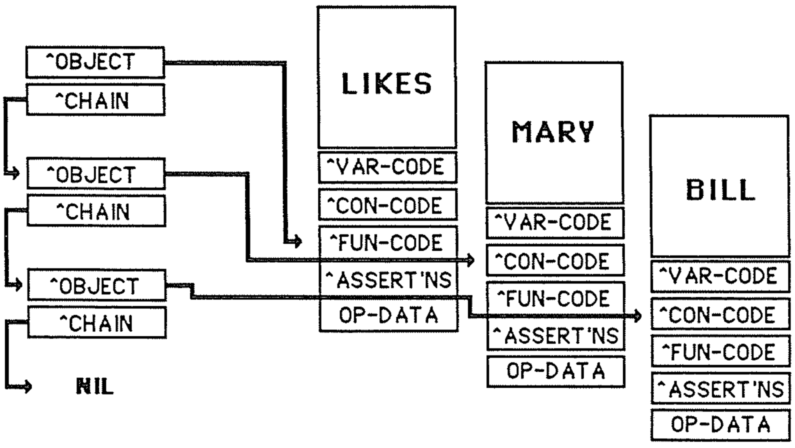
Complex terms are represented as chains of word (memory location) pairs. The first word of a pair points to an object; the second word of a pair points to the rest of the chain (for LISPers: all complex terms are lists—the word pairs constitute a CONS cell). A chain is terminated when the second word of a pair points to an object that is not an object chain. For example, lists are terminated with a pointer to the special object **NIL**, which is both a constant and the representation of the empty list. The list of terms **[likes,marry,bill]** is represented internally as:



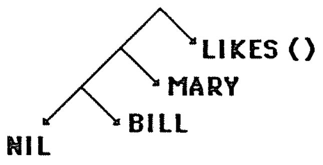
A more concise representation of the structure of complex terms is a tree diagram whose nodes represent word pairs. Each right subtree of a node is a representation of the object pointed to by the first word of the pair. Each left subtree of a node is a representation of the object pointed to by the second word of the pair (i.e., the rest of the chain). In this notation, the tree representation of the list **[likes,mary,bill]** is:



More general complex terms such as predicates (logical functions) are represented as chains in which the first object pointer points to the function-code field of a named object. Thus, the predicate **likes(mary,bill)** is represented internally as:



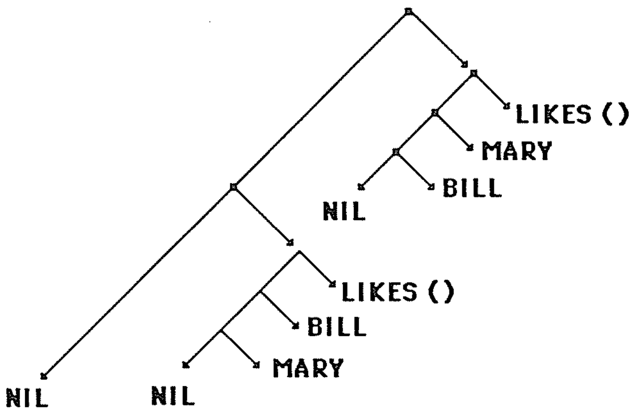
The tree representation of the predicate **likes(mary,bill)** is:



where the pair of parentheses denotes that the object pointer points to the function-code field of the named object.

Clauses

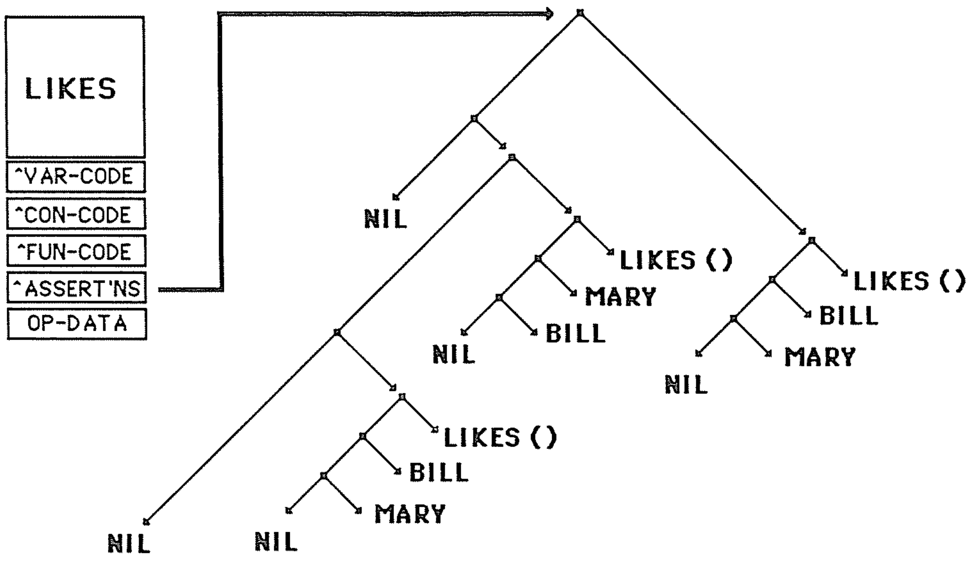
First some terminology: for the clause **A :- B,C,D.**, **A** is the head of the clause, **:-** is the neck, and **B,C,D** is the body of the clause. Clauses are represented internally as lists of terms, either simple or complex, where the first term in the list denotes the head of the clause. The remainder of the list is the body of the clause. Thus, the clause **likes(mary,bill) :- likes(bill,mary).** may be represented by the tree:



The main functor of a clause is the functor (function name) of the head of the clause. A list of all clauses with the same main functor is an assertions list (e.g., a list of all the clauses whose functor is **likes**). The assertions field of the main functor points to this list. Thus (using a mixed representation), the list of clauses

[(likes(bill,mary)), (likes(mary,bill) :- likes(bill,mary))]

is stored as:



Memory Usage

This implementation intermingles the object-name and program spaces in the Forth dictionary and provides for no garbage collection of either names or program. A heap might be used to handle de-allocation of program space following the retraction of a clause, although this procedure would need to be handled with care because the retracted clause may still have a reference—as an untried alternative that may be required to resatisfy an earlier goal.

In addition to the name and program spaces and the space allocated to the usual Forth return and data stacks, there are a stack that holds variable bindings (binding stack) and a separate stack for control information (goal stack). These stacks are made as large as possible because this is where the action is in a Prolog computation. In the version used in our application [PAL87] 32K bytes are allocated for each stack.

The goal stack is used for saving goal frames during a computation (it corresponds to the return stack in Forth). Each goal frame has 6 fields (2 bytes each) that will be described later. One of these fields points into the binding stack and indicates the top of the binding stack relative to that goal frame (i.e., the set of bindings in effect when that goal frame was entered). In this way variable bindings can be redone, if backtracking is necessary, simply by resetting the binding stack pointer. A new goal frame is allocated on successful resolution of a goal with the head of an assertion by advancing the goal stack pointer, copying the contents of the last goal frame and updating the fields.

Variable Bindings

Variables are bound as a result of unification during the resolution step. Each use of a variable name must be associated somehow with the call to the procedure that the variable appears in. We use goal frame indices for this purpose. Prior to unification, a pointer to the goal is associated with the value of the current index (INDEX.C) and a pointer to the head of the first clause in the assertions list is associated with the value of the next index (INDEX.N).

During unification, all substructures of the goal and all substructures of the clause head are associated with the indices of their parent structures. Saving a variable binding consists of saving both the pointers and indices of the variable and the object it is bound to. The binding stack is where these variable bindings are stored, and each element of the stack is thus an 8-byte data structure (**^var var.index ^obj obj.index**).

Variables are de-referenced by the Forth word @BINDING, which just searches the binding stack for a given variable name-index pair and recurses if the variable is bound to yet another variable.

Control

A Prolog computation needs to keep track of both the clause it is executing and any alternative clauses (in case backtracking is necessary), of which subgoal to execute next on a successful exit from the current subgoal, and of all variable bindings in effect when the clause was called. This information constitutes the state of the computation. We represent the state of a computation by a 6-word structure (called a goal frame). Each such state corresponds to a goal in the computation. The fields of the goal frame are:

Name	**Contents**
INDEX.N	next index
<ENV>	environment pointer
ASSERTIONS	assertions list pointer
GOALS	goal list pointer
INDEX.C	current index
REST.GOALS	indirect goal list pointer

The indices INDEX.N and INDEX.C (next index and current index respectively) are used in associating variables with the frame they were bound in, as described in the previous section on variable bindings. The environment pointer <ENV> points into the binding stack and indicates the binding environment on entry into the frame so the environment can be restored on backtracking. The GOALS pointer points to the goals list, a subset of the subgoals of the parent goal frame of the current goal frame. The GOALS list is used to determine the next goal if the current one is successful. The REST.GOALS pointer points into the goal frame of the parent; specifically, it points at the GOALS field of the parent frame. REST.GOALS is used to determine the next goal on exhausting the GOALS list in the current frame. The ASSERTIONS pointer points to the assertions list at the clause that was most recently resolved successfully with the first goal in the goals list. This is used to keep track of the remaining alternatives.

Given a goal frame (call it the parent frame) whose GOALS pointer points to some list of goals, the goals in the list are executed as follows. The first goal of the parent frame's goal list is made the current goal, and a pointer to it is stored in the variable GOAL (GOAL is a global variable, not part of the goal frame). The assertions list for the current goal is obtained from the assertions field of the main functor of the goal, and an attempt is made to resolve the current goal with the heads of the clauses in the assertions list. If this attempt is successful for some assertion in the list, then a goal frame is created (allocated on the backtrack stack), saving a pointer to the current binding environment in <ENV> and a pointer to the GOALS field of the parent frame in the REST.GOALS field. A pointer into the assertions list at the successfully resolved clause is saved in the ASSERTIONS field. If the computation ever has to backtrack to this point, it can then pick up the process of resolving the current goal with the assertions list at the point where it left off.

Assuming the resolution step is successful, the GOALS field of the new goal frame is filled with a pointer to the list of terms comprising the body of the clause just resolved (i.e., the body of the clause becomes the current goals list). If the body of the clause is empty (there are no subgoals), then the GOALS field of the current goal frame is filled with a pointer to the rest of the goals in the goals list of the parent frame. In other words, if there are no subgoals to consider, then the current goal (the first goal in the goal list of the parent frame) has been satisfied so the next step is to try and satisfy the remaining goals in the goal list of the parent frame.

If the attempt to resolve the current goal with the head of a clause in the assertions list fails for all clauses in the list, then the computation backtracks to the most recent goal frame that has remaining alternative ways to satisfy its goal. The computation proceeds until all the goals in the initial list have been satisfied (success) or until all the ways to satisfy the first goal in the initial list have been exhausted (failure).

The resolution and backtracking steps described are incorporated in the Forth words RESOLVE.FUN and RESOLVE.SINGLE. These words are the Prolog interpreter. RESOLVE.SINGLE is a special version of RESOLVE.FUN used to handle goals that have no arguments. The code addresses of these words are stored in the function-code and constant-code fields respectively of each named object (constants or variables). A Prolog procedure is called by placing a pointer to the appropriate field of a Prolog constant on the return stack and then exiting. The Forth inner interpreter then takes this return address as a pointer into the parameter field of some Forth word and so proceeds to execute either RESOLVE.FUN or RESOLVE.SINGLE, with the return stack containing a pointer that can be used to retrieve the assertions list.

Forth Interface

Built-in predicates are implemented in Forth, and use the predicate **builtin** as the interface between Prolog and Forth. **builtin** takes the name of a Forth word as its single argument and is distinguished from other predicates in that the function-code field of **builtin** does not contain the cfa of the word RESOLVE.FUN. Instead, control transfers directly to the Forth word that is the argument of the **builtin** call. This predicate is available for the user to access any of the underlying Forth system.

The only discipline required for user-defined built-ins is that the Forth word called drops the top of the return stack on entry and then exits by calling either $TRUE or $FALSE, which indicate, respectively, success or failure of the Prolog goal. Several Forth words are provided for parameter passing between Forth and Prolog. These words either retrieve the bindings of variables in the clause head or unify terms with variables in the head.

Invoking Prolog from Forth

Starting Up

Prolog is invoked by the Forth word PROLOG. The backtrack and binding stacks are initialized, and the Prolog interpreter is given the goal **prolog**, which implements a top-level read-execute-print loop:

```
prolog :- 'repeat ?-',read(X),execute(X).

    execute(X) :-
        (call(X),'print answer',tab(1),get0(Y),Y\=59,
        nl,display(yes) ;
        nl,display(no) ),
        !,fail.
```

The procedure **'repeat ?-'** is just a Prolog repeat which prints the **?-** prompt. The procedure **execute** takes the input goal and calls it. If the goal fails, **no** is output, **execute** fails and the prompt is repeated. If the goal succeeds, **'print answer'** prints the variable bindings and then waits for a single character input. If the character is **;** , we backtrack to **call** and try to satisfy the goal in another way. Otherwise **yes** is output, **execute** fails and the prompt is repeated.

A read-assert loop is entered by calling **user**:

```
user :- repeat,read(X),(X=stop;assertz(X),fail).
```

For example, the input text block that follows adds three clauses to the data base between **user** and **stop**. Following **stop** the user is back in the read-execute-print loop.

```
?- user.
likes(bob,mary).
likes(mary,X) :- is_funny(X).
is_funny(bob).
stop.
?-
```

In contrast to most Prolog implementations, variables retain their names in the internal representation of clauses. The negative aspect of this feature is that variable names take up space in the dictionary. One interesting experiment would be to implement predicates that create and switch Forth vocabularies. This would permit different Prolog data bases to exist in different vocabularies. Because of Forth we can get "hierarchical multiple worlds" for free.

Size, Speed and All That

The entire interpreter, including the parser but exclusive of the stacks, takes up on the order of 10K bytes. Running entirely in Forth (MicroMotion MasterForth on the Apple Macintosh), the interpreter will do about 22 LIPS (Logical Inferences Per Second, which effectively measure the procedure call rate) as timed with naive reverse. By recoding just the word `@BINDING` in assembler, there is a threefold increase in speed, and more efficiency can be gained by recoding other words in the inner loop of the Prolog interpreter. There is, thus, every indication that, with tuning, this implementation could be made competitive in speed with other Macintosh implementations of Prolog (see [PIE87] for a comparison of four Macintosh Prolog products). The version used in our application [PAL87] is written in polyFORTH running on a PC. No performance measurements have been made on the PC version.

One of the inefficiencies in the space usage of this implementation comes from not reclaiming goal frames. This means that the backtrack stack contains a goal frame for every inference that has been made, thereby saving more than is absolutely necessary for backtracking. A goal frame could be reclaimed on return (i.e., when `GOALS` list has only one element) if its `ASSERTIONS` list has only one element (no more alternatives).

Other Implementations

Several other implementations of Prolog in Forth have been mentioned in the literature. Harris has described a Prolog interpreter that has been used in space-related work [HAR86]; however, few details of this implementation have been published. The most extensive treatment is by Townsend and Feucht [TOW86] who present a very good discussion of the Prolog interpreter mechanism. Their implementation of binding environments is similar to that described here, and both are derived from the simple schemes used in the primitive LISP implementations of Prolog [NIL84].

It should be noted that the Prolog syntax used in [TOW86] is nonstandard, and no mention is made of the implementation of the standard Prolog built-in predicates; however, note that writing the Prolog parser is a major effort, as is writing a complete set of built-ins. In our implementation there are an equal number of screens (approximately 30) devoted to the interpreter, the parser and the built-in predicates.

Summary of Evaluable Predicates

The following is a list of the built-in predicates provided in this implementation, each with a brief description of its semantics. The majority of the Clocksin and Mellish predicates are included. Greater detail can be found in [CLO81].

arg(N,T,A)	The **N**th argument of term **T** is **A**.
asserta(C)	Assert **C** as first clause.
assertz(C)	Assert **C** as last clause.
atom(T)	Term **T** is an atom.
atomic(T)	Term **T** is an atom on an integer.
call(P)	Execute the Prolog procedure call **P**.
builtin(W)	Execute the Forth word `W`.
clause(P,Q)	There is a clause with head **P** and body **Q**.
consult(N0,N1)	Extend program with clauses from screen N0 thru N1.
display(T)	Display term **T** on terminal.
fail	Backtrack immediately.
functor(T,F,N)	The top functor of term **T** has name **F**, arity **N**.
get(C)	The next non-blank character input is **C**.
get0(C)	The next character input is **C**.
halt	Halt Prolog, exit to Forth.
integer(T)	Term **T** is an integer.
Y is X	**Y** is the value of the arithmetic expression **X**.
listing(P)	List the procedure(s) **P**.
name(A,L)	The name of the atom A is string L.
nl	Output a new line.
nonvar(T)	The term **T** is a non-variable.
not(P)	Goal **P** is not provable.
op(P,T,A)	Make atom A an operator of type **T**, precedence **P.**
put(C)	The next character output is **C**.
read(T)	Read term T.
repeat	Succeed repeatedly.
retract(C)	Erase the first clause of form C.
skip(C)	Skip input characters until after character C.
tab(N)	Output N spaces.
trace	Start tracing.
true	Succeed.
untrace	End tracing.
var(T)	Term **T** is a variable.
write(T)	Write the term T.
!	Cut any choices taken in the current procedure.
X < Y	As numbers, X is less than Y.
X =< Y	As numbers, X is less than or equal to Y.
X > Y	As numbers, X is greater than Y.
X >= Y	As numbers, X is greater than or equal to Y.
X = Y	Terms X and Y are unified.
X \= Y	Terms X and Y are not unified.
T =.. L	The functor and args of term T comprise the list L.
X == Y	The terms X and Y are strictly identical.
X \== Y	The terms X and Y are not strictly identical.

The following is a list of the Clocksin and Mellish predicates that have not been implemented, each with a brief description of its semantics.

debugging	List all current spy points.
nodebug	Remove all current spy points.
nospy P	Remove spy point from predicate **P**.
see(X)	Open file **X** for input.
seeing(X)	File **X** is open for input.
seen	Close file **X** for input.
spy P	Set a spy point on predicate **P**.
tell(X)	Open file **X** for output.
telling(X)	File **X** is open for output.
told	Close file **X** for output.

References

[CLO81] Clocksin, W. F., and Mellish, C. S. 1981. *Programming in Prolog*. New York: Springer-Verlag.

[HAR86] Harris, H. M. 1986. Development of an expert system for command and control of an orbiting spacecraft. *J. Forth Appl. and Res.* 4(2):305.

[NIL84] Nilsson, M. 1984. The world's shortest prolog interpreter? In *Implementations of Prolog*, ed. J. A. Campbell. Sussex, England: Ellis Horwood.

[PAL87] Paloski, W. H., Odette, L. L., Krever, A. J., and West, A. K. 1986. Use of a Forth-based Prolog for real-time expert systems. I. Spacelab life sciences experiment application. *J. Forth Appl. and Res.*, this issue.

[PIE87] Pierson, D. L. 1987. AI: four Prologs for the Macintosh. *Dr. Dobb's Journal of Software Tools* 12(4):30-41.

[TOW86] Townsend, C. and Feucht, D. 1986. *Designing and programming personal expert systems.* Blue Ridge Summit, PA: TAB Books.

Manuscript received August 1986.