22

Higher-order extensions to PROLOG: are they needed?

D. H. D. Warren
Department of Artificial Intelligence
University of Edinburgh

Abstract

PROLOG is a simple and powerful progamming language based on first-order logic. This paper examines two possible extensions to the language which would generally be considered "higher-order".† The first extension introduces lambda expressions and predicate variables so that functions and relations can be treated as 'first class' data objects. We argue that this extension does not add anything to the real power of the language. The other extension concerns the introduction of set expressions to denote the set of all (provable) solutions to some goal. We argue that this extension does indeed fill a real gap in the language, but must be defined with care.

1. INTRODUCTION AND SUMMARY OF PROLOG

PROLOG (Roussel 1975, Warren, Pereira and Pereira 1977 and Warren 1979) is a simple and powerful programming language based on first-order logic. It was conceived around 1971 by Alain Colmerauer at the University of Marseille, and has subsequently been put to use in a wide variety of applications, e.g. Bergman and Kanoui 1975, Warren 1976, Darvas, Futo and Szeredi 1977, Markusz 1977, Dahl 1977, Warren 1980, Bundy, Byrd, Luger, Mellish and Palmer 1979 and Dwiggins and Silva 1979.

The purpose of this paper is to discuss two possible extensions to PROLOG, both involving what are normally considered in the computing world to be 'higher order'† facilities. The first extension permits predicates to be treated as data objects; the second extension introduces expressions to denote the set of all solutions to some goal. The two sections of the paper covering these extensions can be read independently.

† Throughout this paper, 'higher-order' is used in the informal (computing) sense of 'pertaining to functions, sets, relations etc.'. The use of this term should not be taken to imply any particular connection with higher-order logic.

For readers not familiar with PROLOG, there follows a brief summary of the language. The syntax and terminology used is that of DEC-10 PROLOG (Pereira, Pereira and Warren 1978).

A PROLOG program comprises a set of *procedures*, each of which constitutes a definition of a certain *predicate*. A procedure consists of a sequence of *clauses*, which have the general form:

$P :- Q_1, Q_2, \ldots, Q_n.$

to be interpreted as:

"P (is true) if Q_1 and Q_2 and ... and Q_n (are true)".

If n is zero, the clause is written simply as:

$P.$

and interpreted as:

"P (is true)".

The P and Q_i are examples of *goals* or *procedure calls*, consisting of a predicate applied to some *arguments*.

For example, here is a procedure defining the predicate 'european' of one argument:

```
european(europe).
european(X) :- partof(X,Y), european(Y).
```

The clauses can be read as:

"Europe is European".
"For any X and Y, X is European if X is part of Y and Y is European".

The arguments of a procedure call are data objects called *terms*. A terms may be an (*atomic*) *constant*, a *variable*, or a *structure*. Variables are distinguished by an initial capital letter. Each variable of a clause is to be interpreted as standing for any arbitrary object.

A structure consists of a *functor* applied to some terms as *arguments*, and is written exactly like a goal, e.g.

```
point(2,3)
```

The functor should be thought of as a record type, and the arguments as the fields of a record.

A PROLOG program is invoked interactively by supplying a *question*, which may be thought of as a clause without a left-hand side, e.g.

```
?- partof(X,britain).
```

to be interpreted as:

"Is X part of Britain? (for any X you know of)"

The PROLOG system responds to a question by generating alternative instances of the variables for which it can deduce that the goal or goals are true. For this example, assuming there is an appropriate procedure for 'partof', these instances might be:

X = england ;
X = scotland ;
X = wales

To find such instances, PROLOG executes a goal by first *matching* it against the left-hand side of some clause and then executing the goals (if any) in the clause's right-hand side. Goals are executed in left-to-right order, and clauses are tried in the order they appear in the procedure. If a match isn't found (or there are no outstanding goals left to execute), PROLOG *backtracks*; that is, it goes back to the most recently executed goal and seeks an alternative match. Since the matching process is actually *unification* (*see* Robinson 1965), the net effect is to generate the most general instances of the goal for which it is true.

For convenience, certain predicates and functors may be written in infix notation, e.g.

13 divides 52 X + 1

and a special syntax is used for those structures called *lists* (cf. LISP), constructed from the constant '[ ]' and the functor '.' of two arguments, e.g.

[X|L] for .(X, L)
[a,b,c] for .(a,.(b,.(c,[])))
[a,b|L] for .(a,.(b, L))

As a final example, here is a procedure for the predicate 'concatenate (X,Y,Z)', meaning "the concatenation of list X with list Y is list Z":

```
concatenate([], L, L).
concatenate([X|L1], L2, [X|L3]) :- concatenate(L1, L2, L3).
```

Notice how what is usually thought of as a function of two arguments producing one result is here regarded as a predicate of three arguments. In PROLOG, a function is always regarded as just a special case of a relation, identified by a predicate with an extra final argument denoting the result of the function. An advantage of this approach is that not only can the function be called in the normal way, e.g. by:

```
?- concatenate([a,b],[c], L).
```

meaning:

"what is the concatenation L of [a,b] with [c] ?"

but also other usages are possible and just as easy to express, e.g.

```
?- concatenate (L1, L2, [a, b, c]).
```

meaning:

"Which lists L1 and L2 have concatenation [a,b,c] ?"

2. PREDICATES AS "FIRST CLASS" DATA OBJECTS

It has often been argued that a programming language such as PROLOG, based on first-order logic, lacks some of the power of "functional" languages such as LISP and POP-2, which provide what are considered to be "higher-order" facilities, namely the ability to treat functions as data objects. To be specific, in these functional languages it is possible to have a function call where the function to be applied is not fixed at "compile-time", but is determined by some run-time computation.

Now, as previously indicated, the procedures of PROLOG compute not only functions, but also more general relations. The PROLOG analogue of the function is therefore the predicate. So, to provide analogous higher-order facilities in PROLOG, one might wish to allow predicates to be treated as data objects, and in particular to allow the predicate symbol in a procedure call to be a variable.

For example, suppose one wanted to check whether all the elements of a list satisfied some property (i.e. unary predicate). A general procedure to do this would then be:

```
have_property([],P).
have_property([X|L],P) :- P(X), have_property(L,P).
```

These clauses can read as:

"All elements of the empty list have property P".
"All elements of the list comprising head X tail L have property P
if property P holds for X and all elements of L have property P".

The procedure call:

```
?- have_property([edinburgh,paris,san_francisco],attractive).
```

would then succeed, provided the following facts are provable:

```
attractive(edinburgh). attractive(paris). attractive(san_francisco).
```

Most functional languages also provide the ability to refer to functional objects "anonymously", through the means of lambda expressions. An analogous extension to PROLOG would allow the use of lamda expressions to denote predicates, where the body of the lambda expression would be a PROLOG goal (or goals). An example would be the call:

```
?- have_property([0,1], lambda(X).square(X,X) ).
```

meaning:

"Do 0 and 1 have the property of being equal to their squares?"
which should succeed, of course.

Do these two extensions – predicate variables and lambda expressions –

really increase the power of PROLOG? I shall argue that they do not, since both can be regarded merely as "syntactic sugar" for standard first-order logic. The mapping into first-order logic is very simple. A procedure call with a variable as predicate:

$P(t_1, \ldots, t_n)$

is transformed into the procedure call:

$apply(P, t_1, \ldots, t_n)$

and in addition a clause:

$apply(foo, X_1, \ldots, X_n) :- foo(X_1, \ldots, X_n).$

is supplied for each predicate *foo* which needs to be treated as a data object (where n is the arity of *foo*).† A lambda expression:

$lambda(X1, \ldots, Xn).E$

is replaced by some unique identifier, say *phi*, which is defined by a separate 'apply' clause:-

$apply(phi, X_1, \ldots, X_n) :- E.$

Effectively, we are just giving the anonymous predicate a name. Thus the second example rewrites to, say:

```
?- have_property([0,1], is_its_own_square).
```

with

```
apply(is_its_own_square,X) :- square(X,X).
```

where the clauses for 'have_property' are now:

```
have_property([],P).
have_property([X|L],P) :- apply(P,X), have_property(L,P).
```

Note that it is possible for a lambda expression to contain free variables. In this case, the identifier which replaces the lambda expression must be parameterised by the free variables. An example of this situation, which also illustrates the versatility of the PROLOG variable, is the following clause:

```
common(R,L,Y) :- have_property(L,lambda(X).R(X,Y) ).
```

to be interpreted as:

> "Attribute R has a common value Y for the elements of the list L if all elements of L have the property of being paired with Y in relation R".

This clause might be invoked by the goal:

```
?- common(pastime,[tom,dick,harry],P).
```

† Note that we have a different 'apply' predicate for each different value of n; we could have given these predicate distinct names if preferred, (say 'apply1', 'apply2', etc.).

to produce alternative solutions of, say:

P = football
P = darts

given that it is possible to prove facts like:

pastime(tom, football).

Here the lambda expression:

lambda(X).R(X,Y)

has free variables 'R' and 'Y', so if we identify it by, say, 'foo(R,Y)', the clause for 'common' rewrites to:

common(R, L, Y) :- have_property(L, foo(R,Y)).

with:

apply(foo(R,Y), X :- apply(R, X, Y).
apply(pastime, X, Y) :- pastime(X, Y).

Observe that PROLOG produces the solutions by first generating a pastime P of 'tom', and then checking that P is also a pastime of 'dick' and 'harry'. Thus the free variables of a lambda expression need not all be known initially. In this respect, we seem to have something essentially more powerful than what is available in functional languages.

Notice also that this treatment of the non-local variables in a lambda expression corresponds exactly to "static binding" as in ALGOL or SCHEME (Steele 1978) in contrast to the "dynamic binding" of LISP and POP-2. Dynamic binding has the curious property that the name chosen for a variable is semantically significant. This would clearly be incongruous in the PROLOG context, and can be viewed as a flaw in the original definition of LISP which has led to the variant, SCHEME.

As a final, more sophisticated, example of treating predicates as data objects, let us look briefly at the PROLOG equivalent of the definition of the 'twice' function (see for example, Turner 1979). In a functional language this is defined as:

twice = lambda(F).(lambda(X).F(F(X)))

Thus 'twice' maps a function F into another function which is the composition of F with itself. If we also have:

succ = lambda(X).X+1

then:

twice(twice)(twice)(succ)(0)

is a valid expression, which in fact evaluates to 16. (Check this for yourself!) Using the rewrite rules outlined above in conjunction with the standard technique

for translating functional notation into predicate notation, this problem translates into PROLOG as:

```
?- apply(twice, twice, F1), apply(F1, twice, F2),
                            apply(F2, succ, F3), apply(F3, 0, Ans).
```

where

```
apply(twice, F, twice(F)).
apply(twice(F), X, Z) :- apply(F, X, Y), apply(F, Y, Z).
apply(succ, X, Y) :- Y is X+1.
```

(where 'is' is a PROLOG built-in predicate that evaluates arithmetic expressions). Executing this question with PROLOG does indeed produce the right result:

```
Ans = 16
```

and the values of the other variables help to show how this result is obtained:

```
F1 = twice(twice),
F2 = twice(twice(twice)),
F3 = twice(twice(twice(twice(succ))))
```

Discussion

To summarise the argument so far: functional languages such as LISP and POP-2 allow functions to be treated as "first class" data objects, and this property is often claimed to give these languages an added power lacking in PROLOG. I have therefore put forward two extensions to PROLOG – predicate variables and lambda expressions – providing what I believe is the exactly analogous property for PROLOG, that of making predicates into "first class" data objects. This belief is backed up by the examples given. I have then shown how these extensions can be regarded as mere syntactic sugar, by presenting a very simple way of translating them back into standard first-order logic.

The translation is such that it does not involve any *unbounded* increase in either the size of the program or in the number of execution steps. I therefore claim that the extensions do not add anything to the strict power of PROLOG; i.e., they do not make it feasible to program anything that was not feasible before (in contrast to other possible extensions – see later, for example).

Of course "power" is often used more loosely to cover such language properties as conciseness, clarity, or implementation efficiency (measured in bytes and milliseconds). So let us consider, on such wider grounds, two possible arguments for nevertheless incorporating these extensions into PROLOG.

1. The extensions should be provided as primitive in the language so that they can be implemented efficiently.
2. The extended syntax is much nicer to read and to use and should be provided as standard.

We will discuss these points one by one.

Efficiency

The standard way of implementing functional objects involves a representation known as a *closure.* Essentially, a closure is a pair of pointers, one pointing to the code for the function concerned, the other pointing to an *environment* containing the values of the function's non-local variables. The advantage of this representation is that it avoids having to make a copy of the function's code for each different binding of the non-local variables.

Now PROLOG implementations commonly use an implementation technique known as *structure-sharing*, whereby any non-atomic data object has a representation very similar to a closure. This representation, called a *molecule*, likewise consists of two pointers, one pointing to (a representation of) the original source term, the other pointing to an environment containing the values of the variables occurring in that source term.

The result of structure-sharing, combined with our technique for translating higher-order expressions into first-order logic, is that higher-order objects automatically get something very close to their standard, closure representation. In particular, we do not need to do any copying of code or variable values in order to create the higher-order object. The only significant difference is that, instead of a direct pointer to the code of the higher-order object, we have an identifier which is mapped into the corresponding code via the appropriate 'apply' clause.

This difference is minimal if the clauses for 'apply' are appropriately indexed on the procedure's first argument, as is provided automatically in DEC-10 PROLOG, for instance. It is then possible to get from the identifier of the higher-order object to its code in a small, fixed number of steps – effectively, the identifier is an indirect pointer.

So, for DEC-10 PROLOG at least, providing a specific implementation of higher-order objects would not produce huge efficiency gains over the approach I am advocating, and for most programs the overall improvement would probably be negligible.

Note that it appears to be possible to apply a directly analogous technique to functional languages to "pre-process away" higher-order expressions. Structures would be constructed to represent the higher-order objects, and an arbitrarily big 'apply' function would be needed to invoke the corresponding code. However the result is not nearly so practical as in the PROLOG case, since, apart from being much less readable, it involves explicitly copying non-local variables and (in the absence of the equivalent of a case expression) executing an arbitrarily large number of tests in the 'apply' function. Moreover, while it is quite natural to add new 'apply' clauses one by one as the need arises, it is much less convenient to have to make modifications to the body of an 'apply' function.

Readability and usability

The extended syntax is obviously more concise, but whether this makes programs easier to understand is highly debatable. It seems to be a matter of taste which syntax one prefers.

Most PROLOG users would argue that one of PROLOG's main strengths is that it avoids deeply nested expressions. Instead the program is broken down into smaller, more easily comprehended units. The resulting program may be slightly longer, but it is easier to read and (what is especially important) easier to modify.

Now the effect of introducing lambda expressions is directly contrary to this philosophy. It results in bigger, more deeply-nested expressions. I am therefore certainly against this part of the extension.

The introduction of predicate variables, however, seems to have a certain elegance. For example, I would probably prefer to write the 'twice' example as:

```
?- twice(twice,F1), F1(twice,F2), F2(succ,F3), F3(0,Ans).
twice(F,twice(F)).
twice(F,X,Z) :- F(X,Y), F(Y,Z).
succ(X,Y) :- Y is X+1.
```

I understand recent PROLOG implementation (see Colmerauer, Kanoui and van Caneghem 1979) allows this kind of syntax.

However, if predicate variables are used in more than small doses, the program becomes excessively abstract and therefore hard to understand. For example, it is possible to define a higher-order procedure 'iterate':

```
iterate([],F,Z,Z).
iterate([X|L],F,Z,Y) :- iterate(L,F,Z,Y0), F(X,Y0,Y)
```

such that list concatenation, and the summation of the elements of a list, are special cases:

```
concatenate(L1,L2,L3) :- iterate(L1,cons,L2,L3).
sum(L,N) :- iterate(L,add,0,N).
```

But this seems a particularly perverse way to define such simple predicates.

On balance, therefore, I do not think the benefits of predicate variables are worth the extra language complexity involved.

3. SET EXPRESSIONS

A difficulty which programmers new to PROLOG soon come up against is that they need to combine information generated on alternative branches of the program. However, with pure PROLOG, all information about a certain branch of the computation is lost on backtracking to an earlier point. To take a very simple example, suppose we have the following facts about the 'drinks' predicate (written in infix notation):

```
david drinks beer.
david drinks milk.
jane drinks water.
ben drinks milk.
```

then it is very easy to represent the question:

"Who drinks milk?"

One simply writes:

?- X drinks milk.

and PROLOG's backtracking generates the two solutions:

X = david ;
X = ben

But how should one represent the question:

"How many people drink milk?"

It seems we already have all the information needed to compute the solution, which should obviously be "2". Somehow we need to combine the solutions generated by backtracking into a single data structure so that they can be counted by a straightforward (recursive) procedure. The problem is how to remember one solution when we backtrack to get another.

At this point, the logician might point out a flaw in the programmer's reasoning. We have no right to assume the answer to the question is "2", since, in the absence of any information about who does not drink milk, it is quite possible that there are other facts we don't know. Maybe, say:

jane drinks milk.

Therefore, from a logical point a view, the question of how many people drink milk can't be answered with the information available.

The programmer, however, is unlikely to be satisfied with this argument, and may explain that what he really meant was:

"How many people are known to drink milk?"

which is surely a reasonable query having "2" as its unequivocal answer.

Ways to get the desired result in this and similar cases are part of the PROLOG folklore. They involve going outside the pure PROLOG described here, in order to preserve information on backtracking. This is done using certain extra facilities provided by Prolog implementations as "built-in procedures". Typically these procedures modify the program by adding and deleting clauses. The trouble with these *ad hoc* solutions is that they generally only provide a partially correct implementation of what is intended, and what is intended is often far from clear. Above all, the parts of the program affected can no longer be interpreted simply as a shorthand for statements of fact in natural language, in the way we have seen so far.

I believe it is possible to replace these *ad hoc* solutions with a single more principled extension to PROLOG, which preserves the "declarative" aspect of the language. This extension has already been incorporated in the latest version of DEC-10 PROLOG. The implementation is essentially an encapsulation of the standard hack in a more general and robust form.

The extension takes the form of a new built-in predicate:

setof(X,P,S)

to be read as:

"The set of instances of X such that P is provable is S".

The term P represents a goal or goals, specified exactly as in the right-hand side of a clause. The term X is a variable occurring in P, or more generally any term containing such variables. The set S is represented as a list whose elements are sorted into a standard order, without any duplicates.

Our previously problematic query:

"How many people (are known to) drink milk?"

can now be expressed as:

```
?- setof(X, X drinks milk, S), size(S, N).
```

which can be interpreted more literally as:

"Is it true of any S and N that
the set of those that drink milk is S and the size of the set S is N?"

to which the Prolog response is:

```
S = [ben, david], N = 2
```

This assumes a definition of 'size'. This predicate simply computes the length of a list, and is easily defined:

```
size([], 0).
size([X|L], N1) :- size(L,N), N1 is N+1.
```

The 'setof' primitive has certain inevitable practical limitations. The set to be enumerated must be finite, and furthermore it must be enumerable by PROLOG in finite time. If there are any infinite branches in a search space, execution of the 'setof' expression will simply not terminate.

Note that since PROLOG always generates most general instances, it is possible to generate set elements containing variables. Our implementation of 'setof' does not prohibit this. However in such cases the list S will only provide an imperfect representation of what is in reality an infinite set. Effectively, the list contains only a single, arbitrary representative of the infinitely many different ways of instantiating each solution containing variables.

The 'setof' primitive must be defined and implemented with particular care if it is to behave correctly in any context. For example, suppose we wish to define a general predicate 'drunk by(D,N)' meaning "D is drunk by a total of N individuals". Then a suitable definition easily follows from our previous example:

```
drunk_by(D,N) :- set of(X, X drinks D, S), size(S, N).
```

A question such as:

```
?- drunk_by(milk, N).
```

will then produce, without difficulty, the solution:

N = 2

But what happens if we ask the question:

?- drunk_by(D,1).

meaning:

"What is drunk by just 1 individual?"

Obviously the correct result should be to produce the two alternative solutions:

D = beer ;
D = water

However this entails that our 'setof' primitive should be "backtrackable", generating alternative sets if the set expression contains any uninstantiated free variables. In the example just given, 'D' was such a variable.

Our implementation of 'setof' fully takes care of such cases. Notice that this also makes it quite all right for set expressions to be nested. For example, one possible way to express the query:

"Which people drink each beverage?"

is:

?- setof(Beverage-People, setof(X, X drinks Beverage, People), S).

Here we have a case where the first argument of a 'setof' is a structure rather than a single variable; ('-' is an infix functor). The response to this question will be:

S = [beer-[david], milk-[ben, david], water-[jane]]

To allow 'setof' to be backtrackable, we have to slightly restrict its meaning. All our sets are implicitly non-empty. Otherwise there would be indefinitely many alternative ways of instantiating the free variables of a set expression to produce the empty set as solution. For example, if empty sets were allowed, the question:

?- setof(X, X drinks D, S).

should presumably have solutions like:

D = treacle, S = [] ;
D = kerosene, S = [] ;

and so on ad infinitum. In general, 'S' is empty where 'D' is anything that is not known to be drunk by somebody.

The problem here is equivalent to that of producing a correct implementation of negation (regarded as non-provability), for which there doesn't yet appear to be a fully satisfactory solution.

Although the restriction to non-empty sets can reasonably be seen as a shortcoming, it does often automatically capture what is required. For example, try formulating the question:

"Which employee's children are all at university?"

Obviously we don't really want to count as a solution an employee who doesn't have any children.

4. CONCLUSION

We have looked at two possible "higher-order" extensions to PROLOG. We have seen that the first extension, which permits predicates to be treated as 'first class' data objects, does not really contribute to the power of the language, although some would argue that it is useful syntactic sugar. In contrast, the addition of set expressions to PROLOG seems to fill a long felt need, and many problems cannot be expressed in pure PROLOG without such an extension. We have seen that it is important for set expressions to be "backtrackable", so that they can be used freely in any context.

Acknowledgements

The 'setof' primitive described here was particularly influenced by the work of A. Colmerauer 1977 and V. Dahl 1977 on natural language analysis. Helpful comments on the paper were made by S. Jones and F. Pereira. The support of the British Science Research Council is gratefully acknowledged.

REFERENCES

Bergman, M. & Kanoui, H. (1975). SYCOPHANTE: Système de calcul formel et d'interrogation symbolique sur l'ordinateur. Marseille: Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II.

Bundy, A., Byrd, L., Luger, G., Mellish, C. & Palmer, M. (1979). Solving mechanics problems using meta-level inference. *Expert Systems in the Micro-Electronic Age*, pp. 50–64 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Colmerauer, A., Kanoui, H. & van Caneghem, M. (1979). Etude et realisation d'un système PROLOG. Marseille: Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II.

Colmerauer, A. (1977). Un sous-ensemble intéressant du francais. *R.A.I.R.O*, **13**, 4, pp. 309–336. (Presented under the title of "An interesting natural language subset" at the Workshop on Logic and Databases, Toulouse 1977).

Dahl, V. (1977). Un système deductif d'interrogation de banques de données en Espagnol. Marseille: Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II.

Darvas, F., Futo, I. & Szeredi, P. (1977). Logic-based program system for predicting drug interaction. *Int. J. Biomed. Comp.*

Dwiggins, D. L. & Silva, G. (1979). A Knowledge-based automated message understanding methodology for an advanced indications system. *Technical Report No. R79-006* Woodland Hills, Ca: Operating Systems Inc.

Markusz, Z. (1977). How to design variants of flats using programming language PROLOG based on mathematical logic. *Information Processing* 77, pp. 885–889 (ed. Gilchrist, B.). Amsterdam, New York, Oxford: North Holland Publishing Co.

Pereira, L. M., Pereira, F. & Warren, D. H. D. (1978). User's guide to DEC system-10 PROLOG. *D.A.I. Research Paper No. 154*. Edinburgh: Department of Artificial Intelligence, University of Edinburgh.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. Ass. Compt. Mach.*, **12**, 227–234.

Roussel, P. (1975). PROLOG: manuel de référence et d'utilisation. Marseille: Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II.

Steele, G. L. (1978). RABBIT: a compiler for SCHEME. M.Sc. Thesis: also published as *AI-TR-474*. Cambridge, Mass: Artificial Intelligence Group, Massachusetts Institute of Technology.

Turner, D. A. (1979). A new implementation technique for applicative languages. *Software Practice and Experience*, 9, 31–49.

Warren, D. H. D. (1976). Generating conditional plans and programs. Preprint: AISB Summer Conference, Edinburgh.

Warren, D. H. D. (1980). Logic programming and compiler writing. *Software Practice and Experience*, 10, 97–125.

Warren, D. H. D. (1979). PROLOG on the DECsystem-10. *Expert Systems in the Micro-Electronic Age*, pp. 112–121 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Warren, D. H. D., Pereira, L. M. & Pereira, F. (1977). PROLOG – the language and its implementation compared with LISP. Symposium on AI and Programming languages, ACM SIGPLAN 12.