PROLOG - THE LANGUAGE AND ITS IMPLEMENTATION COMPARED WITH LISP

David H D Warren
Department of Artificial Intelligence
University of Edinburgh
Scotland

Luis M Pereira
Fernando Pereira
Divisao de Informatica
Laboratorio Nacional de Engenharia Civil
Lisbon, Portugal

Abstract

Prolog is a simple but powerful programming language founded on symbolic logic. The basic computational mechanism is a pattern matching process ("unification") operating on general record structures ("terms" of logic). We briefly review the language and compare it especially with pure Lisp. The remainder of the paper discusses techniques for implementing Prolog efficiently; in particular we describe how to compile the patterns involved in the matching process. These techniques are as incorporated in our DECsystem-10 Prolog compiler (written in Prolog). The code it generates is comparable in speed with that produced by existing DEC10 Lisp compilers. We argue that pattern matching is a better method for expressing operations on structured data than conventional selectors and constructors - both for the user and for the implementor.

Introduction

Prolog is a very simple, but surprisingly powerful, programming language developed at the University of Marseille [Roussel 1975], as a practical tool for "logic programming" [Kowalski 1974] [Colmerauer 1975] [van Emden 1975]. From a user's point of view the major attraction of the language is ease of programming. Clear, readable, concise programs can be written quickly with few errors.

We have been concerned with implementing a Prolog system [Pereira 1977] [Warren 1977] specifically for the DECsystem-10 [DEC 1974]. Our implementation includes an interpreter and a compiler, both written in Prolog itself. The main aim of this paper is to describe some of the novel aspects of the work, especially the concept of compiling a "pattern matching" language such as Prolog. However, since Prolog is not widely known, we shall begin with a brief description of the language itself, drawing attention to the special features which make its implementation interesting and worthwhile. The discussion will cover the basic language, but not various built-in procedures for input-output hardware arithmetic, etc. Please note that we have not made (nor wanted to make) any original contribution to the basic language design.

The Language

Prolog has many parallels with Lisp. Both are interactive languages designed primarily for symbolic data processing. Both are founded on formal mathematical systems - Lisp on Church's lambda calculus, Prolog on a subset of classical logic. Like pure Lisp, the Prolog language does not (explicitly) incorporate the machine-oriented concepts of assignment and references (pointers). Furthermore, pure Lisp can be viewed as a specialisation of Prolog, where procedures are restricted to simple functions and data structures are restricted to lists. Let us therefore start by translating some elementary Lisp functions into Prolog:-

```
append[x;y]=
   [null[x] -> y;
    T -> cons[car[x];append[cdr[x];y]]]

nreverse[x]=
   [null[x] -> nil;
    T -> append[nreverse[cdr[x]];cons[car[x];
                                          nil]]]
```

These functions for list concatenation and (naive) list reversal are equivalent to the following two Prolog procedures:

```
append(nil,Y,Y).
append((A.B),Y,(A.B1)) :- append(B,Y,B1).

nreverse(nil,nil).
nreverse((A.B),Y):-
nreverse(B,Y0),append(Y0,(A.nil),Y).
```

Each procedure comprises a number of clauses corresponding to the branches of the conditional(s) in the Lisp function definition. The procedure name is called a predicate, and has an arity one greater than the corresponding Lisp function. The extra argument expresses the result of the function. The leftmost part of a clause is its head or procedure entry point, and displays a possible form of the arguments to the predicate. The remainder of the clause, its body, consists of a number (possibly zero) of goals or procedure calls, which impose conditions for the head to be true. If the body is empty we speak of a unit clause.

We see that the Prolog formulation does not require data selectors (car and cdr) or constructors (cons). Instead, the form or "pattern" of the procedure's input and output is displayed as explicit data structures. In this example these correspond to Lisp S-expressions. In general, Prolog data objects are called terms. A term is either a variable (distinguished by an initial capital letter), an atom (such as 'nil') or a compound term (such as '(A.B)'). A compound term comprises a functor of some arity N >= 1, with a sequence of N terms as arguments. For instance the functor of '(A.B)' is '.' of arity 2 and the arguments are A,B. We have written this term using an optional infix notation. The standard notation would be '.(A,B)'. One should think of a functor as a record type and the arguments of a term as fields of a record. An atom is treated as a functor of arity 0. The head and goals of a clause are considered to be terms, so a predicate is merely a functor which occurs in a particular context.

To make use of the Prolog procedure for reversing a list, one might execute (or solve) a goal such as:-

```
nreverse((1.2.3.nil),X)
```

Note that (1.2.3.nil) is merely a shorthand for (1.(2.(3.nil))). The effect of the procedure call will be to give the variable X a value which is the term:-

```
(3.2.1.nil)
```

Prolog differs from most programming languages in that there are two quite distinct ways to understand its semantics. The procedural semantics is the more conventional, and describes in the usual way the sequence of states passed through when executing a program. In addition a Prolog program can be understood as a set of descriptive statements about a problem. The declarative semantics which Prolog inherits from logic provides a formal basis for such a reading. Informally, one interprets terms as shorthand for natural language phrases by applying a uniform translation of each functor. e.g.:-

```
nil = "the empty list"
(A.B) = "the list whose first element is A
         and remaining elements are B"
nreverse(X,Y) = "the reverse of X is Y"
```

A clause 'P :- Q, R, S.' is interpreted as:-

"P if Q and R and S"

Each variable in a clause should be interpreted as some arbitrary object.

The declarative semantics simply defines (recursively) the set of terms which are asserted to be true according to a program. A term is true if it is the head of some clause instance and each of the goals (if any) of that clause instance is true, where an instance of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

Thus the only instance of the goal:-

```
nreverse((1.2.3.nil),X)
```

which is true is:-

```
nreverse((1.2.3.nil),(3.2.1.nil))
```

In this way the declarative semantics gives one some understanding of a Prolog program without looking into the details of how it is executed.

It is the declarative aspect of Prolog which is responsible for promoting clear, rapid, accurate programming. It allows a program to be broken down into small, independently meaningful units (clauses).

The procedural semantics describes the way a goal is executed. The object of the execution is to produce true instances of the goal. It is important to notice that the ordering of clauses in a program, and goals in a clause, which are irrelevant as far as the declarative semantics is concerned, constitute crucial control information for the procedural semantics.

To execute a goal, the system searches for the first clause whose head matches or unifies with the goal. The unification process [Robinson 1965] finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then activated by executing in turn, from left to right, each of the goals of its body (if any). If at any time the system fails to find a match for a goal, it backtracks, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

Let us now briefly look at how the goal:-

```
nreverse((1.2.3.nil),X)
```

is actually executed. The goal only matches the second of the two clauses for 'nreverse'. The body of the matching clause instance is:-

```
nreverse((2.3.nil),Y0), append(Y0,(1.nil),X)
```

The result of executing the first of these two goals is to instantiate Y0 to (3.2.nil) leaving the second goal as:-

```
append((3.2.nil),(1.nil),X)
```

This matches only the second clause for 'append', instantiating X to (3.X1) and producing a recursive procedure call:-

```
append((2.nil),(1.nil),X1)
```

(The name chosen for the new variable X1 is arbitrary). Eventually we hit the bottom of the recursion with the goal:-

```
append(nil,(1.nil),X2)
```

This now matches only the first clause for 'append', instantiating X2 to (1.nil) thus completing the final result:-

```
X = (3.2.1.nil)
```

Prolog owes its simplicity firstly to a generalisation of certain aspects of other programming languages, and secondly to omission of many other features which are no longer strictly essential. This generalisation gives Prolog a number of novel properties (compared in particular with Lisp). We shall briefly summarise these and then give two illustrative examples.

(1) General record structures take the place of Lisp's S-expressions. An unlimited number of different record types may be used. Records with any number of fields are possible, giving the equivalent of fixed bound arrays. There are no type restrictions on the fields of a record.

(2) Pattern matching replaces the use of selector and constructor functions for operating on structured data.

(3) Procedures may have multiple outputs as well as multiple inputs.

(4) The input and output arguments of a procedure do not have to be distinguished in advance, but may vary from one call to another. Procedures can thus be multi-purpose.

(5) Procedures may generate, through backtracking, a sequence of alternative results. This amounts to a high level form of iteration.

(6) Unification includes certain features which are not found in the simpler pattern matching provided by languages such as Microplanner. We sum this up in the "equation":-

```
   unification = pattern matching
               + the logical variable
```

The characteristics of the "logical" variable are as follows. An "incomplete" data structure (ie. containing free variables) may be returned as a procedure's output. The free variables can later be filled in by other procedures, giving the effect of implicit assignments to a data structure (cf. Lisp's rplaca, rplacd). Where necessary, free variables are automatically linked together by "invisible" references. As a result, values may have to be "dereferenced". This is also performed automatically by the system. Thus the programmer need not be concerned with the exact status of a variable - assigned or unassigned, bound to a reference or not. In particular, the occurences of a variable in a pattern do not need any prefixes to indicate the status of the variable at that point in the pattern matching process (contrast Microplanner etc.). In short, the logical variable incorporates much of the power of assignment and references in other languages. This is reminiscent of the way most uses of goto can be obviated in a language with "well-structured" control primitives.

(7) Program and data are identical in form. Clauses can usefully be employed for expressing data.

(8) As we have already seen, there is a natural declarative semantics in addition to the usual procedural semantics.

(9) The (procedural) semantics of a syntactically correct program is totally defined. It is impossible for an error condition to arise or for an undefined operation to be performed. This is in contrast to most programming languages including pure Lisp (cf. cars and cdrs of atoms, unbound variables). A totally defined semantics ensures that programming errors do not result in bizarre program behaviour or incomprehensible error messages.

The following example illustrates the identity of program and data in Prolog, and shows the language's potential as a natural medium for database interrogation. A "database" of unit clauses provides information on the populations (in millions) and areas (in thousands of square miles) of various countries. A procedure 'density' supplies "virtual data" on population densities (per square mile):-

```
pop(china,825).   area(china,3380).
pop(india,586).   area(india,1139).
pop(ussr,252).    area(ussr, 8708).
pop(usa, 212).    area(usa,  3609).
:
```

```
density(C,D):-
   pop(C,P), area(C,A), D is (P*1000)/A.
```

The following clause represents a database query to find countries of similar population density (differing by less than 5%):-

```
ans(C1,D1,C2,D2) :-
   density(C1,D1), density(C2,D2),
   D1>D2, 20*D1 <21*D2.
```

Executing the goal 'ans(C1,D1,C2,D2)' will supply, through backtracking, the sequence of solutions required. Notice how the density procedure generates multiple results. The two calls to 'density' have exactly the same effect as nested iterations in a conventional language. In implementation terms, a sequence of different values is assigned to the variables C1,D1,C2,D2; cf. the following Algol-style procedure:-

```
for C1 from 1 to N do
   int D1 := pop[C1]*1000/area[C1];
   for C2 from 1 to N do
      int D2 := pop[C2]*1000/area[C2];
      if  D1>D2 and 20*D1 <21*D2
      then output(country[C1],D1,
                  country[C2],D2);
   repeat
repeat
```

The second example displays many of the characteristics which make Prolog an agreeable language for compiler writing (as applied in the case of our own Prolog compiler). The task is to generate a list of serial numbers for the items of a given list, the members of which are to be numbered in alphabetical order eg.

```
(p.r.o.l.o.g.nil) -> (4.5.3.2.3.1.nil)
```

As with many Prolog programs, the key to arriving at the required algorithm is to first conceive a procedure which checks whether a proposed list of serial numbers is a correct solution. This can be done by pairing up the items of the input list with their proposed serial numbers as an "association list", arranging these pairs in alphabetical order, and then finally checking whether the serial numbers are in the correct consecutive order. i.e.-

```
serialise(L,R) :-
   pairlists(L,R,A),
   arrange(A,T),
   numbered(T,1,N).
```

The pairing is done by a procedure very similar to the pairlis function of the Lisp 1.5 manual, but with the pairs represented as terms 'pair(X,Y)':-

```
pairlists((X.L),(Y.R),(pair(X,Y).A)) :-
   pairlists(L,R,A).
pairlists(nil,nil,nil).
```

The arrangement in alphabetical order and checking of the numbers could be done using only lists, however it is much more convenient to use binary trees. We represent a tree as a term of the form 'void' ("the void tree") or 'tree(T1,X,T2)' (" a tree with X at the root and subtrees T1 and T2").

```
arrange((X.L),tree(T1,X,T2)) :-
     partition(L,X,L1,L2),
     arrange(L1,T1),
     arrange(L2,T2).
arrange(nil,void).
```

```
partition((X.L),X,L1,L2) :- partition(L,X,L1,L2).
partition((X.L),Y,(X.L1),L2) :-
   before(X,Y), partition(L,Y,L1,L2).
partition((X.L),Y,L1,(X.L2)) :-
   before(Y,X), partition(L,Y,L1,L2).
partition(nil,Y,nil,nil).

before(pair(X1,Y1),pair(X2,Y2)) :- X1 < X2.

numbered(tree(T1,pair(X,N1),T2),N0,N) :-
   numbered(T1,N0,N1),
   N2 is N1+1,
   numbered(T2,N2,N).
numbered(void,N,N).
```

This procedure for verifying a solution illustrates the advantages of having more general record structures and of manipulating them by pattern matching. Notice also how the partition procedure returns two outputs.

Now it happens that the 'serialise' procedure is multi-purpose - it can be used not only for verification, but will also actually construct the required list of serial numbers if the initial goal does not provide one. This remarkable property owes something to judicious design, but is made possible by the characteristics of the "logical" variable. Let us consider what happens when a goal such as:-

```
serialise((d.a.t.a.nil),R)
```

is executed. The call to 'pairlists' returns two outputs (thus 'pairlists' can also serve more than one purpose):-

```
R = (X1.X2.X3.X4.nil)
A = (pair(d,X1).pair(a,X2).pair(t,X3).
     pair(a,X4).nil)
```

Both these data structures are incomplete. i.e. In implementation terms, they contain references to 4 empty cells created by 'pairlists'. The list R is the partially completed output for 'serialise'. Next the call to 'arrange' generates a tree of the form:-

```
              *  <d,X1>
             / \
            /   \
  <a,X2> *       *  <t,X3>
```

Notice that variable X4 has become bound to X2 (it could equally well be the other way round). The implementation has created a reference to X2's cell and assigned it to X4's. Finally the call to 'numbered' completes the tree returned by 'arrange' and causes X2,X1,X3 to be bound to 1,2,3. This has the effect of assignments to a data structure. One should also note that the finally completed list returned by 'serialise' still contains a reference (i.e. the field corresponding to X4 will require an extra step of indirection when accessed).

It is difficult to see how this algorithm could be simulated in pure Lisp. Use of rplaca and replacd would almost certainly be called for, resulting in a less transparent program. Notice how the Prolog programmer is spared all the intricate implementation details.

Implementation

The first experimental interpreter for Prolog was written in Algol-W by Philippe Roussel [1972]. This work led to better techniques for implementing the language, incorporated in the more widely used Marseille interpreter written in Fortran by Battani and Meloni [1973]. More recently, Maurice Bruynooghe [1976] has implemented a Prolog interpreter in Pascal. He gives a good introduction to the fundamentals of Prolog implementation and describes a space saving technique using a "heap". Other Prolog interpreters have been implemented at the University of Waterloo, Canada (for IBM 370) and at Budapest (in CDL for ICL 1900).

Our implementation is based on a compiler from Prolog to DECsystem-10 assembly language. Like Bruynooghe, we use many of the same techniques as were developed at Marseille. The main innovations are:-

(1) the concepts for compiling Prolog into a machine language;

(2) indexing of clauses within a procedure;

(3) some particular measures to economise on space required during execution.

The most important innovation is compilation. Recall that a Prolog computation is essentially just a sequence of unifications or "pattern matching" operations. Each unification involves matching two terms or "patterns", one a goal or "procedure call", the other a clause head or "procedure entry point". The principal effect of compilation is to translate the head of each clause into instructions which will do the work of matching against any goal pattern. Of the two patterns involved in a matching, we choose to compile the clause head because it is initially uninstantiated, unlike the goal. Also the uninstantiated form of the goal which appears in the source program typically has little structure that can be compiled.

In any clause head, the first occurrence of a variable can be translated into a straightforward assignment, since the variable will be initially uninstantiated. If the variable is at the outermost level of the pattern, e.g.

```
append((A.B),Y,(A.B1)) :- append(B,Y,B1).
               *
```

it will be assigned an argument of the procedure call. Otherwise the assignment will have the effect of selecting a component of a data structure, e.g. in the case of:-

```
append((A.B),Y,(A.B1)) :- append(B,Y,B1).
          *
```

Now in practice most of the symbols in a clause head are first occurrences of variables, so much of the pattern just translates into conventional assignments. The code for the rarer case of a subsequent occurrence of a variable, e.g.

```
append(nil,Y,Y).
               *
```

is more complex, and may involve calling a recursive subroutine to do the matching. If a variable has just a single occurrence in a clause, no executable code need be generated for it.

The code generated for a compound subterm (or subpattern), e.g.

```
   append((A.B),Y,(A.B1)) :- append(B,Y,B1).
                   ******
```

has to distinguish between two cases. If the subterm matches against a variable, a new data structure has to be constructed (cf. cons) and assigned to the variable. The variable assigned to will usually have to be remembered on a push-down list (called the "trail") to allow back-tracking to "undo" the assignment later. This first case is handled by an out-of-line subroutine.

The other case concerns matching against a non-variable. This is performed essentially by in-line code. It comprises a test for matching functors (record types), followed by the compiled form of each of the subterms of the compound term. This code will be responsible for accessing sub-components of the matching data structure (cf. car and cdr).

In describing the various unification steps, we have so far passed over the creation of references and their subsequent dereferencing. If a variable matches against another variable, a reference to one of the variable's cells is created and assigned to the other. Each unification step has to be prepared to dereference an arbitrarily long chain of these references. In practice, however, constructed data structures usually contain no references, and chains of references are even rarer. Our implementation is such that a single test suffices to determine the required action for the commonest cases of most unification steps.

A useful function performed by our compiler, and not found in previous implementations of Prolog, is to index the different clauses in a procedure, giving the effect of a "switch" or "computed goto". If the clauses would conventionally be considered as data, the effect is to store this data in an array or "hash table". For simplicity of implementation, the indexing is only on the form (i.e. principal functor) of the first subterm in the head of each clause, but this is perfectly adequate for most actual Prolog programs. For example the clauses for 'pop' and 'area' in the populations example are indexed by country (the first argument) but not by the numeric value which is the second argument. Effectively, the clauses are compiled into two arrays of numbers with "non-numeric" subscripts.

Our implementation uses the same "structure-sharing" technique for the internal representation of constructed data as was introduced by the Marseille interpreter. The technique is a novel and elegant alternative to the "literal" representation based on linked records in "heap" storage which is conventional for other languages (including Lisp). The basis of the technique is to represent a compound data object by a pair of pointers called a "molecule". One pointer indicates a "skeleton" structure (i.e. a compound term) occurring in the source program, the other points to a vector of cells called a "frame". The frame contains the values of variables occurring in the skeleton. This representation facilitates very rapid creation of structured data at the expense of somewhat slower access to its components. A further advantage is greater compactness in most cases.

Although structure-sharing entails extra work to access the components of a data structure, the overhead is small on a machine with good "indirect addressing" facilities. The architecture of the DEC10 is particularly favourable, as a variable in a skeleton can be nicely represented by a DEC10 "address word". This specifies the address of the variable's cell as an offset relative to the contents of an index register. Any DEC10 instruction can obtain its operand indirectly by referring to an address word. This means that, once the frame address for a molecule has been loaded into an index register, each of the fields of the structure-shared record can be accessed in just one instruction. The impact on overall performance is substantial.

The main drawback of the Marseille interpreter is its tendency to require unacceptable amounts of working storage. The source of the problem is that Prolog views non-determinate computation as the rule rather than the exception. A procedure cannot "return" in the conventional way until after it has generated all of its alternative results (i.e. until backtracking occurs). When just one of the results has been completed (i.e. the end of a clause is reached), the implementation can't "pop" the stack in the usual way. Instead the effect is to immediately invoke the caller's "continuation". Our space economy measures rely, like Bruynooghe's technique, on the fact that in practice most Prolog procedures produce just a single result. The major step is to classify Prolog variables into "locals" and "globals". This is performed by the compiler and need be of no concern to user. Storage for the two types is allocated from different areas, the local and global stacks, analagous to the "stack" and "heap" of Algol-68. Now when the end of a clause is reached, and provided the procedure can generate no further results, the local storage for the procedure is recovered automatically by a stack mechanism, as for a conventional language. No garbage collector is needed for this process, unlike Bruynooghe's method. Note that clause indexing helps the system to detect when a procedure can produce no more results.

Most Prolog procedures are not in practice used in a multi-purpose way. For example the 'append' procedure might only be used to concatenate two given lists, and not, say, to generate pairs of lists which when concatenated give a specified third list. In our implementation, the user can notify the system of such restrictions on the usage of procedures through optional "mode declarations". These enable a higher proportion of variables to be placed in the more desirable "local" category, and also help to improve the compactness of the compiled code.

In addition to these measures, our system can also recover storage from the global stack by garbage collection, cf. Algol-68's heap. The garbage collector used has to be quite intricate even by normal standards. After what is in principle a conventional "trace and mark", space is recovered by compacting global storage still in use to the bottom of the stack. This involves "remapping" all addresses pointing to the global stack.

It is important to notice that a garbage collector is not essential for our system (unlike for example Lisp implementations). If the user restricts himself to tasks smaller than a certain size, the garbage collector need never be used. This is because a general stack mechanism recovers all storage on backtracking, or when each task is complete, as for the Marseille interpreter.

We may finally remark that our implementation automatically adjusts the sizes of the different storage areas during execution.

Performance Comparisons

Some detailed performance comparisons have been made on a DEC10 (K1 processor) of compiled Prolog with interpreted Prolog (Marseille), and also with compiled Lisp (Stanford, with NOUUO option).

(a) Speed

There is a 15 to 20-fold improvement over the Marseille interpreter. Simple functions over lists (e.g. the naive reverse example) run quite uniformly at about 50% to 70% of the Lisp speed. Note that the compiler treats lists no differently from other terms. Simple functions over more general data structures can equal or better the speed of the corresponding pure Lisp function operating on data encoded as lists. For example, the following differentiation procedure:-

```
d(U+V,X,DU+DV) :- d(U,X,DU),d(V,X,DV).
d(U-V,X,DU-DV) :- d(U,X,DU),d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- d(U,X,DU),d(V,X,DV).
d(U/V,X,(DU*V-U*DV)/V^2) :- d(U,X,DU),d(V,X,DV).
d(U^N,X,DU*N*U^N1) :-
    integer(N), N1 is N-1,d(U,X,DU).
d(X,X,1).
d(C,X,0) :- atomic(C), C ≠ X.
```

runs (depending on the data) 1.1 to 2.6 times faster than the equivalent Lisp DERIV function given on p. 167 of Weissman's [1967] Lisp primer.

(b) Space

The saving on working storage relative to Marseille depends greatly on the degree of determinacy of the program. At worst it is 2-times better due simply to tighter packing of data into the machine word. Figures for the compiler itself indicate roughly a 10-fold improvement. Recall that the compiler is a Prolog program, originally "bootstrapped" using the Marseille interpreter. It now rarely requires more than 5K words total for the trail and two stacks.

The compiled code itself is relatively compact at about 2 words per source symbol.

It is difficult to make meaningful space comparisons with Lisp and we have not so far attempted to do so.

(c) Comments

The tests show that Prolog speed compares quite well with pure Lisp, especially where a wider range of record types is really called for. Of course such a comparison only evaluates a limited part of Prolog and can't be entirely fair since Lisp is specialised to just this area. Moreover, Lisp systems do not provide complete security against program error - car and cdr are allowed to apply indiscriminately to any object. As a result no run-time checks are needed, and the fundamental selectors reduce to very simple machine-oriented operations - effectively hardware instructions on the DEC10. One might therefore have expected Lisp to be considerably faster than Prolog. There are two main factors acting in favour of the Prolog implementation.

Firstly, and perhaps surprisingly, there are good reasons to expect pattern matching to promote better implementation than conventional selectors and constructors. Productive computation is easily integrated with procedure call so minimising the overheads of argument passing, a process which is usually "red tape" in other languages, including Lisp. In particular, no location needs to be set up for an argument to a procedure if only its components are to be referred to. These are selected once and for all by pattern matching, in a single efficient process without having to reload index registers for each component. No optimisation is necessary to avoid duplication of work brought about in Lisp, when, for example, car of an object is repeatedly referred to, or caddr and cdddr are applied to the same object. These points apply a fortiori for a language such as Pop-2 with multiple data types requiring run-time type checking, since the type of an object is only checked once in pattern matching. Finally, pattern directed invocation of the different clauses of a procedure enables and encourages the implementation to incorporate computed gotos automatically where appropriate.

The second factor favouring Prolog is structure-sharing. Ironically, this technique was first devised by Boyer and Moore [1972] as a means of saving space. However it is even more important for its contribution to Prolog's speed. Essentially it enables a "cons" to be effected faster than in Lisp. Partly this is because the nature of Prolog permits storage to be allocated in stacks, so there are none of the costs associated with allocating records individually from a "heap". Neither need there be any garbage collection overheads, since global storage can normally be recovered by the stack mechanism. Both these advantages have even greater force if one compares with a language allowing more than one record size. An additional feature of the stack regime is the avoidance of random memory accesses, enabling better exploitation of a paged machine. The other main reason for the speed of the structure-sharing "cons" is the avoidance of the copying of information which occurs when a conventional cons initialises a new list cell. The greater the number of symbols in a "skeleton" term, the greater is this saving. Essentially, structure-sharing replaces copying in "cons" by extra indirection in "car" and "cdr". As we have already seen, the extra indirection costs very little on suitable machines such as the DEC10, with its "effective address mechanism".

Conclusion

Pattern matching should not be considered an "exotic extra" when designing a programming language. It is the preferable method for specifying operations on structured data, both from the user's and the implementor's point of view. This is especially so where more than one record type is allowed. Hoare [1975] makes a similar case for a more limited form of pattern-matching in the context of an Algol-like language.

For applications requiring an easy-to-use and transparent language for "symbol processing", Prolog seems to offer significant advantages over Lisp. Even ignoring Lisp's unfortunate syntax and variable binding mechanism, a major barrier to its readability is the size and complexity (degree of nesting) of typical function definitions. Prolog allows a program to be formulated in smaller units, each having a natural declarative reading. In addition it gives the programmer generalised record structures with an elegant mechanism for manipulating them. The pure Lisp view of computation as simple function evaluation is too restrictive for typical applications, so extensive use is normally made of lower-level extensions to the language (<u>prog</u>, <u>rplaca</u> etc.) resulting in less transparent programs. Prolog allows programs with similar behaviour to be written without having to resort to machine- or implementation-oriented concepts. For example, our compiler is written almost entirely in "pure" Prolog (i.e. clauses with a valid declarative interpretation). Finally, our work shows that the use of Prolog as opposed to (pure) Lisp need involve no great loss of efficiency, if indeed any.

Acknowledgements

We are indebted to members of the Groupe d'Intelligence Artificielle, Marseille, for developing both the Prolog language and the fundamental structure-sharing technique. The work was supported by a British Science Research Council Grant B/RG 9972 and by L.N.E.C., Lisbon.

References

Battani G and Meloni H [1973]
Interpreteur du langage de programmation Prolog.
Groupe d'Intelligence Artificielle, Marseille-Luminy, 1973

Boyer R S and Moore J S [1972]
The sharing of structure in theorem proving programs.
Machine Intelligence 7 (ed. Meltzer & Michie), Edinburgh U. Press, 1972.

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

Colmerauer A [1975]
Les grammaires de metamorphase.
Groupe d'Intelligence Artificielle, Marseille, Marseille-Luminy, Nov 1975.

DEC [1974]
DECsystem10 System Ref Manual (3rd edition)
Digital Equipment Corporation, Maynard, Mass. Aug 1974.

van Emden M H [1975]
Programming with resolution logic.
Report CS-75-30, Dept. of Computer Science, University of Waterloo, Canada. Nov 1975.

Hoare C A R [1973]
Recursive data structures.
Stanford AI Memo 223, Calif. Oct 1973.

Kowalski R A [1974]
Logic for problem solving.
DCL Memo 75, Dept of AI, Edinburgh. Mar 1974.

McCarthy J et al. [1962]
LISP 1.5 Programmer's Manual.
MIT Press, MIT, Cambridge, Mass. Aug 1962.

Pereira L M [1977]
User's Guide to DECsystem-10 Prolog.
Forthcoming publication, Divisao de Informatica, Lab. Nac. de Engenharia Civil, Lisbon. 1977.

Robinson J A [1965]
A machine-oriented logic based on the resolution principle.
JACM vol 12, pp. 23-44. 1965.

Roussel P [1972]
Definition et traitement de l'egalite formelle en demonstration automatique.
These 3me. cycle, UER de Luminy, Marseille. 1972.

Roussel P [1975]
Prolog: Manual de reference et d'utilisation.
Groupe d'Intelligence Artificielle, Marseille-Luminy. Sep 1975.

Warren D H D [1977]
Implementing Prolog - compiling predicate logic programs.
Forthcoming report, Dept of AI, Edinburgh. 1977.

Weissman C [1976]
Lisp 1.5 Primer.
Dickenson Publishing Co. 1967.