## PROLOG ON THE DECsystem-10

David Warren

### 1. INTRODUCTION

Prolog [19] [16] is a simple and powerful programming language for non-numeric applications. It was originally devised around 1972 for the purpose of implementing a natural language question-answering system [8], by a team under Alain Colmerauer at the University of Marseille. At Edinburgh, we have produced a Prolog compiler/interpreter [23] for the DECsystem-10 [13]. This implementation shows Prolog to have an efficiency comparable with (compiled) Lisp.

Prolog has been put to practical use in a number of areas outside pure research. Examples include a package for doing algebraic "symbol crunching" [3], an architectural design aid to assist in planning the layout of a building [14], a system to help predict the properties of organic compounds (needed in designing drugs) [12], and the implementation of a compiler (DEC-10 Prolog itself) [24]. Applications within Artificial Intelligence research include programs for plan generation [22], equation solving [6], natural language analysis [10], and solving mechanics problems [5]. All of the above are large and complex programs which would probably never have got written at all with the available manpower, were it not for the relative ease of writing them in Prolog.

The basic idea of Prolog is that a collection of logic statements of a restricted form – **clauses** – can be regarded as a program, and that the execution of such a program is nothing other than a suitably controlled logical deduction from the clauses forming the program.

It is, however, not necessary to know about logic to understand and use Prolog. A Prolog program can be regarded simply as a collection of statements of fact – the **declarative view**. The program can also be understood as a number of procedure definitions – the **procedural view**. The different clauses in a procedure represent alternative **cases** of the procedure. The appropriate clause (or clauses) is selected by a **pattern matching** operation (**unification** [18]), according to the form of the procedure call. Pattern matching is the **sole** data manipulation operation. Data items in Prolog are called **terms**, and may be thought of as complex record structure written in a textual, machine independent form, not involving the notion of reference or pointer.

From a practical point of view, programmers like Prolog because it enables them to write clearer, more concise programs, with less effort, and with less likelihood of error. The language could perhaps be summed up as " pointer manipulation made easy".

### 2. THE LANGUAGE

#### Procedures

A Prolog program consists of a sequence of statements called **clauses**. Here is

a simple example, consisting of six clauses:

```
    descendant (X, Y) :- offspring (X, Y).
    descendant (X, Z) :- offspring (X, Y), descendant (Y, Z).

    offspring (abraham, ishmael). offspring (abraham, isaac).
    offspring (isaac, esau).      offspring (isaac, jacob).
```

Clauses can be understood in two ways. Firstly, they can be interpreted as statements of fact. For instance the first clause says that, whatever may be the values of the **variables** X and Y, "Y is a descendant of X if Y is one of the offspring of X". And the last clause says that "Jacob is one of the offspring of Isaac". Note that variables in different clauses are considered distinct, even if they have the same name.

The second way to understand clauses is as pieces of program. Each clause corresponds to a "case" of a **procedure**. Looked at in this way, the first clause can be read as "To find a Y that is a descendant of X, find a Y that is one of the offspring of X", and the last clause as "When seeking an offspring of Isaac, return the solution Jacob".

The six clauses of the example serve to define two procedures, named 'descendant' and 'offspring'. Each clause consists of a **head**, or "procedure entry point", followed by a (possibly empty) **body**. The body consists of a number of **goals**, or "procedure calls". A clause with an empty body is called a **unit clause.**

How does a Prolog program actually work? To run the program, one provides an initial goal such as:

```
    descendant (abraham, X)
```

The result of executing this goal will be to enumerate descendants of Abraham and return them, one by one, as values of the variable X.

To execute such a goal, the Prolog system **matches** it against the head of some clause and then executes the goals (if any) in the body of that clause, in left-to-right order. In seeking a match, Prolog tries the clauses of the procedure concerned in the order they appear in the program text. The matching process, known technically as **unification**, succeeds if the goal and clause head can be made identical by "filling in" suitable values for the variables. For example the goal 'offspring (X, ishmael)' matches the first clause for 'offspring' if X is given the value 'abraham'. The variable X is then said to be **instantiated** to 'abraham'. When one solution to a goal has been finished with, or when no match can be found for a goal, the Prolog system **backtracks.** That is, it goes back to the most recently executed goal, and looks for an alternative match. If backtracking generates more than one solution to a goal, the corresponding procedure is said to be **non-determinate.**

So what happens when the initial goal 'descendant (abraham, X)' is executed? Through matching the goal against the first clause for 'descendant', Prolog starts off by looking for the immediate offspring of Abraham, and returns successively X = 'ishmael' and X = 'isaac'. Then backtracking causes the second clause for 'descendant' to be used. This results in the 'descendant' procedure being called recursively for each of the offspring of Abraham, giving further descendants, Esau and Jacob.

## Structures

Prolog data objects are called **terms**. So far, the only kinds we have seen have been variables, and unstructured constants (called **atoms**). Prolog also provides for structured data objects (called **complex terms**). An example is the binary tree data type. The following procedure checks whether a particular item is present in an ordered binary tree:

```
in (X, tree (T1, X, T2)).
in (X, tree (T1, Y, T2)) :- before (X, Y), in (X, T1).
in (X, tree (T1, Y, T2)) :- before (Y, X), in (X, T2).
```

Here 'tree' is a **functor** of 3 **arguments**. It can be thought of as a record type with 3 fields. The arguments stand for the left subtree, the item at the root node, and the right subtree. The first clause says that X is present in the ordered binary tree <T1, X, T2>, for any values of X, T1 and T2. The last clause says that X is present in the ordered binary tree <T1, Y, T2> if Y is before X and X is present in T2, for any values of X, Y, T1 and T2.

Another, very commonly used, data type is the **list**. These are essentially the same structures as in Lisp. For example, here is the Prolog procedure for concatenating lists:

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

We read these clauses as 'the empty list concatenated with L yields L' and 'a list of the form X followed by L1 concatenated with L2 yields a list of the form X followed by L3, where L1 concatenated with L2 yields L3'. Thus a Prolog list is either the atom '[ ]' (the empty list), or a structured object such as '[X | L]', where X is the first element of the list and L is a variable standing for the remainder, or "tail", of the list. In fact '[X | L]' is just an ordinary term of two arguments and a particular functor, written in a special syntax. The syntax also allows one to write '[1, 2, 3]' instead of '[1 | [2 | [3 | [ ]]]]'. Executing the procedure call:

```
concatenate ([1, 2, 3], [4, 5], L)
```

produces as value for L the list '[1, 2, 3, 4, 5]'. However the 'concatenate' procedure can be used much more flexibly than this. For example, execution of:

```
concatenate (L1, L2, [1, 2, 3])
```

will return, as successive values for L1 and L2, all pairs of lists which when concatenated give the list '[1, 2, 3]'.

As a final, more meaty, example, here is a Prolog version of Hoare's "quick-sort" algorithm:

```
qsort ([ ], L, L).
qsort ([X | L], R1, R) :-
    split (L, X, L1, L2), qsort (L1, [X | R2], R), qsort (L2, R1, R2).
split ([ ], Y, [ ], [ ]).
split ([X | L], Y, [X | L1], L2) :- X ≤ Y, split (L, Y, L1, L2).
split ([X | L], Y, L1, [X | L2]) :- X > Y, split (L, Y, L1, L2).
```

'X ≤ Y' and 'X > Y' are calls to **built-in** procedures which compare numeric values. As an example of how "quick-sort" is used:

```
qsort ([1, 9, 8, 4], [], L)
```

returns '[1, 4, 8, 9]' as the value of L. The interested reader should be able to figure out how the program works from what has been said already.

## 3. IMPLEMENTATION

At first sight, Prolog doesn't appear to be at all a machine-oriented language. However, in terms of efficiency, it can compare very favourably with other high-level languages.

I'll now describe some of the key ideas behind Prolog implementation, with particular reference to the DEC-10 compiler/interpreter. Incidentally, interpreters compatible with the DEC-10 system have been written at Edinburgh for PDP-11 [15] and (in IMP under EMAS) for ICL-system4 [11]. Other Prolog interpreters include implementations in Fortran [2], Pascal [4] and CDL [21], and systems for IBM-370 [17] and the Motorola 6800 microprocessor [9]. Other, more experimental, systems for logic programming also exist [7].

### Compilation

Although most existing Prolog systems are interpreters, it is perfectly feasible to compile Prolog into the kind of instruction set typical of present-day hardware. The only case in which a call to an out-of-line routine is essential is in clauses where a particular variable occurs more than once in the head. However. to achieve compact in-line code, many other operations will typically need to be performed out-of-line.

Each symbol (ie. functor or variable occurrence) in the head of the clause will compile into instructions responsible for matching that symbol against the corresponding term in the goal. In general a symbol will map into two pieces of code. One piece will deal with the case of decomposing an existing structure. The other will be needed in the case where a new structure has to be created. However, this duplication of information can be avoided where a procedure argument is known to be either always **input** (ie. instantiated to a non-variable in the call) or always **output** (ie. instantiated to a variable in the call). The user can notify such restrictions on procedure usage by supplying optional **mode declarations**, e.g.

```
:- mode concatenate (+, +, –).
```

This declares that 'concatenate' will always be called with the first two arguments as input and the last as output.

The system can also exploit mode declarations to generate much faster code, and to save run-time storage in "structure-sharing" implementations. ("Structure sharing" is one possible method for representing the new structures (ie. complex terms) created during a Prolog execution, and is explained in [25]).

## Indexing

The number of clauses in a procedure is often quite large. This is typical of "database" applications, where a relation is represented "extensionally" as many unit clauses. See, for instance, the way 'offspring' was defined above. However, even when a procedure is made up largely of non-unit clauses, there can still be a lot of them, corresponding to a procedure with many "cases". This is typical, for instance, of compiler writing applications.

With either type of procedure, one does not want the system to have to run through all the clauses in order to find the ones which match. The DEC-10 compiler therefore **indexes** the clauses according to the atom or principal functor of the first argument in the head. (Normally a user will choose this position for the main input to a procedure). Then, provided the first argument in the call is instantiated to a non-variable (ie. the argument actually is input), the only clauses which need to be even considered for a match are those which have the same functor, or a variable, in the indexed position. Often there will be only one alternative. The list of potentially matching clauses is found by a hash-coding technique in a time that does not depend on the number of clauses in the procedure. This relatively simple indexing mechanism is quite adequate in most practical cases.

## Bookkeeping – the "Tail Recursion" Optimisation

As in most high-level languages, the Prolog system maintains a stack of **frames**, one for each **active** procedure. Each frame contains bookkeeping information, together with the values of variables in the clause concerned.

Because of Prolog's non-determinancy, a procedure may remain active even though it has successfully "returned". For it may still be possible to back-track back into the procedure. In such cases the stack frame must be retained. However when a procedure returns determinately – ie. with no choices remaining inside it – it is possible to reclaim the stack frame, exactly as in a conventional language. Stack frames are also reclaimed on backtracking, as are any new structures created by the procedure. Since backtracking will always occur eventually, a conventional garbage collector is not strictly essential with Prolog (in contrast to most languages which allow a procedure to return structures). Nevertheless, DEC-10 Prolog does include a garbage collector, because the amount of structure created may grow too large to fit in available memory before backtracking gets round to reclaiming it.

As I mentioned, each stack frame contains bookkeeping information. First, there is the **call information**, a pair consisting of a pointer into the code for a clause (indicating the arguments of the call, and the position at which to resume

execution: the **return address**), together with a pointer to the stack frame associated with that clause. Secondly, there are four items needed principally for backtracking. These are:

(a) a pointer to the clause (if any) which is the next alternative in this procedure;

(b) a stack pointer indicating where to backtrack to should this procedure fail;

(c) a pointer to a push-down list of variable addresses, called the **trail**, which is used on backtracking to reset variables which have been assigned to during unification;

(d) a pointer to a frame in an auxiliary stack, which contains the structures (if any) created by this procedure.

With the current DEC-10 system, this information is created on procedure entry, and is discarded on backtracking or, if the procedure is determinate, on reaching the end of a clause. However, it is possible to adopt a more sophisticated strategy, which brings quite a profound improvement. The reclaiming of the stack frame in the determinate case does not have to wait until the end of clause. It can be accomplished immediately prior to executing the last goal in the clause (provided no choices have been taken up until that point in the procedure). This modification is analogous to what has been called a "tail recursion" optimisation in other languages [20].

I am currently working on a new DEC-10 Prolog compiler, which incorporates this and other improvements. Most of the essentials are already complete, but much detailed work remains to be done before it can be released as a usable system.

To support the tail recursion optimisation, there are some details to attend to:

(a) The values of the procedure's arguments must be copied out into registers, and later stored in the new frame as extra bookkeeping information (looking exactly like ordinary variable cells).

(b) The call information now consists just of a return address and its associated stack frame pointer. This pair is called a **continuation**. The return address can no longer point to the end of a clause. Instead the system keeps track of the actual goal to be executed next.

(c) There is a snag if one of the procedure's arguments is a pointer to an uninstantiated variable in the frame about to be discarded. In practice this occurs quite rarely, but some fix must be found.

The most obvious benefit of this optimisation is that it saves stack space. 'Quicksort' now only requires a stack of size order log N instead of order N. And 'concatenate' now never uses more than **one** stack frame! Also, a determinate, tail recursive procedure which creates no new structure can now recurse indefinitely, without being limited by the size of the stack. However, in practice, most Prolog procedures do create structure, so the effect on total working storage requirements (which is what the user is aware of) is usually less dramatic.

The less obvious, but perhaps more important, benefit of this optimisation is that is saves time. When one frame is discarded and another overwrites it, most of the bookkeeping information can be retained intact. The continuation remains the same. And it turns out that none of the other items needs to be touched, except perhaps the auxiliary stack pointer in cases where new structure has been created.

The net result is that the final call in a determinate procedure is little more than a simple **goto**. For instance, if 'concatenate' has mode '(+, +, –)', the main clause:

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

compiles essentially into the following iteration:-

```
    while Arg1 is a non-empty list
    do
        let List be a new record with 2 fields;
        head (List): = head (Arg1);
        field pointed to by Arg3: = the list List;
        Arg3: = address of tail (List);
        Arg1: = tail (Arg1)
    repeat
```

Some systems for other languages, eg. SCHEME [20], also automatically perform a tail recursion optimisation. However, as far as I know, none of them will produce iterative code for 'concatenate' (for example), because it is only in Prolog that the recursive call to 'concatenate' is the last thing to be done in the procedure. In other languages, the last step is a call to **cons** (the built-in function which creates a new list cell). The unique feature of Prolog which is being capitalised on here is the ability to create a new structure before all the parts are known, and to leave the unknown parts as variables.

Although obviously a programmer can write an iterative version of 'concatenate' if he uses a sufficiently machine-oriented language, he will have to grapple with pointers and pointer assignments, and it is easy to make mistakes. He'll probably decide that it isn't worth the trouble, and will stick to the simple recursive version. Thus by taking over responsibility for machine-oriented matters, the Prolog compiler can actually produce better code than is likely from a programmer in a low-level language.

### Performance

The performance of the code produced by the current DEC-10 Prolog compiler has been compared with that of the Stanford DEC-10 Lisp compiler, which is recognised to produce quite fast code. For simple functions over lists, Prolog is somewhat slower than Lisp (by a factor of about 0.6). For examples requiring more general structures, which have to be encoded as lists in Lisp, Prolog can be significantly faster than Lisp (by a factor of as much as 2 or more).

It is too early to say precisely how the new Prolog compiler will affect these figures. However a hand calculation indicates that 'concatenate' will be 1.8 times faster than before. Although this example is in some ways a best case, it is nevertheless probably quite representative of the innermost cycle of a typical

Prolog program.

The improved speed comes partly from the tail recursion optimisation, and partly from taking fuller advantage of mode declarations. Experiments with handcoding 'concatenate', keeping the same data representation, suggest that there is room for further improvement by only a factor of 2, and most of this improvement comes from avoiding jumps to and from out-of-line routines, which greatly increases the size of the in-line code. The compiler, on the other hand, strives to produce compact code, which is more important for most users than speed.

### Interactive Environment

Performance is all very well. What the programmer really needs is a good interactive environment for developing his programs. To address this need, DEC-10 Prolog provides an interpreter in addition to the compiler.

The interpreter allows a program to be read in quickly, and to be modified on-line, by adding and deleting single clauses, or by updating whole procedures. Goals to be executed can be entered directly from the terminal. An execution can be traced, interrupted, or suspended while other actions are performed. At any time, the state of the system can be saved, and resumed later if required. The system maintains, on a disk file, a complete log of all interactions with the user's terminal. After a session, the user can examine this file, and print it out on hard copy if required.

## 4. CONCLUSION

Most so-called high-level languages still perpetuate low-level concepts that in reality are a hangover from machine code programming. Examples are assignment, pointers, **gotos** and iteration. These concepts presuppose a particular computer architecture, what John Backus has called "the von Neumann computer".

No such concepts exist in Prolog (as far as the user is concerned). One might say that Prolog is a truly high-level language. But, as this paper has sought to show, it is often possible to compile Prolog into code which is almost exactly equivalent to that which would have been obtained by programming in a conventional language. In some cases, as an example has shown, the Prolog code is actually likely to be better. In a sense, unification subsumes assignment and pointers, while Prolog's procedure call subsumes **gotos** and iteration. And all this can be achieved with a relatively simple compiler, without the need for preliminary program transformation.

## ACKNOWLEDGEMENTS

The suggestion for putting a tail recursion optimisation into Prolog came from Keith Clark. I argued at first that the optimisation was not worthwile, because the space saving would be barely perceptible in most cases, but changed my mind when I realised the potential for improving speed as well.

My work is supported by a Science Research Council grant.

## REFERENCES

[1] Backus J.
Can programming be liberated from the von Neumann style?
*CACM* 21 (8): 613-641, August 1978.

[2] Battani G and Meloni H.
*Interpreteur du langage de programmation Prolog.*
Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1975.

[3] Bergman M and Kanoui H.
*Sycophante: Systeme de calcul formel et d'interrogation symbolique sur l'ordinateur.*
Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1975.

[4] Bruynooghe M.
*An interpreter for predicate logic programs: Part 1.*
Applied Maths & Programming Division, Katholieke Univ Leuven, Belgium, 1976.
Report CW 10.

[5] Bundy A. *et al.*
*Solving mechanics problems using meta-level inference.*
(this volume)

[6] Bundy A and Welham R.
*Using meta-level description for selective application of multiple rewrite rules in algebraic manipulation.*
Dept of Artificial Intelligence, Univ of Edinburgh, 1979. Working Paper (forthcoming).

[7] Clark K.L. and McCabe F.G.
*The control facilities of IC-PROLOG.*
(this volume)

[8] Colmerauer A.
*An interesting natural language subset.*
Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1977.
[To appear in CACM].

[9] Colmerauer A, Kanoui H and van Caneghem M.
*Etude et realisation d'un systeme Prolog.*
Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1979.

[10] Dahl, V.
*Un systeme deductif d'interrogation de banques de donnẽes en Espagnol.*
Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1977.

[11] Damas L.
[Information about EMAS Prolog is available from its author].
Dept of Computer Science, University of Edinburgh.

[12] Darvas F, Futo I and Szeredi P.
Logic based program system for predicting drug interactions.
Int. J. of Biomedical Computing, 1977.

[13] DEC.
*DECsystem-10 System Reference Manual.*
Digital Equipment Corporation, Maynard, Mass, 1974.

[14] Markusz Z.
*Designing variants of flats.*
IFIP Conference, 1977.

[15] Mellish C.
*Minimal documentation of the PDP-11 Prolog system.*
Dept of Artificial Intelligence, Univ of Edinburgh, 1978.
[Informal note].

[16] Pereira L M, Pereira F and Warren D H D.
*User's Guide to DECsystem-10 Prolog.*
Dept of Artificial Intelligence, University of Edinburgh, 1978.

[17] Roberts G.M.
An implementation of Prolog.
Master's thesis, Dept of Computer Science, Univ of Waterloo, Canada, 1977.

[18] Robinson J A.
A machine-oriented logic based on the resolution principle.
*JACM* 12 (1): 227-234, December 1965.

[19] Roussel P.
*Prolog: Manuel de Reference et d'Utilisation.*
Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1975.

[20] Steele G L.
RABBIT: A compiler for SCHEME.
Master's thesis, MIT, May, 1978.
AI-TR-474.

[21] Szeredi P.
*Prolog – a very high level language based on predicate logic.*
2nd Hungarian Conference on Computer Science, Budapest, June, 1977.

[22] Warren D H D.
*Generating conditional plans and programs.*
AISB Summer Conference, Edinburgh, July, 1976.

[23] Warren D H D.
*Implementing Prolog – compiling predicate logic programs.*
Dept of Artificial Intelligence, Univ of Edinburgh, 1977.
Research Reports 39 & 40.

[24] Warren D H D.
*Logic programming and compiler writing.*
Dept of Artificial Intelligence, Univ of Edinburgh, 1977.
Research Report 44 [To appear in Software Practice and Experience].

[25] Warren D H D, Pereira L M and Pereira F.
*Prolog – the language and its implementation compared with Lisp.*
ACM Symposium on AI and Programming Languages, August, 1977.