Research Report

IMPLEMENTING PROLOG

- compiling predicate logic programs

Volume 1

by

David H D Warren

D.A.I. Research Report No. 39

Department of Artificial Intelligence
University of Edinburgh

IMPLEMENTING PROLOG

- compiling predicate logic programs

Volume 1

by

David H D Warren

D.A.I. Research Report No. 39

May 1977

1.0 CONTENTS

2.0 ABSTRACT

Prolog is a simple but powerful programming language founded on symbolic logic. It encourages rapid, error-free programming and clear, readable, concise programs. The basic computational mechanism is a pattern matching process ("unification") operating on general record structures ("terms" of logic). This report describes techniques for implementing Prolog efficiently. In particular we show how to compile the patterns involved in the matching process into instructions of a low-level language. Our implementation is actually a compiler (written in Prolog) from Prolog to DECsystem-10 assembly language, but the principles involved are explained more abstractly in terms of a "Prolog Machine". The code generated is comparable in speed with that produced by existing DEC10 Lisp compilers. Comparison is possible since pure Lisp can be viewed as a (rather restricted) subset of Prolog. We argue that structured data objects, such as lists and trees, can be manipulated by pattern matching using a "structure sharing" representation as efficiently as by conventional selector and constructor functions operating on linked records in "heap" storage. Moreover the pattern matching formulation actually helps the implementor to produce a better implementation.

Keywords

Prolog, programming, logic, implementation, compiler, data structures, matching, unification.

3.0 INTRODUCTION

This report describes techniques for efficiently implementing the programming language Prolog. It is written mainly for those having some familiarity with Prolog. For the benefit of a wider readership, we begin by attempting to answer briefly the questions "Why implement yet another programming language?", "What is so different about Prolog?". A precise definition of the basic Prolog language is given in Section /4./. The sample programs listed in Appendix /5./ and referred to in Section /10.1/ may be useful.

The second part of this introduction summarises the history and nature of Prolog implementation.

3.1 Why

Prolog is a simple but 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 one of Prolog's main attractions is ease of programming. Clear, readable, concise programs can be written quickly with few errors. Prolog is especially suited to "symbol processing" applications such as natural language systems [Colmerauer 1975] [Dahl & Sambuc 1976], compiler writing [Colmerauer 1975] [Warren 1977], algebraic manipulation [Bergman & Kanoui 1975] [Bundy et al. 1976], and the automatic generation of plans and programs [Warren 1974] [Warren 1976].

Data structures in Prolog are general trees, constructed from records of various types. An unlimited number of different types may be used and they do not have to be separately declared. 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.

The conventional way of manipulating structured data is to apply previously defined constructor and selector functions (cf. Algol-68, Lisp, Pop-2). These operations are expressed more graphically in Prolog by a form of "pattern matching", provided through a process called "unification". There is a similarity to the treatment of "recursive data structures" advocated by Hoare [1973]. Unification can also be seen as a generalisation of the pattern matching provided in languages such as Microplanner [Sussman & Winograd 1970] and its successors.

For the user, Prolog is an exceptionally simple language. Almost all the essential machinery he needs is inherent in the unification process. So, in fact, a Prolog computation consists of little more than a sequence of pattern-directed procedure invocations. Since the procedure call plays such a vital part, it is necessarily a more flexible mechanism than in other languages. Firstly, when a procedure "returns" it can send back more than one output, just as (in the conventional way) it may have received more than one input. Moreover, which arguments of a procedure are inputs and which will be output slots doesn′t have to be determined in advance. It may vary from one call to another. This property allows procedures to be "multi-purpose". An additional feature is that a procedure may "return" several times sending back alternative results. Such procedures are called "non-determinate" or "multiple-result". The process of reactivating a procedure which has already returned one result is known as "backtracking". Backtracking provides a high-level equivalent of iterative loops in a conventional language.

There is no distinction in Prolog between procedures and what would conventionally be regarded as tables or files of data. Program and data are accessed in the same way and may be mixed together. Thus in general a Prolog procedure comprises a mixture of explicit facts and rules for computing further "virtual" data. This and other characteristics give Prolog interesting potential as a query language for a relational database (cf. [van Emden 1976] and Zloof's "Query by Example" [1974]).

Earlier we compared unification with Microplanner-style pattern matching. There is an important difference which we summarise in the "equation":-

unification = pattern matching + the logical variable

The distinction lies in the special nature and more flexible behaviour of the variable in Prolog, referred to as the "logical" variable. Briefly, each use of a Prolog variable stands for a particular, unchangeable data item. However the actual value need not be specified immediately, and may remain unspecified for as long as is required. The computational behaviour is such that the programmer need not be concerned whether or not the variable has been given a value at a particular point in the computation. This behaviour is entirely a consequence of constraints arising from logic, the language on which Prolog is founded.

By contrast, the variable in most other programming languages is a name for a machine storage location, and the way it functions can only be understood in this light. The "assigning" of values to variables is the programmer's responsibility and in many situations he must guarantee that the variable is not left unassigned. This applies equally to the variables used in the Planner family of pattern matching languages. There, each occurrence of a variable in a pattern has to be given a prefix to indicate the status (assigned or unassigned) of the variable at that point. The programmer must understand details of the implementation and sequencing of the pattern matching process, whereas Prolog's unification is a "black box" as far as the user is concerned.

There are some other programming languages where the variable does not have to be thought of as a machine location, most notably pure Lisp. In pure Lisp as in Prolog, the behaviour of the variable is governed by an underlying formal mathematical system, in this case Church's lambda calculus. As a consequence, the machine-oriented concepts of assignment and references (pointers) are not an (explicit) part of either language. These are just some of a number of close parallels between Prolog and pure Lisp.

Now it is well known that pure Lisp is too weak for many purposes. Various extensions to the language are a practical necessity. In particular the operations <u>rplaca</u> and <u>rplacd</u> are provided to allow components of a data structure to be overwritten. This immediately introduces into the language the concepts of assignment and reference which were previously avoided.

No similar extension is provided in Prolog, nor is it needed owing to the special properties of the logical variable. The main point is that a Prolog procedure may return as output an "incomplete" data structure containing variables whose values have not yet been specified. These "free" variables can subsequently be "filled in" by other procedures. This is achieved in the course of the normal matching process, but has much the same effect as explicit assignments to the fields of a data structure. A necessary corollary is that when two variables are matched together, they become linked as one. In implementation terms, a reference to one variable is assigned to the cell of the other. These references are completely invisible to the user; all necessary dereferencing is handled automatically behind the scenes.

In general, the logical variable provides much of the power of assignment and references, but in a higher-level, easier-to-understand framework. This is reminiscent of the way most uses of goto can be avoided in a language with "well-structured" control primitives.

There is an important relationship between co-routining and the logical variable. Co-routining is the ability to suspend the execution of one procedure and communicate a partial result to another. Although not provided as such in Prolog, it is easily programmed without resort to low-level concepts, because the logical variable provides the means for partial results and suspended processes to be treated as regular data structures. The main difficulty is to determine when to co-routine, but this problem is common to languages with explicit co-routining primitives.

So far we have previewed Prolog as a "set of features". The features are significant primarily because they mesh together well to make the task of programming less laborious. They can be looked on as a useful selection and generalisation of elements from other programming languages. However Prolog actually arose by a different route. It has a unique and more fundamental property which largely determines the nature of the other features. This property, that a Prolog program can be interpreted declaratively as well as procedurally, is the real reason why Prolog is an easier language to use.

For most programming languages, a program is simply a description of a process. The only way to understand the program and see whether it is correct is to <u>run</u> it - either on a machine with real data, or symbolically in the mind's eye. Prolog programs can also be understood this way, and indeed this view is vital when considering efficiency. We say that Prolog, like other languages, has a <u>procedural</u> semantics, one which determines the sequence of states passed through when executing a program.

However, there is another way of looking at a Prolog program which does not involve any notion of time. Here the program is interpreted declaratively, as a set of descriptive statements about a problem domain. From this standpoint, the "lines" of the program are nothing more than a convenient shorthand for ordinary natural language sentences. Each line is a statement which makes sense in isolation, and which is about objects (concrete or abstract) that are separate from the program or machine itself. The program is correct if each statement is true.

The natural declarative reading is possible, basically because the procedural semantics of Prolog is governed by an additional declarative semantics, inherited straight from logic. The statements which make up a Prolog program are in fact actually statements of logic. The declarative semantics defines what facts can be inferred from these statements. It lays down the law as to what is a correct result of executing a Prolog program. How the program is executed is the province of the procedural semantics.

The declarative semantics helps one to understand a program in the same kind of way as the law of conservation of energy helps one to understand a mechanical system without looking in detail at the forces involved. Analogously, the Prolog programmer can initially ignore procedural details and concentrate on the (declarative) essentials of the algorithm. Having the program broken down into small independently meaningful units makes it much easier to understand. This inherent modularity also reduces the interfacing problems when several programmers are working on a project. Bugs are less likely, perhaps because it is difficult to make a "logical error" in a program when its logic is actually expressed in logic!

Of course there will always be errors due to typing mistakes, oversights or plain muddled thinking. Such errors are, however, relatively harmless because of one other very important property of (basic) Prolog - that it has a totally defined (procedural) semantics. This means that it is impossible for a syntactically correct program to perform (or even attempt to perform) an illegal or undefined operation. This is in contrast to most other programming languages (cf. array indices out of bounds in Fortran, or car of an atom in

Lisp). An error in a Prolog program will never cause bizarre behaviour. Nor will the program be halted prematurely with an error message indicating that an illegal condition has arisen.

3.2 What

The first implementation of Prolog was an interpreter written in Algol-W by Philippe Roussel [1972]. This work led to better techniques for implementing the language, which were realised in a second interpreter, written in Fortran by Battani and Meloni [1973]. A useful account in English of this implementation is given by Lichtman [1975]. A notable feature of the design is the novel and elegant "structure-sharing" technique [Boyer, Moore 1972] for representing structured data inside the machine. The basis of the technique is to represent a compound data object by a pair of pointers. One pointer indicates a "skeleton" structure occurring in the source program, the other points to a vector of value cells for variables occurring in the skeleton. The representation enables structured data to be created and discarded very rapidly, in comparison with the conventional "literal" representation based on linked records in "heap" storage. A further advantage is greater compactness in most cases.

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).

The main subject of this report is a Prolog system written specifically for the DECsystem-10 [DEC 1974] by the author, in collaboration with Luis Pereira and Fernando Pereira of the National Civil Engineering Laboratory, Lisbon. The system includes a compiler from Prolog into DEC10 assembly language and a conversational Prolog interpreter. It uses the same fundamental design, including the "structure-sharing" technique, as was developed for the second Marseille interpreter. However the implementation is considerably faster, owing to compilation, and also because it was possible to capitalise on the elegant DEC10 architecture which is particularly favourable to the structure-sharing technique.

A variable in a "skeleton" structure 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 appropriate index register has been loaded, each of the fields of a structure-shared record can be accessed in just one instruction.

It was in fact the possibilities of compilation and the DEC10 which originally inspired the writing of a new system. (A preliminary version which compiled into BCPL was abandoned at an early stage since it was found impossible to fully exploit the potential of the DEC10.) The compiled code shows a 15 to 20-fold speed improvement over the Marseille interpreter. It is quite compact at about 2 words per source symbol. The compiler itself is written in Prolog and was "bootstrapped" using the Marseille interpreter. The new interpreter is also largely implemented in Prolog.

Much of the material in this report will be a description of techniques developed by others (although nowhere fully documented). The main innovations are:-

(1) the concept of compiling Prolog,

(2) certain measures to economise on space required during execution,

(3) improved indexing of clauses.

The most important innovation is compilation. Now 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 term is a "goal" (or "procedure call") and is instantiated. The other is the uninstantiated "head" of a clause (or "procedure entry point"). The principal effect of compilation is to translate the head of each clause into low-level instructions which will do the work of matching against any goal pattern. Thus there remains little need for a general matching procedure. Specialised code has been generated to replace most uses of it.

Much of the code just amounts to simple tests and assignments. In particular, all that has to be done for the first occurrence of a variable is to assign the matching term to the variable's cell. Thus this very common case is also very fast.

The code generated for a compound subterm (or sub-pattern) splits into two cases. If the matching term is a variable, a new data structure is constructed (using structure-sharing) and assigned to the variable. The code for the other case is responsible for accessing subcomponents of the matching term, ie. it does the work of selectors in a conventional language.

The main drawback of the Marseille interpreter is its unacceptable appetite for working storage. Like Bruynooghe, we have devoted considerable attention to this problem. Our solution 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. When execution of a procedure has been completed "determinately" (ie. there are no further multiple results to be produced), local storage is recovered automatically by a stack mechanism, as for a conventional language. No garbage collector is needed for this process.

The space saving achieved through this process can be improved if the user supplies optional pragmatic information via an innovation known as "mode decalarations". A mode declaration declares a restriction on the use of a procedure, ie. one or more arguments are declared to be always "input" (a non-variable) or always "output" (a variable). Thus the user is forgoing some of the flexibility of Prolog's "multi-purpose" procedures. This enables the system to place a higher proportion of variables in the more desirable "local" category and also helps 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 intrincate 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. If the user restricts himself to small tasks the garbage collector need never be used. This is because a stack mechanism recovers all storage automatically on backtracking, or when the overall task is complete, as for the Marseille interpreter. An additional point of practical importance is that our implementation automatically adjusts the sizes of the different storage areas during execution (remapping addresses as necessary).

The combined effect of these space saving measures is a substantial reduction in run-time storage needed for programs which are totally determinate (eg. the compiler itself) or partly determinate (most Prolog programs in practice). A 10-fold improvement over the Marseille interpreter would seem to be not unusual, although this depends very much on the actual program. (Even in the worst case of a totally non-determinate program, there is still a 2-fold improvement due simply to a better packing of information into the DEC10 word.)

In the Marseille interpreter, the clauses which make up both program and data are only indexed by the predicate (ie. procedure or relation name) to which they relate. Our compiler indexes clauses both by predicate and by the form of the first argument to this predicate. This is tantamount, for a procedure, to case selection by a fast "switch" (or computed goto). For data, it amounts to storing information about a relation in an array (or hash table).

Our description of Prolog implementation will take the form of a definition of a "Prolog machine" (PLM). The aim is to present, in as general a way as possible, the essential features of our DEC10 implementation, especially compilation. Although the structure of the PLM is directed specifically to the needs of Prolog, the result is a comparatively low-level machine with an architecture of a quite conventional form. It operates on data items of fixed sizes, which may be stored in special registers, areas of consecutively addressed locations, and "push-down" lists. The machine has a repertoire of instructions, each taking a small fixed number of arguments of definite size. In most cases, the processing of one instruction involves only a small and bounded amount of computation.

The Prolog machine has of course been designed primarily with the DEC10 in mind. As we have previously mentioned, DEC10's "effective address" mechanism greatly promotes the structure-sharing technique. However it should not be difficult to implement the design on any conventional computer, although the result might not be quite so efficient. More exciting perhaps would be the possibility of realising the machine in microprogram or even hardware.

In our DEC10 implementation, the effect of each Prolog machine instruction is achieved partly by in-line code and partly by calls to out-of-line subroutines. The optimal mixture is a tactical decision which has varied considerably during the course of implementation. The efficient "indirect addressing" and subroutine call of the DEC10 mean that operations can be performed out-of-line with very little overhead.

At present the Prolog compiler compiles directly into DEC10 assembler. Since the compiler is itself written in Prolog, it could easily be adapted to generate "Prolog machine code" as such. This code could be interpreted by an autonomous program written in almost any programming language. Alternatively it should not be difficult to produce a version of the compiler which translates into the assembly language of some other machine. The compiler itself is not described here (see [Warren 1977] for a general discussion of compiler writing in Prolog). However the function it performs should be clear from the relationship between Prolog machine instructions and Prolog source programs documented in Section /6.9/ and Appendix /2./.

Note: This report does not attempt to describe the implementation of the "evaluable predicates" etc. which are essential to a usable interactive system. These provide, among other things, built-in arithmetic, input-output, file handling, state saving, internal "database", and meta-logical facilities. It is an unfortunate fact that the major labour involved in implementing a Prolog system is providing such "trimmings".

4.0 THE PROLOG LANGUAGE

The basic Prolog language is best considered as being made up of two parts. On the one hand, a Prolog program consists of a set of logical statements, of the form known as Horn clauses. Clauses are just a simple normal form, (classically) equivalent to general logical statements. Horn clauses are an important sub-class, which amounts essentially to dropping disjunction ("or") from the logic*. {* This subclass appears to be common ground between classical and intuitionist logic.}

The second part of Prolog consists of a very elementary control language, although "language" is really too strong a word. Through this control information, the programmer determines how the Prolog system is to set about constructing a proof. ie. The programmer is specifying exactly how he wants his computation done. The control language consists merely of simple sequencing information, plus a primitive which restricts the system from considering unwanted alternatives in constructing a proof.

There are two distinct ways of understanding the meaning of a Prolog program, one declarative and one imperative or procedural. As far as the declarative reading is concerned, one can ignore the control component of the program. The declarative reading is used to see that the program is correct. The procedural reading is necessary to see whether the program is efficient or indeed practical. Generally speaking, a Prolog program is first conceived declaratively, and then control information is added to obtain a satisfactory procedural aspect.

In the rest of this section we shall merely summarise the syntax we use, and briefly describe the semantics (both declarative and procedural) of the language. For a fuller discussion, see the references on Prolog and logic programming quoted earlier. The reader unfamiliar with Prolog may also find it useful to look at the comparative examples of Prolog, Lisp and Pop-2 listed in Appendix /4./ and discussed in Section /10.1/.

4.1 Syntax And Terminology

A Prolog *program* is a sequence of *clauses*. Each *clause* comprises a *head* and a *body*. The *body* consists of a sequence of zero or more *goals* (or *procedure calls*). For example the clause written:-

```
P :- Q, R, S.
```

has P as its head and Q, R and S as the goals making up its body. A *unit clause* is a clause with an empty body and is written simply as:-

```
P.
```

The head and goals of a clause are all examples of *terms* and are referred to as *boolean terms*.

In general, a term is either an *elementary term* or a *compound term*. An *elementary term* is either a *variable* or a *constant*.

A *variable* is an identifier beginning with a capital letter or with the character '_' (underline). For example:-

```
X, Tree, _LIMIT
```

are all variables. If a variable has just a single occurrence in the clause this may be written simply as (underline):-

```
_
```

(Note that a variable is limited in scope to a single clause, so that variables with the same name in different clauses are regarded as distinct).

A constant is either an atom* or an integer. {* Not to be confused with the use of "atom" in resolution theory, cf. instead Lisp.} An atom is any sequence of characters, which must be written in single quotes unless it is an identifier not confusable with a variable or an integer. For example:-

a, null, =, >=, 'DEC system 10'

are all atoms. Integers are constants distinct from atoms. An identifier consisting of only decimal digits will always represent an integer. For example:-

999, 0, 727

A compound term comprises a functor (called the principal functor of the term) and a list of one or more terms called arguments. Each argument in the list has a position, numbered from 0 upwards. A functor is characterised by its name, which is an atom, and its arity or number of arguments. For example the compound term, whose functor is named "point" of arity 3, with arguments X,Y and Z is written:-

point(X,Y,Z)

In addition to this standard notation for compound terms certain functors may be declared as prefix, infix or postfix operators enabling alternative notation such as

X+Y, (P;Q), 3<4, not P, N factorial

instead of

+(X,Y), ;(P,Q), <(3,4), not(P), factorial(N)

A constant is considered to be a functor of arity 0. Thus the principal functor of a constant is the constant itself.

The principal functor of a boolean term is called a _predicate_. The sequence of clauses whose heads all have the same predicate is called the _procedure_ for that predicate. The depth of nesting of a term in a clause is specified by a _level number_. The head and goals of a clause are at level 0, their immediate arguments at level 1, and so on for levels 2, 3, etc. In general we do not allow a level 0 term to be a variable or integer. A compound term not at level 0 is called a _skeleton term_.

Some sample clauses (for list concatenation and a rather inefficient list reversal) are:-

```
concatenate(cons(X,L1),L2,cons(X,L3)) :-
    concatenate(L1,L2,L3).

concatenate(nil,L,L).

reverse(cons(X,L0),L) :-
    reverse(L0,L1), concatenate(L1,cons(X,nil),L).

reverse(nil,nil).
```

4.2 _Declarative And Procedural Semantics_

The key to understanding a Prolog program is to interpret each clause informally as a shorthand for a statement of natural language. A non-unit clause:-

```
P :- Q, R, S.
```

is interpreted as:-

```
P if Q and R and S.
```

We now have to interpret each boolean term in the program as a simple statement. To do this, one should apply a uniform interpretation of each functor used in the program. eg. for the sample clauses above:-

```
nil = "the empty list"

cons(X,L) = "the list whose first element is X
             and remaining elements are L"

concatenate(L1,L2,L3) = "L1 concatenated with L2 is L3"

reverse(L1,L2) = "the reverse of L1 is L2"
```

Each variable in a clause is to be interpreted as some arbitrary object. Now our four clauses are seen to be shorthand for the following stilted but otherwise intelligible English sentences:-

"The list, whose first element is X and remaining elements are L1, concatenated with L2 is the list, whose first element is X and remaining elements are L3, if L1 concatenated with L2 is L3."

"The empty list concatenated with L is L."

"The reverse of the list, whose first element is X and remaining elements are L0, is L if the reverse of L0 is L1 and L1 concatenated with the list, whose first element is X and remaining elements are the empty list, is L."

"The reverse of the empty list is the empty list."

The <u>declarative semantics</u> of Prolog defines the set of boolean terms which may be deduced to be true according to the program. We say that a boolean term is <u>true</u> if it is the head of some clause instance and each of the goals (if any) of that clause instance is true, where an <u>instance</u> of a term (or clause) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable. That completes the declarative semantics of Prolog.

Note that this recursive definition of truth makes no reference to the sequencing of clauses or the sequencing of goals within a clause. Such sequencing constitutes control information. It plays a role in the procedural semantics, which describes the way the Prolog system executes a program. Here, the head of a clause is interpreted as a procedure entry point and a goal is interpreted as a procedure call. The procedural semantics defines the way a given goal is executed. The aim is to demonstrate that some instance of the given goal is true.

To execute (or solve) goal P, the system searches for the first clause whose head matches or unifies with P. 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. ie. 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. Execution terminates successfully when there are no more goals waiting to be executed. (The system has found an instance of the original goal P which must be true.) Execution terminates unsuccessfully when all choices for matching the original goal P have been rejected. Execution is, of course, not guaranteed to terminate.

In general, backtracking can cause execution of a goal P to terminate successfully several times. The different instances of P obtained represent different solutions (usually). In this way the procedure corresponding to P is enumerating a set of solutions by iteration.

We say that a goal (or the corresponding procedure) has been executed <u>determinately</u> if execution is complete and no alternative clauses exist for any of the goals invoked during the execution (including the original goal).

4.3 <u>The Cut Operation</u>

Besides the sequencing of goals and clauses, Prolog provides one other very important facility for specifying control information. This is the "cut" operator, written '!'. (Originally written '/' and dubbed "slash".) It is inserted in the program exactly like a goal, but is not to be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned. Examples of its use are:-

```
member(X,cons(X,_)):-!.
member(X,cons(_,L)) :- member(X,L).

compile(S,C) :- translate(S,C),!,assemble(C).
```

The effect of the cut operator is as follows. When first encountered as a "goal", cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the goal which caused the clause containing the cut to be activated. In other words, the cut operation <u>commits</u> the system to all choices made since the parent goal was invoked. It renders determinate all computation performed since and including invocation of the parent goal, up until the cut.

Thus the second example above may be read _declaratively_ as "C is a compilation of S if C is a translation of S and C is assembled" and _procedurally_ as "In order to compile C, take the first translation of C you can find and assemble it". If the cut were not used here, the system might go on to consider other ways of translating C which, _although_ _correct_, are unnecessary or are unwanted.

Such uses of cut do no violence to the declarative reading of the program. The only effect is to cause the system to ignore superfluous solutions to a goal. This is the commonest use of cut. However, it is sometimes used in such a way that part of the program can only be interpreted procedurally. Often these cases suggest higher level extensions that might ideally be provided. For example:-

```
          property(X) :- exceptional(X),!,fail.
          property(X).
```

might perhaps be better expressed as:-

```
          property(X) :- unless exceptional(X).
```

Clearly it is not intended that 'property(X)' should be a _bona_ _fide_ solution for any X, as a declarative reading of the second clause would indicate.

Even if better alternatives could be found for the controversial uses of cut, there seems no reason to object to its legitimate use as a purely control device. Consequently we shall treat cut as a basic part of the Prolog language.

5.0 OVERVIEW OF PROLOG IMPLEMENTATION

Prolog implementation rests on the design of processes for:-

(1) (recursive) procedure call,

(2) unification,

(3) backtracking,

(4) the cut operation.

The first is a familiar problem in the implementation of high-level languages and is solved in the usual way through the use of one or more stacks. However because of the nondeterminate nature of Prolog, one cannot automatically contract the stack(s) on procedure exit as is usual. In general this process has to wait until backtracking has caused the procedure to iterate through to its last result.

Unification takes the place of tests and assignment in conventional languages. The major problem is how to represent the new terms (data structures) which are created. The solution devised at Marseille is a novel and elegant approach to the problem of representing structured data. It is essentially the same as Boyer-Moore's "structure sharing with a value array", developed at Edinburgh.

Backtracking requires the ability to remember and rapidly restore an earlier state of computation. Solutions have been devised for a number of experimental languages. Usually the implementation reflects the fact that facilities for nondeterminate computation have been built on top of an existing language. Backtracking is an integral part of Prolog, and consequently is less easily separated from the overall design of an implementation. Indeed it strongly influences the choice of structure sharing, because of the speed with which new data structure can be discarded as well as created through this technique.

The cut operation restores conventional determinacy to a procedure and allows the system to discard "red-tape" information required for backtracking. The internal state becomes closer to that of a conventional high-level language implementation. It will be seen that the implementation of cut is closely bound up with that of backtracking.

5.1 Structure Sharing

The key problem solved by structure sharing is how to represent an instance of a term occurring in the original source program. We shall call the original term a source term* and the new instance a constructed term. {*Also called input terms in the literature on resolution.} The solution is to represent the constructed term by a pair:-

< source term, frame >

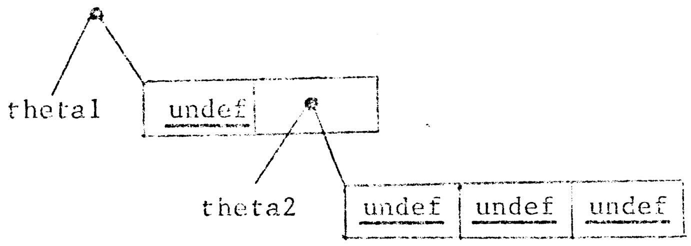
where the frame is a vector of constructed terms representing the values of the variables in the source term. Each variable is given a number which indicates the position in the frame of its value. (We shall also say the variable is bound to that value.) If the variable is unbound, its value is a special construct called 'undef'.

Thus if we are given source terms:-

```
        theta1 = tree(X1,a,X2)

        theta2 = tree(Y1,Y2,Y3)
```

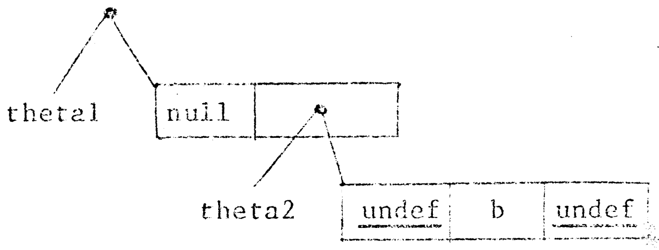
then the constructed term pictured as:-



represents the term:-

```
        tree(X1,a,tree(Y1,Y2,Y3))
```

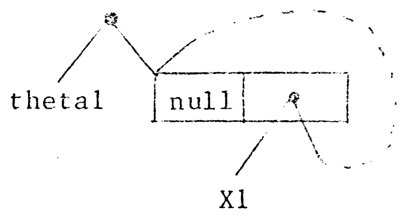
If the source term is a constant then there is no need to provide a frame, so we shall treat constants as being both source terms and constructed terms. Thus the constructed term pictured as:-



represents the term:-

```
        tree(null,a,tree(Y1,b,Y3))
```

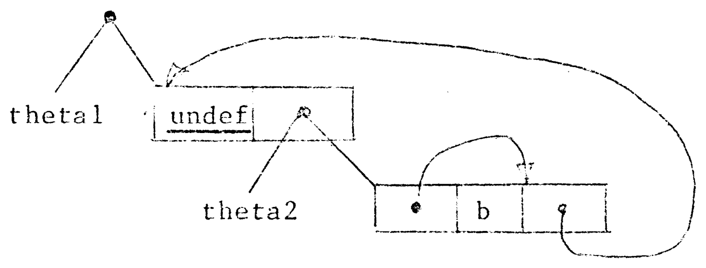
Notice also that the source part of a constructed term may be a variable so that if, for example, X2 in theta1 is bound to X1 and X1 is in turn bound to 'null', then:-



represents:-

tree(null,a,null)

In an actual implementation, a constructed term would generally correspond to a pair of addresses, where one address would point to a literal representation of the source term and the other to a vector of storage cells. In practice we only use this form where the source term is a skeleton, the resulting object being called a molecule. If the source term is a variable, the constructed term corresponds simply to the address of the variable's cell and is called a reference. Thus the constructed term:-



represents:-

tree(X1,a,tree(X1,b,X1))

The advantage of the structure sharing representation is that the cost (in terms of both time and storage) of constructing new terms from skeletons occurring in a clause is, at worst, proportional to the number of distinct variables in those skeletons. If the same data representation were used for constructed terms and source terms (as in Lisp say), then the cost would be at least proportional to the total number of subterms (or, equivalently, of symbols) in the skeletons.

Of course the "direct" representation makes subsequent reference to the components of the data structure somewhat easier. However for most machines (particularly those like the DEC10 with good indirect addressing facilities) this loss of speed is quite small and amply repaid by the savings of space and the speed of creating and discarding new data structure.

When complex terms are built up by unification one cannot in general prevent chains of references being created. When unifying two terms it is important to dereference both values by tracing down any reference chains.

A final point concerns what is known as the "occur check". Strictly a unification should not be allowed which binds a variable to a term containing that variable. This would result in "infinite terms", for example consider:-

```
        infinitelist(X,L) :- L = cons(X,L).
        X = X.
```

In practice this condition never arises in most normal Prolog programs. Where it does, the programmer may well be wanting to consider the infinite term as a legitimate data object (although this is dangerous fo several reasons). Accordingly, Prolog implementations do not bother to make the occur check, as it seems to require an inordinate amount of computation for little practical benefit.

5.2 Procedure Invocation And Backtracking

Just as structure sharing represents an instance of a source term by a pair < source term, frame >, so we may notionally represent a clause instance by a pair:-

< clause, environment >

The environment consists of one or more frames containing the value cells for each variable occurring in the clause, plus all other information associated with this clause instance ie. management information. The environment is created in the course of unifying the head of the clause with the activating goal. The information it comprises may conveniently be stored on one or more stacks, as it is created (by clause activation) and destroyed (by backtracking) on a "last in first out" basis. We may summarise the management information as follows:-

* A record of the parent (activating) goal and its continuation, ie. the goals to be executed when the parent goal is solved. This item can be thought of as a molecule-like pair:-

< parent goal + continuation (both in source program form),
parent environment >

* A list of the remaining source clauses which are alternative candidates for matching the parent goal.

* The environment to backtrack to if the parent goal fails. ie. The most recent environment preceding the current one for which the clause activated is not the only remaining alternative for the activating goal.

* A list of variables bound in the course of unifying the parent goal with the head of this clause. The list need not include variables

whose cells would be discarded anyway on backtracking eg. those in the present environment.

The last three items are needed for backtracking. Effectively, unification is allowed to side-effect existing variable cells (thereby modifying the parent goal and its continuation) but has to leave a record of the variables affected. Backtracking uses this list to reset such variables to '<u>undef</u>'. Unification is also responsible for setting every variable cell in the new environment to '<u>undef</u>' if it is not otherwise initialised.

In our implementation, the environment is split into two frames, local and global, allocated from, respectively, local and global stacks, plus some locations for the "reset list" on a pushdown list called the "trail". The global frame contains the cells for variables occurring in skeletons. The local frame contains the cells for other variables, plus all the management information (apart from the reset list). When a procedure has been executed determinately, the local frame is discarded automatically by a stack mechanism.

5.3 <u>Implementing The Cut Operation</u>

To implement the cut operation it suffices to take the parent's backtrack environment as the current backtrack environment. Optionally one may "tidy up" reset lists for the parent environment onwards, by removing entries for variables which would now be discarded anyway on backtracking.

In our implementation, the "tidying up" is mandatory, since otherwise a "dangling reference" to a discarded local frame may be left in the list of reset variables. A similar argument applies to the global frame if a garbage collector is used. All local frames used in the execution of any goals preceding the cut symbol in the clause concerned are discarded when the cut is effected.

5.4 Compilation

One of the chief innovations of our Prolog implementation is that the clauses are compiled into sequences of simple instructions to be executed directly. This is in contrast to execution by a separate interpreter, where clauses are stored in a more or less literal form. The main effect of the compilation is to translate the head of each clause into a sequence of instructions specialising the unification algorithm to the case where one of the terms to be unified is the head of the clause concerned.

Before describing compilation in detail (see Section /6.9/), it may be helpful to give the flavour of the process through an example. We shall translate Prolog clauses for list concatenation into an informal Algol-style procedure. The clauses are:-

```
concatenate(cons(X,L1),L2,cons(X,L3)) :- concatenate(L1,L2,L3).
concatenate(nil,L,L).
```

The translation follows. The most important point to notice is that much of the unification process is translated into simple assignments.

```
    procedure concatenate is (
          try clause1;
          try clause2;
          fail;
    clause1: (
          local variable L2;
          global variable X,L1,L3;
          prematch skeleton cons(X,L1) against term[1];
*         X,L1 := undef;
*         if need to match subterms then (
               X := subterm[1];
               L1 := subterm[2] );
          L2 := term[2];
          prematch skeleton cons(X,L3) against term[3];
          L3 := undef;
*         if need to match subterms then (
*              match value of X against subterm[1];
*              L3 := subterm[2] );
          claim space for [X,L1,L2,L3];
          call concatenate(L1,L2,L3);
          succeed );
    clause2: (
          temporary variable L;
          match atom nil against term[1];
          L := term[2];
          match value of L against term[3];
          succeed ) )
```

The arguments of the matching goal (= a call to procedure 'concatenate') are referred to as 'term[1]', 'term[2]', ...etc. The arguments of each of these terms are referred to as 'subterm[1]', 'subterm[2]', ...etc. The context for the latter is given by the preceding 'prematch skeleton ...' instruction. 'prematch' is only responsible for matching at the "outermost level". If the corresponding goal argument is a variable, 'prematch' creates a new molecular term and assigns it to this variable. Otherwise 'prematch' merely checks for matching functors; matching of subterms is handled by the instructions which follow the 'prematch'.

If the programmer can guarantee that the 'concatenate' procedure will only be called with first argument as "input" (ie. a non-variable) and third argument as "output" (ie. a variable), then the procedure can be somewhat simplified. Essentially, the lines marked "*" can be omitted and variable L1 becomes a local instead of a global.

6.0 THE PROLOG MACHINE

In the previous sections, we have taken a general look at the processes involved in executing a Prolog program, and have seen how complex terms are built up using the structure-sharing technique. We can now examine in more detail how all this realised in the Prolog machine. Full reference details of the machine are given in Appendices /1./ and /2./.

6.1 The Main Data Areas

Each clause of a Prolog source probgram is represented by a sequence of PLM instructions and literals.* {* Not to be confused with the use of "literal" in resolution theory.} Roughly speaking, there is one instruction or literal for each Prolog symbol (ie. variable, atom or functor). Instructions are executable whereas literals represent fixed data. Both are stored in an area of the machine called the code area. Unlike the other areas of the machine, information in the code area is generally accessed in a "read-only" manner.

The two major writeable areas of the machine are the local stack and the global stack. As their names imply, these areas are used as stacks, that is all storage before a certain point (the "top" of the stack) is in use and all storage after that point is not in use. Furthermore the storage that is in use is referred to in a random access manner. The top of each stack varies continually during the course of a computation. Thus a stack amounts to a variable length vector of storage.

The global stack contains the value cells for global variables, that is variables that occur in at least one skeleton, and which therefore may play a role in constructing new molecules. Other variables are called local variables and their value cells are placed in the local stack. These variables serve merely to transmit information from one goal to another. In addition, the local stack contains management information which determines what happens next in the event of a goal succeeding or failing, and is also used to effect a cut.

Both stacks increase in size when a new goal is tackled, and contract on backtracking. Space can also be recovered from the top of the local stack when a goal is successfully completed and no alternative choices remain in the solution of that goal. It is for just this reason that two stacks are used rather than one. The resulting saving of space can be very substantial for programs which are determinate or partially determinate, as most in fact are. The recovery of space occurs (a) when the end of a clause is reached and the machine can detect that no other choices are open, (b) when a cut is effected and at least one goal precedes the cut in the clause in question. In the latter case all the local stack consumed during the execution of the preceding goals is recovered.

The other main writeable area of the PLM is called the trail. This area is used as a push-down list, ie. it is like a stack, with the difference that items are "pushed" on or "popped" off one at a time on a last-in first-out basis, and are not accessed in any other way. The trail is used to store the addresses of variable cells which need to be reset to 'undef' on backtracking. As with the local and global stacks, it generally increases in size with each new goal and is reduced by backtracking. The cut operation may also have the effect of removing items from the trail.

PLM data items and storage locations come in two sizes, namely short and long. Each area of the PLM comprises a sequence of locations of the same size identified by consecutive addresses.* {* As the trail area is used as a push down list, its locations do not strictly need to be addressable.} A short location is big enough to hold at least one machine address. A long location has room for two addresses. (NB. Short and long locations need not in practice be different in size. In our DEC10 implementation they both correspond to 36-bit locations.) Each variable cell is a long location, so the two stacks comprise long locations, while the trail is made up of short locations. The locations in the code area are short; instructions and literals should be thought of as short items, or multiples thereof.

6.2 Special Registers Etc

Besides the main areas, the PLM has a number of special locations called registers. In general these need only be short locations. Registers V and V1 hold the addresses of the top of the local and global stack respectively. Register TR holds a "push-down list pointer" to the top of the trail.

The environment for each clause instance is represented by a local frame and a global frame, plus some trail entries. The layout is shown in Appendix 1. The global frame is simply a vector of cells for the global variables of the clause. The local frame comprises a vector of local variable cells, preceded by 3 long locations containing management information.

For most of the time, the PLM is in the process of trying to unify the head of some clause against an existing goal. Register A contains the address of a vector of literals representing the arguments of the goal followed by its continuation. The continuation is the instruction at which to continue execution when the goal is solved. The environment of the current goal is indicated by registers X and X1 which hold the addresses of, respectively, the local and global frames for the clause instance in which the goal occurs. Registers V and V1 therefore contain the corresponding information for the environment that unification is endeavouring to construct. The machine insures there is always a sufficient margin of space on each stack above V and V1 for the environment of any clause. It is only when a unification is successfully completed that the V and V1 pointers are advanced.

Registers VV and VV1 indicate, in a similar way, the most recent environment for which the parent goal could possibly be matched by alternative clause(s). Usually we shall have VV=V and VV1=V1, as there will be other clauses in the current procedure which could potentially match the current goal. In this case, register FL contains the address of the instructtion at which to continue if unification should fail.

There are two other important registers which may be set during a unification : register B is set to the address of a vector of literals representing a skeleton, and register Y to the address of the corresponding global frame.

<u>Note:</u> It may be helpful to think of <A,X> as being a molecule representing the current goal and <B,Y> as a molecule representing a level 1 subterm of that goal.

The 3 long locations of management information in each local frame comprise 6 short item fields as illustrated below (the precise arrangement is not really significant):-

VV	FL
X	A
V1	TR

The parent goal is indicated by the X and A fields, mirroring the appropriate values for the X and A registers.

The V1 field contains the address of the corresponding global frame mirroring the V1 register.

The VV field contains the value of the VV register prior to the invocation of the parent goal for this environment. It therefore indicates the most recent choice point prior to this environment.

The FL field contains the failure label for this environment, if any, and is undefined otherwise. The failure label is the address of an instruction at which to continue for an alternative match to the parent goal.

The TR field contains a value corresponding to the state of the TR register at the point the parent goal was invoked.

The VV, FL and TR fields are needed primarily for backtracking purposes.

6.3 Literals

Literals are PLM data items that serve as building blocks to provide a direct representation for certain subterms of the original Prolog source program. In particular they are needed to give skeleton terms a concrete form so that structure sharing can be applied. We shall not attempt to give more details of their internal structure than is necessary. The different types of literal mentioned are assumed to be readily distinguishable.

A *skeleton literal* represents a skeleton term and is a structure comprising a *functor literal* followed by a vector of *inner literals*. Each inner literal is a short item, typically an address which serves as a pointer to the value of the subterm. The size of a functor literal is left undefined, but it contains sufficient information for it to be identified as the functor literal for a particular functor of non-zero arity. It will be written as 'fn(I)' where 'I' uniquely identifies the functor in question. (In our DEC10 implementation, functors and atoms are numbered from 0 upwards and 'I' refers to this number.)

An inner literal is either an <u>inner variable literal</u> or the address of a skeleton literal, <u>atom literal</u> or <u>integer literal</u>. Atom and integer literals are long items written as 'atom(I)' or 'int(N)' where 'I' uniquely identifies the atom in question and 'N' is the value of the integer in question. An inner variable literal will be written 'var(I)' where I is a number identifying the corresponding global variable in the clause concerned. For structure sharing purposes this number is used as an index to select the appropriate cell from an associated frame of (global) variable cells.

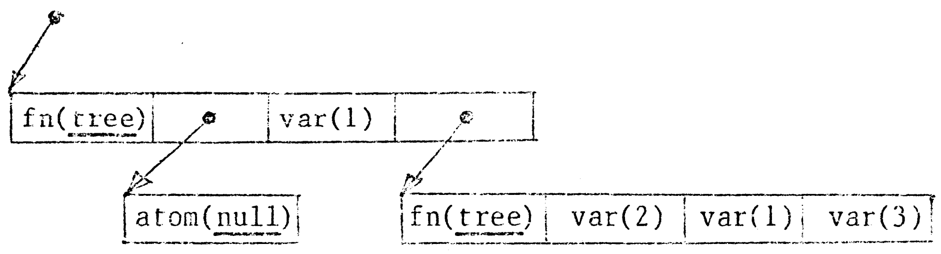
We shall write '[S]' for the address of a structure S. Thus the address of the literal corresponding to the skeleton:-

```
        tree(null,X,tree(Y,X,Z))
```

might be written:-

```
        [fn(tree),
         [atom(null)],
         var(1),
         [fn(tree),
          var(2),
          var(1),
          var(3)]]
```

and pictured as:-



Besides inner literals, which represent the arguments of a skeleton term, the PLM needs <u>outer literals</u> to represent the arguments of a goal. An outer literal is either the address of an atom integer or skeleton literal, or is a <u>local literal</u>, a <u>global literal</u> or a <u>void</u>

literal. Like inner literals, outer literals are short items, which serve as pointers to the values of the subterms they represent.

If a goal argument is a variable, and the variable occurs somewhere else in the clause within a skeleton term, then the argument is represented by a global literal, written 'global(I)' where 'I' is the number of that global variable. If a goal argument is a variable, and that variable occurs nowhere else in the clause then the variable is represented by a void literal, written 'void'. Otherwise a variable appearing as an argument of a goal is represented by a local literal, written 'local(I)' where 'I' is a number identifying the local variable.

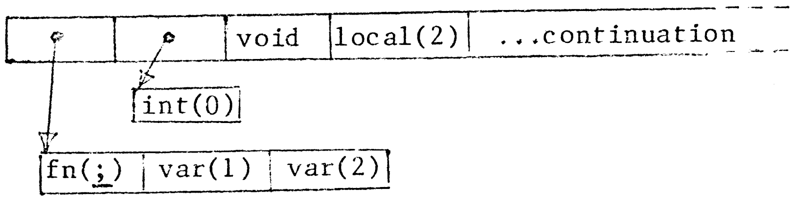
Thus the arguments of the second goal in the clause:-

```
compile(S,C) :- translate(S,D,E), assemble((E;D),0,N,C).
```

might be represented by:-

```
[[fn(;),var(1),var(2)],[int(0)],void,local(2)]
```

or pictured as:-



remembering that the continuation always follows immediately after the last argument literal of the goal.

6.4 Constructs

The set of PLM data items which can appear as the values of variable cells are called constructs. They serve to represent constructed terms in a structure-sharing manner. Once again we shall not attempt to give unnecessary details of their internal structure, but will assume that they are long items and that the different types are readily distinguishable.

The cell for an unbound variable contains the empty construct, written 'undef'. The cell for a variable which has been bound to another variable contains a reference, written 'ref(R)' where R is the address of the other variable's cell. If a variable is bound to an atom or an integer, its value cell will contain the corresponding atom or integer literal. Finally if a variable is bound to an instance of some skeleton, the corresponding construct is called a molecule and written 'mol(S,X)' where S is the address of the corresponding skeleton and X is the address of the corresponding frame.

6.5 Dereferencing

In the following, the process of dereferencing a variable will often be referred to. At any point in a Prolog computation, this process associates a certain non-empty construct with each variable. This construct is said to be the (dereferenced) value of the variable at that point. It is obtained by examining the contents of the variable's cell and repeatedly following any references until a cell is reached which contains a non-reference construct. If this construct is 'undef' the result of the dereferencing is a reference to the cell which contains '<u>undef</u>'. Otherwise the result is the final construct examined.

6.6 <u>Unification Of Constructs</u>

We are now in a position to see how unification works out in practice. Unifying two terms reduces to the task of unifying two constructs which represent them. The first essential is to ensure that the two constructs are fully dereferenced.

If neither construct is a reference, then unification will <u>fail</u> unless we have two equal atoms or two equal integers or two molecules with the same principal functor. In the last case the unification process has to recurse and unify each of the arguments. (The action to be taken on failure is described later.)

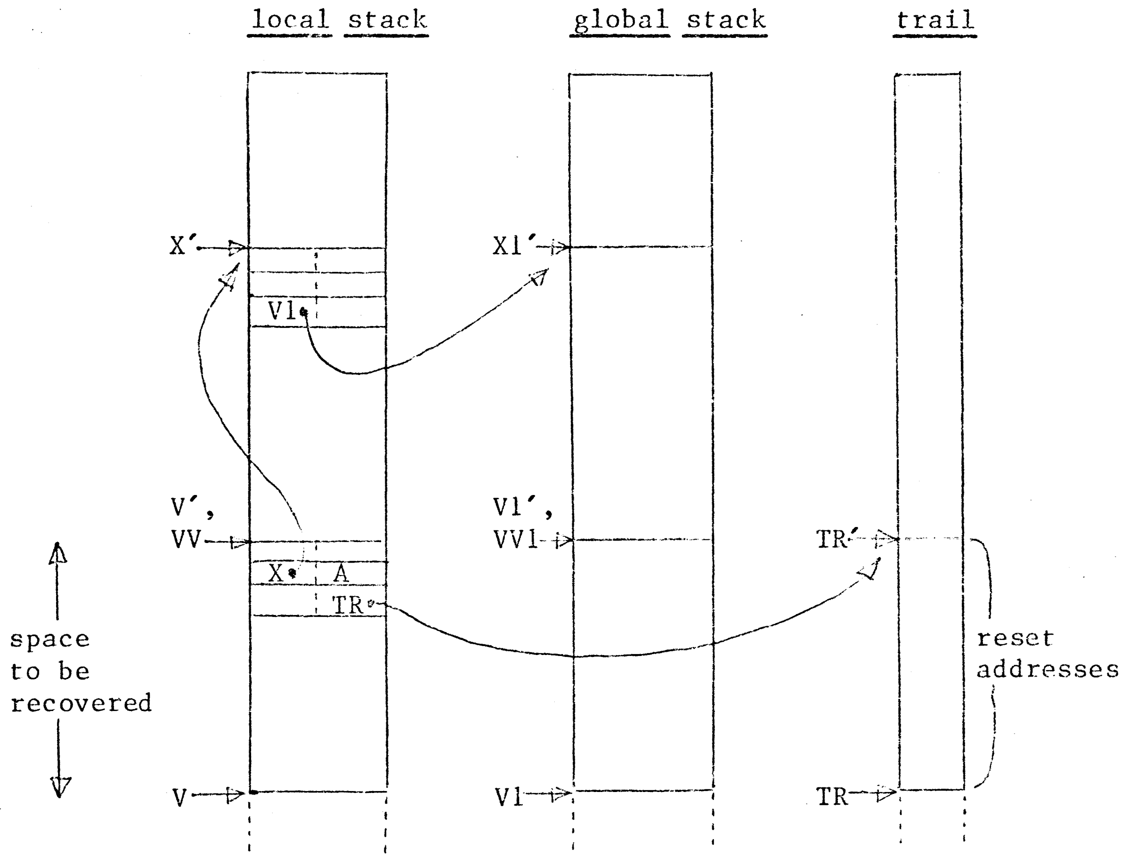
If just one of the constructs is a reference, then the other construct has to be assigned to the cell indicated by the reference.

If both constructs are references, then clearly one reference must be assigned to the cell of the other. It happens to be very important that the more <u>senior</u> reference is assigned to the cell of the more <u>junior</u> reference. <u>A cell in the global stack is always more senior than any cell in the local stack.</u> Otherwise seniority is determined by the cells' addresses - the one earlier in the stack is considered more senior. These precautions are essential to prevent "dangling references" when space is recovered from the local stack following the determinate solution of a goal. (The "dangling reference" is a well known nightmare where a location is left containing the address of a part of storage which has been de-allocated from its original use.) The rules also play an important role for efficiency in tending to prevent long chains of references being built up. In typical Prolog programs it is quite rare for dereferencing beyond the first step to be necessary, if the above scheme is applied.

Whenever a cell is assigned a (non-empty) value, it is usually necessary to "remember" the assignment so that it can be "undone" on subsequent backtracking. The exception is where the cell will in any case be discarded on backtracking. This condition can easily be detected in the PLM by the fact that the cell's address will be greater than the contents of register VV for a local cell or register VV1 for a globalcell. When the assignment has to be remembered the address of the cell concerned is trailed, ie. pushed on to the trail push-down list pointed to by register TR.

6.7 Backtracking

When unification fails, the PLM has to backtrack to the most recent goal for which there are other alternatives still to be tried. Any environments created since the backtrack point are to be erased and the space occupied on the local stack, global stack and trail is to be recovered. Before attempting another unification, all assignments made since the backtrack point to cells which existed before the backtrack point must be undone by setting the values of such cells to 'undef'.



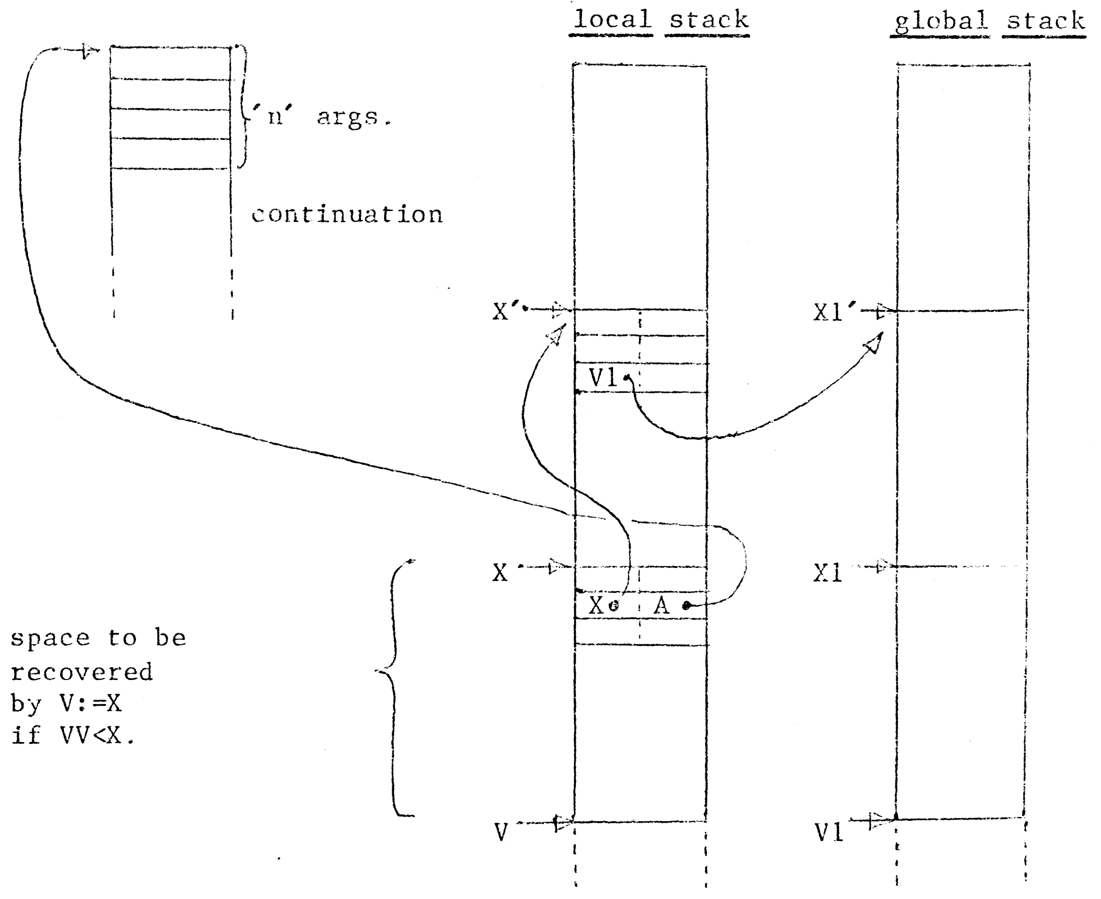
The PLM keeps an up-to-date record of the environment to backtrack to in registers VV and VV1. VV contains the address of the local frame, VV1 the address of the global frame. Note that VV1 is strictly superfluous since it merely shadows the contents of the V1 field in the local frame indicated by VV. The state of the trail corresponding to the backtrack point is indicated by the TR field. The necessary undoing of assignments is achieved by popping addresses of the trail until the original trail state is reached; each cell so addressed is reset to 'undef'. (Some of these cells are probably about to be discarded anyway, but it is harmless to reset them regardless, and this is likely to be simpler.)

For the remainder of the backtracking process, it is convenient to consider two cases. The first is _shallow_ backtracking, where there are other alternatives for the current goal. This is of course easily detected by the fact that VV=V. All that has to be done in this case is to resume execution at the instruction indicated by FL.

In the case of _deep_ backtracking, V and V1 have to reset from VV and VV1 respectively. Registers X, A and FL are reset according to the corresponding fields in the local frame indicated by VV. Register X1 is reset from the V1 field in the local frame corresponding to X. Finally, execution is resumed at the instruction indicated by FL.

6.8 Successful Exit From A Procedure

Backtracking generally corresponds to a failure exit from a procedure. A _success_ exit occurs when the end of a clause is reached If the procedure exit is _determinate_, indicated by VV<X and showing that no choices were made (or remain) in the execution of the procedure, then local storage can be recovered by resetting V from X. Registers X and A are reset from the corresponding fields in the local frame indicated by the present value of X. Register X1 is then reset from the V1 field of the local frame now indicated by X. Finally execution is resumed at the continuation instruction which follows the n short items addressed by A, where n is the arity of the predicate for the procedure concerned.



6.9 Instructions

Having covered the basic structure and function of the PLM, it remains to describe how the clauses which drive it are actually represented. It should be clear that a clause could be stored in a very literal form (cf. a skeleton term) and interpreted directly. This is precisely the way the Marseille interpreter operates. However much of the work that such an interpreter would have to perform can be eliminated by using extra information which is easily computed at the time clauses are first introduced ("compile-time"). This includes:-

(1) Recognising that matching against the first occurrence of a variable in the head of a clause is a special case. The variable must obviously be as yet unbound and one simply has to bind the matching term to it. There is no need to have previously initialised the variable's cell to '<u>undef</u>'. The whole operation is far simpler than in the general case of a subsequent occurrence of the variable.

(2) If one is matching a variable in the head of a clause, and that variable has no other occurrence in the clause, no action at all need be taken. Furthermore if the occurrence is at level 1, no cell need be created for that variable. Similarly, no cell is needed for a single-occurrence variable at level 1 in a goal. Variables with a single occurrence, which is at level 1, are called <u>void variables</u>.

(3) The interpreter generally has to make a recursive call when matching the arguments of a skeleton against a non-variable term. This overhead can be avoided if the skeleton occurs in the current clause head, by associating information about depth of nesting (<u>level number</u>) with each symbol in the head of a clause. (The details will be explained later.) Similarly, the need to keep a count of arguments (of a skeleton or clause head) already matched can be avoided by associating an <u>argument number</u> with each symbol in the head of a clause. (The arguments of a functor are numbered from 0 upwards.)

(4) Normally an interpreter would allocate, and initialise to '<u>undef</u>', all cells for a clause before commencing unification. We have seen that much of this initialisation can be avoided. Also one can postpone the remaining initialisation, and the "red-tape" of storage allocation, as late as possible in the hope that a failure will render them unnecessary.

(5) Variables can be categorised into different types (global, local and temporary), depending on the way they occur in the clause, so that the space occupied by certain variable cells can be recovered earlier than is possible in general.

(6) By bringing together information from the different clauses in a procedure one can optimise the selection of potentially matching clauses and/or share part of the work involved in unifying with each clause head, and in addition provide a means of detecting the important case where the selection of a particular clause is determinate. (See the later section on "Optional Extras").

In general, one Prolog source symbol plus the relevant extra information corresponds to a specific simple operation on the Prolog Machine. If one discounts dereferencing and cases resulting in a failure of unification, the operation usually involves a strictly bounded amount of processing. It is therefore natural to think of the augmented symbols as primitive _machine instructions_ of the PLM.* {* In fact the analogy with a convential machine like the DEC10 is quite close if one compares dereferencing with the DEC10's _effective address calculation_ and unbounded operations with DEC10's block transfer (BLT) and execute (XCT) instructions.}

No executable instructions are generated for the arguments and subterms of a goal. These are represented purely by literal data as indicated earlier. Also, no executable instructions are generated for symbols deeper than the levels 1 and 2 in the head of a clause. This is a purely arbitrary limit based on considerations of cost-effectiveness in practical examples of Prolog programs.

In general, the code for a clause has the form:-

instructions for unification	head of the clause
'neck' instruction	completes the new environment
'call' instructions each followed by outer literals	body of the clause
'foot' instruction	transfers control to parent goal's continuation
skeleton literals (if any)	data (which could be placed elsewhere)

Each goal is represented by an instruction 'call(P)' followed by a list of outer literals for its arguments. 'P' is the address of the procedure code for that predicate. This takes the form:-

```
enter
try(C1)
try(C2)
.
.
trylast(Cn)
```

'enter' is an instruction for initialising part of the management information in a new environment. This function could perhaps better be included in the operation 'call' so that 'enter' would be an ignorable no-operation. (It is included as a separate instruction because of the way it is handled in the DEC10 implementation.) C1 to Cn are the addresses of the code for each of the clauses in the procedure (in order). The last executable instruction in a clause is generally 'foot(N)' where 'N' is the arity of the head predicate.

Before proceeding with a description of the instructions for the head of a clause, we must first complete discussion of the different categories of variable and the exact layout of an environment. The variables of a clause are categorised according to expected "lifetimes" which end when there is no longer any need to remember the variable′s value. The categories are as follows:-

Name	Lifetime ends	Criterion
Global	Backtracking.	Occurs in a skeleton.
Local	Procedure completed successfully and determinately, ie. no choices remain within the procedure.	Multiple occurrences, with at least one in the body and none in a skeleton.
Temporary	Completion of unification with the head of the clause.	Multiple occurrences, all in the head of the clause and none in a skeleton.
Void	None.	A single occurrence, not in a skeleton.

The global variables of a clause are numbered in some arbitrary order which determines their positions in the global frame. Similarly local and temporary variables are numbered to determine their positions in the local frame. The only constraint is that locals precede temporaries. This is so that the temporary part of the local frame can be discarded at the end of unification (see the diagram in Appendix 1). Variables in either frame are numbered from 0 (zero) upwards. No cell is allocated for a void variable. In showing examples of Prolog machine code, we shall assume that the variables of each type are numbered according to their order of appearance in the source clause.

We can now return to the discussion of instructions for the head of a clause. The head is terminated by an instruction 'neck(I,J)' where 'I' is the number of local variables (= the number of the first temporary if any) and 'J' is the number of global variables.

The instructions for an occurrence of a variable in the head are:-

```
        uvar(N,E,I)          uvarl(N,E,I)
        uref(N,E,I)          urefl(N,E,I)
```

'uvar' or 'uvarl' is used if it is the first occurrence, 'uref' or 'urefl' otherwise. 'uvar' corresponds to level 1 and 'uvarl' to level 2, and <u>similarly for all other pairs of instructions named 'name' and 'namel'</u>. 'N' is the argument number of the occurrence, 'E' is the frame ('local' or 'global') containing the variable's cell, and 'I' is the number of the variable. No instruction is needed for a variable with a single occurrence.

Similarly there are instructions for an occurrence of an atom or integer in the head:-

```
        uatom(N,I)           uatoml(N,I)
        uint(N,I)            uintl(N,I)
```

Once again, 'N' is the argument number of the occurrence. For an integer, 'I' is the actual value of the integer. For an atom, 'I' uniquely identifies that atom.

For a skeleton at level 1, the instructions are:-

```
                uskel(N,S)
                init(I,J)
                ifdone(L)
                .
                . argument
                . instructions
                .
          L:
```

'N' is the argument number of the skeleton within the head. 'S' is the address of a corresponding skeleton literal (which may be assumed to be placed after the 'foot' instruction). The global variables which have their first occurrences within the skeleton are numbered from 'I' through 'J'-1. The effect of the 'init' instruction is to initialise these variables to '<u>undef</u>'. If 'I'='J', the instruction is a no-operation and may be omitted. The instruction 'ifdone' causes the instructions for the arguments of the skeleton to be skipped if the matching construct is a reference. 'L' is the address of the instruction following the last argument instruction.

Note that the arguments of the skeleton could be coded in any order since each instruction contains the argument number explicitly. (A "first occurrence" of a variable would then mean the first occurrence in the code.) Similarly for the arguments of the head boolean term itself.

A skeleton at level 2 is coded simply as:-

```
                uskel1(N,S)
```

where 'N' and 'S' are analogous to the use in 'uskel'.

Immediately before a 'neck' instruction there are two instructions:-

```
                init(I1,J1)
                localinit(I2,J2)
```

The global and local variables which have their first occurrences in the body of the clause are numbered respectively from 'I1' through 'J1'-1 and from 'I2' through 'J2'-1. Once again, either instruction is an omissable no-operation if the two numbers are equal.

The instruction corresponding to the cut symbol is 'cut(I)' where 'I' is the number of local variabes in the clause. There are a number of instructions which simply replace some common combinations of instructions:-

```
neckfoot(J,N)       :  neck(0,J); foot(N)
neckcut(I,J)        :  neck(I,J); cut(I)
neckcutfoot(J,N)    :  neck(0,J); cut(0); foot(N)
```

That completes the basic instruction set of the PLM. We have not described in detail the effect of each instruction, although this should be clear from earlier discussion of how the PLM operates. Full details are given in Appendix 2.

6.10 Examples Of Prolog Machine Code

Let us now illustrate the way Prolog source clauses are translated into Prolog Machine Code by considering some examples.

6.10.1

List membership is defined by the following straight-forward clauses:-

```
member(X,cons(X,L)).
member(X,cons(Y,L)) :- member(X,L).
```

The first clause has two global variables X and L. The second has one local X and two globals Y and L. The code for the clauses is as follows. Addresses etc. are represented by underlined identifiers and where appropriate the corresponding instruction is indicated by a label as in conventional assembly languages.

```
           Code                          Source

clause1:   uvar(0,global,0)              member(X,
           uskel(1,label2)               cons(
           init(1,2)
           ifdone(label1)
           urefl(0,global,0)             X,L)
label1:    neckfoot(2,2)                 ).
label2:    fn(cons)
           var(0)
           var(1)

clause2:   uvar(0,local,0)               member(X,
           uskel(1,label4)               cons(Y,
           init(0,2)
           ifdone(label3)
           uvarl(1,global,1)             L)
label3:    neck(1,2)                     ):-
           call(member)                  member(
           local(0)                      X,
           global(1)                     L)
           foot(2)                       .
           fn(cons)
           var(0)
           var(1)

member:    enter
           try(clause1)
           trylast(clause2)
```

6.10.2

An example of a use of 'cut' is the following definition of the maximum of two quantities:-

```
maximum(X,Y,Y) :- X<Y, !.
maximum(X,Y,X).
```

(Here cut is not purely a control device; the second clause can be interpreted as "the maximum of X and Y is X by default if it is not the case that X is less than Y".) The first clause has two local variables while the second has one temporary X and one void Y. The corresponding code is:-

```
              Code                          Source

   clause1:   uvar(0,local,0)               maximum(X,
              uvar(1,local,1)               Y,
              uref(2,local,1)               Y
              neck(2,0)                     ):-
              call(<)                       <(
              local(0)                      X,
              local(1)                      Y),
              cut(2)                        !
              foot(3)                       .

   clause2:   uvar(0,local,0)               maximum(X,Y,
              uref(2,local,0)               X
              neckfoot(0,3)                 ).

   maximum:   enter
              try(clause1)
              trylast(clause2)
```

6.11 Mode Declarations

In the previous section we saw that the code for list membership included skeleton literals. Now these skeleton literals are only really used if the membership procedure needs to construct new lists, ie. when the second argument in the call is (dereferences to) a reference construct. This is unlikely to be the case. Usually the programmer will call 'member' simply to check whether something is a member of an existing list. In this case the 'cons' subterms of the 'member' procedure will serve only to decompose an existing data structure, not to construct a new one.

If the programmer can guarantee to restrict the use of a predicate in this kind of way, then the system can optimise the code generated. The main potential improvements are:-

* Unnecessary code can be dispensed with. If a skeleton term always serves as a "destructor" then a skeleton literal is not needed. If it always serves as a "constructor" then no executable instructions are needed for the arguments.

* If these changes result in a variable no longer appearing in a skeleton literal, then that variable no longer needs to be global. Its cell can therefore be allocated on the local stack and space recovered on determinate procedure exit.

Accordingly, the PLM allows the programmer to specify an optional mode declaration for each predicate. Some examples of the syntax used are:-

```
:-mode member(?,+).
:-mode concatenate(+,+,-).
```

The first declaration states that, in any call of 'member', the second argument will be a non-reference construct and the first argument is unrestricted. The declaration for 'concatenate' indicates that the first two arguments are always non-reference constructs and the third is always a reference. ie. 'concatenate' is applied to two given lists to create a new third list.

These examples illustrate all the cases of mode information currently accepted by, and useful to, the PLM. The idea could obviously be extended. We should emphasise that the declarations are optional and do not affect the visible behaviour of the program except in regard to efficiency (provided the restrictions imposed are valid). If no mode declaration is given for a predicate, it is equivalent to a declaration with all arguments '?'.

The effect on the PLM of a mode declaration is limited to changes to the code generated for skeletons at level 1 and consequent re-categorisation of variables. If a skeleton is in a '-' position, it is playing a purely "constructive" role and the code is:-

```
uskelc(N,S)
init(I,J)
```

ie. A 'uskelc' instruction replaces the 'uskel' instruction and the 'ifdone' and argument instructions are dropped.

If the skeleton is in a '+' position, it is playing a purely "destructive" role and the code is:-

```
uskeld(N,I)
.
.   argument
.   instructions
.
```

Here 'I' uniquely identifies the functor of the skeleton. The 'init' and 'ifdone' instructions are dropped and no skeleton literal is necessary. However if any argument of the skeleton is itself a skeleton, the code for that argument becomes:-

```
init(I,J)
uskell(N,S)
```

'N' and 'S' are the argument number and address of a skeleton literal for the subterm. 'I' through 'J'-1 are the numbers of the global variables having their first occurrences in 'S'. As usual, the 'init' instruction can be omitted if 'I'='J'.

Note that if 'uskelc' encounters a non-reference, or 'uskeld' a reference, an error message is given and a failure occurs.

Finally we should observe that in the previously stated criteria for categorising variables, "occurrence in a skeleton" should be construed as "occurrence in a skeleton literal". From a practical point of view it is the re-classification of variables into more desirable categories which is of major importance. The full benefit of using two stacks rather than one for variable cells can only be obtained if mode declarations are used. For this reason we have not treated mode declarations as one of the "optional extras" considered later.

6.12 More Examples Of Prolog Machine Code

6.12.1

Let us now see how the mode declaration given for 'member' affects the code. There are no longer any global variables. Two of them become voids, one temporary and one local:-

```
            Code                         Source

clause1:    uvar(0,local,0)              member(X,
            uskeld(1,cons)               cons(
            urefl(0,local,0)             X,L)
            neckfoot(0,2)                ).

clause2:    uvar(0,local,0)              member(X,
            uskeld(1,cons)               cons(Y,
            uvarl(1,local,1)             L)
            neck(2,0)                    ):-
            call(member)                 member(
            local(0)                     X,
            local(1)                     L)
            foot(2)                      .

member:     enter
            try(clause1)
            trylast(clause2)
```

6.12.2

A good example for illustrating many different features of code generation is the following "quick-sort" procedure:-

```
:-mode sort(+,-).
:-mode qsort(+,-,+).
:-mode partition(+,+,-,-).

sort(L0,L) :- qsort(L0,L,nil).

qsort(cons(X,L),R,R0) :-
   partition(L,X,L1,L2),
   qsort(L2,R1,R0),
   qsort(L1,R,cons(X,R1)).
qsort(nil,R,R).

partition(cons(X,L),Y,cons(X,L1),L2) :-
   X =< Y, !, partition(L,Y,L1,L2).
partition(cons(X,L),Y,L1,cons(X,L2)) :-
   partition(L,Y,L1,L2).
partition(nil,_,nil,nil).
```

The code generated is as follows:-

	Code	Source
clause1:	uvar(0,local,0)	sort(L0,
	uvar(1,local,1)	L
	neck(2,0)	):-
	local(0)	L0,
	local(1)	L,
	[atom(nil)]	nil)
	foot(2)	.
clause2:	uskeld(0,cons)	qsort(cons(
	uvarl(0,global,0)	X,
	uvarl(1,local,0)	L),
	uvar(1,local,1)	R,
	uvar(2,local,2)	R0
	init(1,2)	
	localinit(3,5)	
	neck(5,2)	):-
	call(partition)	partition(
	local(0)	L,
	global(0)	X,
	local(3)	L1,
	local(4)	L2),
	call(qsort)	qsort(
	local(4)	L2,
	global(1)	R1,
	local(2)	R0),
	call(qsort)	qsort(
	local(3)	L1,

```
          local(2)                      R,
          label1                        cons(X,R1))
          foot(3)                       .

label1:   fn(cons)
          var(0)
          var(1)

clause3:  uatom(0,nil)                  qsort(nil,
          uvar(1,local,0)               R,
          uref(2,local,0)               R,
          neckfoot(0,3)                 ).

clause4:  uskeld(0,cons)                partition(cons(
          uvarl(0,global,0)             X,
          uvarl(1,local,0)              L),
          uvar(1,local,1)               Y,
          uskelc(2,label2)              cons(X,L1),
          init(1,2)
          uvar(3,local,2)               L2
          neck(3,2)                     ):-
          call(=<)                      =<(
          global(0)                     X,
          local(1)                      Y),
          cut(3)                        !,
          call(partition)               partition(
          local(0)                      L,
          local(1)                      Y,
          global(1)                     L1,
          local(2)                      L2)
          foot(4)                       .
label2:   fn(cons)
          var(0)
          var(1)

clause5:  uskeld(0,cons)                partition(cons(
          uvarl(0,global,0)             X,
          uvarl(1,local,0)              L),
          uvar(1,local,1)               Y,
          uvar(2,local,2)               L1,
          uskelc(3,label3)              cons(X,L2)
          init(1,2)
          neck(3,2)                     ):-
          call(partition)               partition(
          local(0)                      L,
          local(1)                      Y,
          local(2)                      L1,
          global(1)                     L2)
          foot(4)                       .
label3:   fn(cons)
          var(0)
          var(1)

clause6:  uatom(0,nil)                  partition(nil,_,
          uatom(2,nil)                  nil,
          uatom(3,nil)                  nil
          neckfoot(0,0).                ).
```

6.12.3

The following example illustrates the coding of nested skeletons:-

```
:-mode rewrite(+,?).

rewrite(X or (Y and Z), (X or Y) and (X or Z)):-!.
```

```
            Code                           Source

clausel:    uskeld(0,or)                   rewrite(or(
            uvarl(0,global,0)              X,
            init(1,3)
            uskell(1,label2)               and(Y,Z)),
            uskel(1,label3)                and(
            ifdone(labell)
            uskell(0,label4)               or(X,Y),
            uskell(1,label5)               or(X,Z))
labell:     neckcutfoot(3,2)               ):-!.
label2:     fn(and)
            var(1)
            var(2)
label3:     fn(and)
            label4
            label5
label4:     fn(or)
            var(0)
            var(1)
label5:     fn(or)
            var(0)
            var(2)
```

7.0 DEC10 IMPLEMENTATION DETAILS

In this section we shall indicate how the PLM can be efficiently realised on a DEC10. A summary of the essential characteristics of this machine is given in Appendix /3./. Fuller details of the implementation of PLM instructions and literals are given in Appendix /2./.

Short and long items both correspond to 36-bit words. A special register corresponds to one of the sixteen fast accumulators. For each writeable area there is set aside a (quasi-) fixed block of storage in the low segment. The trail is accessed via a push-down list pointer held in TR.

The DEC10 effective address mechanism contributes crucially to the overall speed of the implementation. Each inner and outer literal is represented by an address word which is generally accessed indirectly. ie. The indirection bit is usually set in any DEC10 instruction which refers to the address word. In particular, the address word for a variable specifies the address of its cell as an offset relative to an index register. The index register will be loaded with the address of the appropriate frame. In other cases (constant or skeleton), the address word will contain a simple address. The net result is that, despite structure-sharing, it only takes <u>one</u> instruction to access a unification argument. Moreover, in the majority of cases no further dereferencing of the argument will be necessary. This can best be illustrated by looking at the code for an example such as 'uvar(3,global,5)' :-

```
        MOVE   T,@3(A)    ;indirect load of argument into T
        TLNN   T,MSKMA    ;check construct is a molecule or constant
        JSP    C,$UVAR    ;if not, call out-of-line subroutine
        MOVEM  T,5(V1)    ;store argument in appropriate cell
```

Thus in the majority of cases only 3 instructions are executed to complete this unification step. The matching term might be 'global(4)' represented by:-

```
        WD 4(Xi)
```

where 'WD' indicates an address word with zero instruction field. If the cell corresponding to this variable contains a molecule say, the effect of the 'MOVE' instruction will be to load the molecule into register T. Note: If the cell contained '<u>undef</u>', subroutine '$UVAR' would be responsible for recovering the address of the cell. This is easily achieved by the instruction:-

```
        MOVEI T,@-3(C)
```

which simply loads the result of the effective address calculation into T. '-3(C)' refers back to the original 'MOVE' instruction. A similar operation is needed if the matching term is a skeleton. More generally, this illustrates how part (or all) of a PLM instruction can be performed out-of-line on the DEC10 with very little overhead, as the subroutine can easily refer back to the in-line code.

A molecule 'mol(Skeleton,Frame)' is represented by a word:-

```
        XWD frame,skeleton
```

The pair is inverted to facilitate accessing the arguments by indexing. A reference construct corresponds to a simple address word with left half zero. In passing, note that although <u>all</u> dereferencing could be accomplished by a single instruction (with a different representation of constructs and the indirection bit set in a reference), this would not be cost-effective (multi-step dereferencing is too rare to justify the extra overheads). 'undef' is represented by a zero word, as this value is easily initialised and recognised.

Both the 'call' and 'try' instructions are represented simply by 'JSP's :-

```
        JSP A,predicate ;call predicate
        JSP FL,clause   ;try clause
```

other instructions are implemented as a mixture of in-line code and call to out-of-line subroutines via:-

```
        JSP C,routine
```

The 'uskel' instruction, if it matches a non-reference has the effect of loading B with the address of the corresponding frame. If it matches a reference, Y is set to zero and 'ifdone' is achieved by:-

```
        JUMPE B,label
```

The TR field in a local frame holds the left-half of the corresponding value for the TR register. This enables the trail to be easily relocated since the TR fields will effectively contain trail offsets rather than trail addresses.

Atom, integer and functor literals are represented by words:-

```
        XWD $ATOM,i
        XWD $INT,i
        XWD $SKEL,i
```

The left halves $ATOM, $INT, $SKEL serve to label the different types of literal. The right half 'i' is either the value of the integer, or a functor table offset. The functor table contains information, such as names and arities, associated with atoms and functors.

8.0 OPTIONAL EXTRAS

In this section we discuss some "optional extras" which can substantially improve the efficiency of the PLM. Because they are not strictly essential, we treat them separately in order to keep the basic description of the PLM as simple as possible. However since both "extras" provide substantial benefits at comparatively little cost, they should be regarded as standard.

8.1 Indexing Of Clauses

The basic PLM eventually tries every clause in a procedure when seeking to match a goal (unless "cut" is used explicitly, or implcitly when a proof has been found). The code for each clause is actually entered, although an early failure in unification may quickly re-route control to the next clause. This is fine so long as there are only a few clauses in a procedure or when a high proportion of the clauses are going to match. However there are often cases where the clauses for a predicate would conventionally correspond to an array or table of information rather than a single procedure. Typically there are many clauses with a variety of different non-variable terms in one or more argument positions of the head predicate. An example might be the clauses for a predicate 'phonenumber(X,N)' where 'N' is the phone number of person 'X'.

Ideally one would like the system to access clauses "associatively", to achieve a higher "hit" ratio of clauses matched to clauses entered. In other words clauses should be <u>indexed</u> on a more detailed basis than head predicate alone. However there is a danger of generating much extra indexing information which is not needed in practice. For example a standard telephone directory is indexed so as to facilitate answering questions of the form 'phonenumber(aperson,X)'. To cater for questions of the form 'phonenumber(X,anumber)' would require another weighty volume which would be useless to the average customer. So in designing an indexing scheme one has to balance generality against the benefit realised in practice from the extra information stored. Also the whole object of the scheme will be nullified if the indexing process is not fast. In Prolog, there is an additional constraint that the clauses must be selected in the order they appear in the program, as this order frequently constitutes vital control information.

Besides the main objective of speeding up the selection of clauses to match a goal, indexing also helps the machine to detect that a choice is determinate because no further clauses in the procedure will match. This is important for determining when space can be reclaimed from the local stack.

The indexing scheme we shall describe is relatively straightforward, and results in clauses being indexed by predicate <u>and</u> principal functor of the first argument in the head (if this term is non-variable). This is achieved by replacing the first PLM instruction in each clause by extra indexing instructions in the procedure code. Much work is thereby telescoped, and clauses can often be selected by a fast "table lookup". It is a simple compromise solution which is perfectly adequate for many cases of practical interest, in particular for compiler writing in Prolog. Moreover in many other cases it is not difficult to rewrite the program to take advantage of the indexing provided, cf. the way two dimensional arrays are conventionally mapped onto one dimensional storage, in Fortran implementations say. For example one might replace a set of unit clauses for 'matrix' by unit clauses for 'vector' plus the clause:-

```
        matrix(I,J,X) :- K is I*20+J, vector(K,X).
```

provided we have:-

```
        :-mode matrix(+,+,?).
```

The indexing then gives rapid access to the X such that 'matrix(I,J,X)' for given 'I' and 'J'. It also enables the machine to take advantage of 'matrix' being a single valued function from 'I' and 'J' to 'X' and avoid retaining any local storage used in a call to 'matrix'. Such rewriting can usually be done without greatly impairing the "naturalness" and readability of the program.

We shall now describe how the improved indexing scheme affects the PLM instructions generated. Basically the first instruction in each clause is to be omitted and the procedure code becomes more complex. The clause sequence of a procedure is divided into sections of consecutive clauses with the same type of argument at position 0 in the head. The two types are "variable" and "non-variable". The former corresponds to a general section and the latter to a special section. The procedure code now takes the form of an 'enter' instruction followed by alternating special and general sections:-

```
enter
.
.
.
gsect
.
.  general
.  section
.  code
.
ssect(L,C)
.
.  special
.  section
.  code
.
.
.
.
```

Each general section commences with an instruction 'gsect'. This instruction is equivalent to 'uvar(0,local,0)'. The clauses for a general section have at least this one mandatory <u>local</u> variable which is bound to the term passed as first argument in the call. If the variable at position 0 in the head is <u>global</u>, an extra instruction:-

```
ugvar(I)
```

is placed at the beginning of the clause code. This instruction has the same effect as 'uvar(0,global,I)'. The code for the general section is simply:-

```
gsect
try(C1)
try(C2)
.
.
try(Cn)
```

where C1 through Cn are the addresses of the clauses in the section. If it is the final section of the procedure, the last instruction is:-

```
trylast(Cn)
```

The code for a special section takes the form:-

```
                    ssect(Label,Next)
                    .
                    .  non-reference
                    .  code
                    .
        Label:      .
                    .  reference
                    .  code
                    .
                    endssect
        Next:
```

'ssect' is responsible for dereferencing the term passed as first argument and if the result is a reference, control is transferred to the <u>reference code</u> commencing at 'Label'. The reference code is a sequence of instructions, each of which is one of:-

```
tryatom(I,C)
tryint(I,C)
tryskel(S,C)
```

according to the form of the first argument in the head of the clause. These instructions are respectively equivalent to:-

```
uatom(0,I); try(C)
uint(0,I); try(C)
uskel(0,S); try(C)
```

for the special case of matching against a reference. If it is the final section of the procedure, the instruction 'endssect' is omitted and one of:-

```
trylastatom(I,C)
trylastint(I,C)
trylastskel(S,C)
```

takes the place of the last instruction in the section. These instructions are equivalent to:-

```
uatom(0,I); trylast(C)
```

etc. The instruction 'endssect' causes the following 'gsect' instruction to be skipped and takes over its role for the special case concerned. The 'endssect' instruction is not strictly essential and could be treated as an ignorable no-operation instead.

The "meat" of the improved indexing scheme lies in the <u>non-reference</u> <u>code</u> which immediately follows an 'ssect' instruction. In general this code has the form:-

```
switch(N)
case(L1)
case(L2)
.
.
case(LN)
.
.
.  testcode
.
.
```

Basically, the code switches on a "hash code" determined by the first argument in the call to some test code which finds (by a sequence of tests against functors having the same "hash code") the appropriate clause(s) (if any) for this functor. Each of these clauses is then 'try'ed in turn. Usually there will be no more than one clause per "hash code" value and so the cost of finding this clause is <u>independent of the number of clauses in the section</u>.

In more detail, instruction 'switch' computes a <u>key</u> determined by the principal functor of the first argument in the call (which has been dereferenced by 'ssect'). 'N' is a certain power of 2 which is the number of 'case' instructions following. The value of N is arbitrary and is currently chosen to be the smallest power of 2 which is not less than the number of clauses in the section. A number M in the range 0 to N-1 is derived from the key by extracting the least significant I bits where N is 2 to the power I. ie. M is the key modulo N. Control is then transferred to the address 'L' where the (M+1)th. 'case' instruction is 'case(L)'. If there are only a few clauses in the section (currently <5) then the 'switch' and 'case' instructions are omitted and testcode as if for a single case follows.

In general the testcode indicated by the address 'L' in a case instruction is of the form:-

```
.
.  'if' instructions
.
.
goto(Next)
```

where 'Next' is the address of the next general section. An instruction 'goto(L)' merely transfers control to address 'L'. If the list of 'if' instructions would otherwise be empty (see below), all the testcode is omitted and the corresponding case instruction is 'case(Next)'. An 'if' instruction is one of:-

```
ifatom(I,Label)
ifint(I,Label)
ifskel(I,Label)
```

There is one 'if' instruction for each different atom, integer or functor which occurs as a principal functor of the first argument of the head of a clause in this section, and whose key corresponds to the case concerned. The 'if' instructions can be ordered arbitrarily. 'I' uniquely identifies the atom, integer or functor concerned. Often there will only be one clause for this constant or functor, in which case 'Label' is the address of the clause's code. The effect of the 'if' instruction is to transfer control to 'Label' if the first argument of the goal matches the constant or functor indicated by 'I'. Since 'ssect(_,Next)' will have set the FL field of the current environment to 'Next', the net effect of the 'if' instruction is as if (for example):-

```
uatom(0,I); try(Label)
```

occurred immediately before the next general section. If there is more than one clause for a particular constant or functor, 'Label' is the address of code of the following form:-

```
try(C1)
try(C2)
.
.
goto(Next)
```

[? Need 'reload' instructions if argument 0 is a skeleton. ?] Here 'Next' is once again the address of the following general section and the Ci are the addresses of the code for the different clauses, in order of the source program.

If a special section is the final section in a procedure, the opening instruction is:-

```
ssectlast(Label)
```

This instruction is like 'ssect' but if the first argument of the call is a non-reference the machine is prepared for deep backtracking on failure. (cf. the relationship between 'try' and 'trylast'). The remaining code is similar to that for 'ssect', with an address '<u>fail</u>' replacing all occurrences of the 'Next' address. If control is transferred to '<u>fail</u>', the effect is to instigate deep backtracking. If there is more than one clause for a constant or functor, the code is:-

```
notlast
try(C1)
try(C2)
.
.
trylast(Cn)
```

Instruction 'notlast' prepares the machine for shallow instead of deep backtracking.

Finally if the type of the first argument is restricted by a mode declaration, part of the special section code can be omitted. If the restriction is '+', the reference code is omitted and the label in the 'ssect' instruction becomes '<u>error</u>'. If control is transferred to '<u>error</u>' a diagnostic message is given followed by deep backtracking. If the restriction is '-', the non-reference code is replaced by the instruction 'goto(<u>error</u>)'. Thus the procedure code checks that the type of the first argument is consistent with any mode declaration.

8.1.1 <u>Example</u> -

We shall now illustrate the clause indexing by showing the indexed procedure code for the following clauses:-

```
        call(X or Y) :- call(X).
        call(X or Y) :- call(Y).
        call(trace) :- trace.
        call(notrace) :- notrace.
        call(read(X)) :- read(X).
        call(write(X)) :- write(X).
        call(nl) :- nl.
        call(X) :- ext(X).
        call(call(X)) :- call(X).
        call(true).
        call(repeat).
        call(repeat) :- call(repeat).
```

The procedure code is as follows:-

```
call:      enter
           ssect(refl,next)
           switch(8)
           case(label1)
           case(label2)
           case(next)
           case(label3)
           case(label4)
           case(label5)
           case(next)
           case(next)
label1:    ifskel(or,list1)
           goto(next)
label2:    ifatom(trace,clause3)
           goto(next)
label3:    ifskel(read,clause5)
           goto(next)
label4:    ifatom(notrace,clause4)
           ifskel(write,clause6)
           goto(next)
label5:    ifatom(nl,clause7)
           goto(next)
list1:     try(clause1)
           try(clause2)
           goto(next)
ref1:      tryskel(or,clause1)
           tryskel(or,clause2)
           tryatom(trace,clause3)
           tryatom(notrace,clause4)
           tryskel(read,clause5)
           tryskel(write,clause6)
           tryatom(nl,clause7)
           endssect
next:      gsect
           try(clause8)
           ssectlast(ref2)
           ifskel(call,clause9)
           ifatom(true,clause10)
           ifatom(repeat,list2)
           goto(fail)
list2:     notlast
           try(clause11)
           trylast(clause12)
ref2:      tryskel(call,clause9)
           tryatom(true,clause10)
           tryatom(repeat,clause11)
           trylastatom(repeat,clause12)
```

8.2 Garbage Collection

We have already seen how local storage used during the determinate execution of a procedure can be recovered at virtually no cost. It is also possible to recover part of the global storage used, though the garbage collection (GC) process needed is rather expensive, hence the importance of classifying variables into locals and globals. Neither of these techniques can reclaim storage from a procedure until it has been completed determinately. While a procedure is still active, there is little potential for recovering any of its storage.

Because of the cost, garbage collection should only be instigated when there is no longer enough free space on the global stack. It involves tracing and marking all the global cells which are still accessible to the program, and then compacting the global stack by discarding inaccessible cells with remapping of any addresses which refer to the global stack. A drawback, attributable to the structure sharing representation, is that not all the inaccessible cells can be discarded. They may be surrounded in the frame by other accessible cells, and the relative positions in the frame of all accessible cells must be preserved. This disadvantage relative to a "direct" representation using "heap" storage is nevertheless probably outweighed in most cases by the general compactness of structure-sharing.

We say that a global frame is active if the corresponding local frame still exists. Otherwise the frame is said to be passive. Passive global frames correspond to procedures which have been completed determinately. The aim of GC is to reduce the sizes of passive global frames by discarding inaccessible cells from either end of the frame. If possible the frame is dispensed with altogether.

In order to perform the GC process, it is necessary to make some slight changes to the format of the data on the two stacks:-

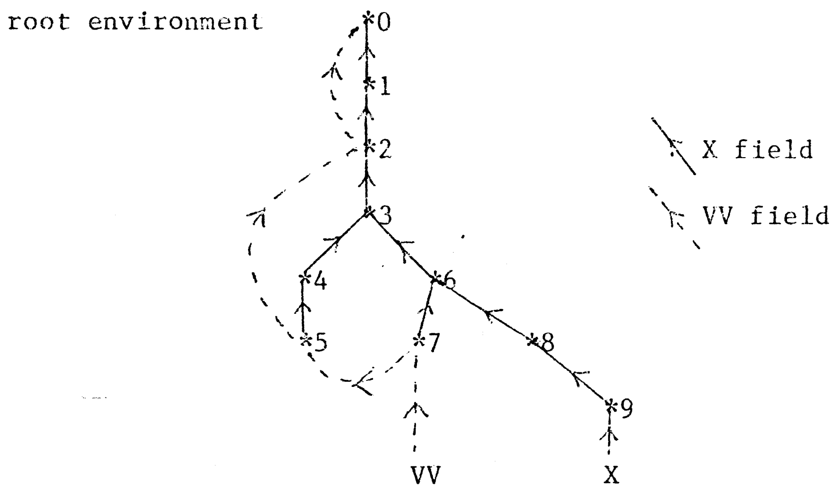
(1) An extra GC bit must be made available in (or associated with) each global cell. This bit will be set during the trace and mark phase if the cell is to be retained.

(2) An extra (long) location is needed at the beginning of each global frame. This contains a special value of type 'mark(N)' distinguishable from other constructs. During GC, this location marks the start of another global frame and the value of N indicates the amount the frame is to be displaced when compaction takes place. If the frame is to be discarded altogether, the value in the location is set to 'discard(N)', where N is the relocation factor which would apply if the frame were not to be discarded.

(3) An extra 1-bit of management information is needed in the local frame. This indicates whether or not there is a corresponding global frame.

The GC process needs to be able to trace all existing local frames (and the corresponding active global frames). The information needed resides in the X and VV fields of the local frames, with the V1 fields indicating the paired active global frames. The following algorithm performs the enumeration:-

```
local frame pointer Parent := register X;
local frame pointer Alternative := register VV;
while Alternative >= root environment, do
     (while Parent > Alternative, do
          (select(Parent);
          Parent := field X of Parent);
     select(Alternative);
     Parent := field X of Alternative;
     Alternative := field VV of Alternative)
```



We can now outline the entire GC process:-

```
preliminaries:
     /* this step reduces recursion during trace+marking */
     for each active global frame,
          mark the GC bit in each cell;
trace+marking:
     for each local frame
     and corresponding active global frame if any,
          (trace+mark each local cell;
          trace+mark each global cell);
computing displacements:
     for each global frame in ascending order,
          compute its displacement and set mark(N) where
          N := displacement of previous frame
               +number of cells dropped from end of previous frame
               +sizes of any intervening frames discarded
               +number of cells dropped at start of this frame;
remapping of global addresses:
     for each local frame,
          (remap-global-pointer for the V1 field;
          remap each local cell);
     for each global frame,
          remap each global cell;
     for each trail item,
          remap the trailed reference;
     also remap-global-pointers X1,V1,VV1;
compacting the global stack:
     physically move the remaining global frames to their new
     positions, unmarking the GC bit in each cell.

procedure trace+mark(Cell):
     uses a pushdown-list set up in free space at the top of the
     local stack;
     mark the GC bit in Cell;
     if Cell contains a reference to a global cell, Gcell,
    and Gcell is not already marked,
         then trace+mark(Gcell)
    else if Cell contains a molecule
         then trace+mark each unmarked global cell
              for the variables in its skeleton
    else return.

procedure remap(Construct):
    if Construct is a global reference,
         then scan back through the frame to the preceding mark(N)
              and subtract N from the reference
    else if Construct is a molecule
    and there is a variable in its skeleton,
         then find the mark(N) preceding the variable's cell
              and subtract N from the frame field of the molecule
    else return.

procedure remap-global-pointer(Address):
    if the location before Address contains 'discard(N)'
         then subtract N from Address
    else the location contains 'mark(N)' in which case subtract N-M
         from Address where M is the number of unmarked cells
         starting at Address.
```

9.0 DESIGN PHILOSOPHY

Having described the main features of our Prolog implementation, it is perhaps worthwhile to comment on the criteria which influenced design decisions. It is hoped this will provide some answer to inevitable questions of the form "Wouldn't it be better if......?" or "Was it really necessary to......?".

Firstly, software implementation has to be judged by the standards of an engineering discipline rather than as an art or science. One cannot hope to achieve an ideal solution to every problem, but it is essential to find adequate solutions to all the major ones. Generally speaking simplest is best.

A good example is the contrast between earlier attempts to use "theorem provers" as "problem solvers" and Prolog itself. The earlier attempts failed because no adequate solution had been found to the problem of controlling the system in a reasonable way. Although the simple solution adopted by the originators of Prolog does not satisfy all the aspirations of "logic programming", and so is perhaps not "ideal", it does transform logic into an adequate, indeed powerful, programming tool.

In our experience of using Prolog we have not found any example which demands more sophisticated control facilities. Nor have we felt any overwhelming need for extensions to the language. By far the worst practical drawback has been the large amounts of working storage required to run the Marseille interpreter. Also, although interpreted Prolog is fast enough for most purposes, it is too slow for running systems programs such as the Prolog "supervisers". This is a pity since Prolog is otherwise an excellent language for software implementation. Therefore improved efficiency, both of space and time, has been the major aim.

In implementing any language, it is important to have in mind some representative programs against which to check the relevance of design issues and on which to base decisions. For this purpose, we have taken the existing Prolog supervisers and the new Prolog compiler itself, as <u>their</u> efficiency is what matters most to the average Prolog user. Looking at typical Prolog programs such as these, one finds that the full generality of Prolog is brought into play only rarely. At almost every step one is dealing with a special case that can be handled more efficiently. Examples are the following:-

* Many procedures are determinate. We can capitalise on this to recover much of the working storage used.

* Of the symbols which make up the head of a clause (functors, constants and variables), the majority are typically variables, and moreover are typically first occurrences of the variable. We have seen that the code for this important case of the first occurrence of a variable performs a relatively very trivial operation.

* In the source program, the arguments of a goal are almost always variables. Hence the decision to generate executable instructions for terms in the head of a clause rather than those in the body.

* Predicates are usually used in a restricted mode with certain arguments providing procedure input and others receiving procedure output. Optional mode declarations enable the system to avoid generating unnecessary code and also to increase the amount of storage recovered automatically when a procedure exit is determinate.

* The first argument of a predicate is analogous to the subject of a natural language sentence, and it is natural for this argument to be an "input" of the procedure. Often the clauses of the procedure concerned represent different cases according to the principal functor of the term supplied. An efficient treatment of such "definition by cases" is implemented which selects the correct case(s) by table lookup. This feature is invaluable for writing compilers in a natural and efficient way.

* Terms are rarely nested to any degree in clauses responsible for major computation. Hence the decision not to bother to generate executable code for terms nested below level 2.

In short, it is the treatment of such special cases which is the decisive factor in determining efficiency.

The design objectives may be summarised as being aimed towards making Prolog a <u>practicable</u> systems programming language. It was considered reasonable for the systems programmer to have to understand some general facts about how the language has been implemented in order to use it with maximum efficiency. eg. The systems programmer is expected to be aware of when his clauses can be compiled into a table lookup and to appreciate the need for mode declarations. However, as far as the naive programmer is concerned, none of this knowledge is necessary to write correct programs.

In most conventional programming languages, it is difficult to separate the essentials of program design from the details of efficient implementation. One cannot state one without the other. For example, PL/1 faces the programmer with choosing, at the outset, the storage class of his data. The choice strongly affects the form of the program. Similarly most languages have mandatory types for all data items and the programmer cannot easily change a data type once "coding" has commenced. This even applies to more high-level languages such as Lisp, where all "abstract" data structures have to be mapped into concrete list structures. It is difficult to avoid becoming committed to referring to some abstract component as CDDAR say.

The approach we favour is to specify an algorithm as an essential core, to which extra pragmas (pragmatic information) are added. The pragmas need not be supplied until a later stage and give guidance on how the core is to be implemented efficiently. They do not affect the correctness of the program. An example of a pragma is the predicate mode declaration supported by this implementation. There are numerous other possibilities in the same vein which could make logic based programs more efficient, while preserving the simplicity and ease of use of the core language.

For example, more sophisticated clause indexing is clearly needed in some cases, yet it is unrealistic to expect the system to arrive at the optimal choice since, among other things, it depends on how the clauses are going to be used. Plainly there is scope for the programmer to give guidance through some new form of pragma.

10.0 PERFORMANCE

10.1 Results

Some simple benchmark tests to assess Prolog performance are presented in Appendix 5. The other languages chosen for comparison are Lisp and Pop-2. The three languages have similar design aims and can usefully be compared. All are intended for interactive use, and are paricularly oriented towards non-numerical applications, with the emphasis on generality, simplicity and ease of programming rather than absolute efficiency. (Also, all are in active use on the Edinburgh DEC10.)

Each benchmark is intended to test a different aspect of Prolog. No fixed criteria were used for selecting the "equivalents" in the other languages, and so each example should be judged on its own merits. One should observe that there is no absolute sense in which the performance of different language implementations can be compared, except where there is a clearly defined correspondence between the programs of the two languages.

In the case of Prolog, Lisp and Pop-2, there is a subset of each for which there is a fairly obvious, objectively defined correspondence, namely the class of procedures which compute simple functions over lists. This correspondence is illustrated by the first benchmark, a "naive" procedure for reversing a list. This procedure is useful as a benchmark simply because it leads to heavy "list crunching". The time ratios quoted are typical of the class. Thus it is usual for compiled Prolog procedures which compute simple list functions to run at 50-70% of the speed of the Lisp equivalents, for example.

The second benchmark is a "quick-sort" algorithm for sorting lists. The auxiliary procedure 'partition' shows the worth of multiple output procedures. For comparison, we have selected a Lisp version which packages the two outputs into a list cell. Nested lambda expressions are required for the unpacking. The Pop-2 version is taken from p.235 of the Pop-2 handbook [Burstall et al. 1971], omitting the refinement which caters for non-random input lists. Thus we have essentially the same algorithm as the Prolog and Lisp versions, but with gotos and explicit stack manipulation replacing normal function calls. This transformation makes the function rather difficult to understand, although evidently it improves the speed. It is interesting to note that the more transparent Prolog formulation is also appreciably faster.

The third benchmark is a much favoured example of non-numerical programming - the differentiation of an algebraic expression. The Lisp version is a slight extension of Weissman's [1967, p.167] DERIV function and the Pop-2 form is likewise extended from an example on p.26 of the Pop-2 handbook. The Prolog formulation is concise and echoes the textbook equations in a way which is immediately apparent. It demonstrates the advantages of general record structures manipulated by pattern matching where the record types do not have to be explicitly declared. Moreover the timing data shows that the Prolog version is fastest. Notice how the Prolog speed is most marked in cases where a lot of data structure is created, eg. when a quotient is differentiated. This characteristic is a result of structure-sharing and will be discussed later.

The fourth benchmark was chosen to test the implementation of the logical variable, and was suggested by the kind of processing which is typical of a compiler. The task is to translate a list of symbols (here actually numbers) into a corresponding list of serial numbers, where the items are to be numbered in "alphabetical" order (here actually numerical order). The 'serialise' procedure pairs up the items of the input list with free variables to produce both the output list and an "association list". The elements of the association list are then sorted and their serial numbers computed to complete the output list. For comparison we show a Lisp implementation which attempts as far as possible to satisfy the conflicting aims of paralleling the Prolog version and remaining close to pure Lisp. The main trick is to operate on the cells of a duplicate list, eventually overwriting the copied elements with their serial numbers. The choice of a Pop-2 version seems even more arbitrary and we have not attempted to provide one.

The final benchmark is designed to test the improvement gained by indexing the clauses of a procedure. The task is to interrogate a "database" to find countries of similar population density (differing by less than 5%). The database contains explicit data on the areas and populations of 25 countries. A procedure 'density' fills in "virtual data" on population densities. As is to be expected, the speed advantage of compiled code is considerably enhanced relative to either Prolog interpreter, neither of which indexes clauses within a procedure. Thus the benefit of compilation is a factor of around 50 instead of the normal 15 to 20. The figures for the 'deriv' example show a similar but less pronounced effect. To illustrate the correspondence between backtracking in Prolog and iterative loops in a conventional language, we show a Pop-2 version of the database example. The demographic data is stored in Pop-2 "strips" (primitive one-dimensional fixed-bound arrays), and the ′query′ clause translates into two nested forall loops. As the timing data shows, the speed of Prolog backtracking can better that of a conventional iterative formulation.

We shall now summarise the results of these benchmark tests and other less direct performance data. Firstly, comparing Prolog implementations, one can say that compilation has improved running speed by a factor of (typically) 15 to 20 relative to the Marseille interpreter. The improvement is greater where clause indexing pays off, and somewhat less in certain cases where terms are nested deeper than level 2 in the head of a clause. The speed of our Prolog interpreter implemented in Prolog is very similar to that of the Marseille interpreter, and their times are remarkably consistent. {In fact, our interpreter could be much faster if the present clumsy method for interpreting the "cut" operator were avoided, eg. through provision in the compiler of "ancestral cut", ie. a "cut" back to an ancestor goal instead of the immediate parent.}

The results of comparing Prolog with a widely used Lisp implementation may be summarised as follows. For computing simple functions over lists, compiled Prolog typically runs no more than 30-50% slower than pure Lisp. 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. In cases where a wider range of data types than simple lists is really called for (or where "conses" outnumber ordinary function calls), Prolog can be significantly faster. For what it is worth, the mean of the 4 common benchmarks (taking only the 'ops8' figures for 'deriv') puts Prolog speed at 0.75 times that of Lisp.

As regards Pop-2, in all the benchmark tests compiled Prolog ran at least 60% faster, even where the Pop-2 version was formulated using more primitive language constructs such as gotos and "strips". The mean for the 4 common benchmarks (again taking the 'ops8' data) puts Prolog 2.4 times faster than Pop-2.

Small benchmark tests can only give a partial and possibly biassed indication of efficiency; an implementation is better evaluated from the performance of large-scale programs. On these grounds it is perhaps useful to look into the performance of the Prolog compiler. Recall that the compiler is itself implemented in Prolog (and furthermore is almost entirely "pure" Prolog, ie. clauses having a declarative semantics). In practice compilation proceeds in two phases, with DEC's MACRO assembler being used for the second phase:-

```
Prolog      Prolog         Assembly                   Relocatable
source      compiler       language      MACRO        code
file    --------------->   file      -------------->  file
            (Phase 1)                    (Phase 2)
```

The ratio of the times for Phase 1 : Phase 2 is usually of the order of 3 to 2. It is surprising the times are not more different, since Phase 2 is a relatively simple process, and the MACRO assembler is commercial software implemented in a low-level language. The compiler is only generating about 2 instructions for each Prolog source symbol, so it is not simply a case of Phase 1 creating voluminous input to Phase 2. An average figure for the compilation speed of the Prolog compiler (Phase 1 only) is 10.6 seconds per 1000 words of code generated. This includes input of the source file and output of the assembly language file.

So far we have only discussed performance in terms of speed. From an historical point-of-view, space economy has been of far more concern to the Prolog user, and accordingly was a major objective of this implementation. It is therefore important to assess how effective the new space-saving techniques have been. From the nature of the techniques, an improvement will only obtain for determinate procedures (apart from an overall 2-fold improvement due simply to tighter packing of information into the machine word), so much depends on how determinate programs are in practice. The compiler itself, a highly determinate Prolog program, now rarely requires more than 5K words total for the trail and two stacks. When the compiler was interpreted by the Marseille interpreter (before it would "bootstrap"), 75K words was not really adequate for the whole system, of which roughly 50K would be available as working storage. This suggests approximately a 10-fold space improvement for determinate programs.

It is difficult to make more direct comparisons with either the Marseille interpreter or the Lisp and Pop-2 systems, and we have not attempted to do so. Firstly none of these systems provides an easy means of determining how much working storage is actually in use (as opposed to available for use). Secondly it is debatable what measurements should be used to compare systems having different storage allocation regimes, especially where memory is paged. For example, how much free storage is "necessary" in a system relying on garbage collection? {The fairest proposal might be to ascertain and compare, for each benchmark, the smallest amount of non-sharable physical memory in which the test will run without degrading performance by more than a certain percentage. This would be a tedious task.}

It is probably fair to say that the "average" compiled Prolog program requires considerably more working storage than Lisp or Pop-2 equivalents, but that with careful and knowledgeable programming (using mode declarations and ensuring determinacy) the Prolog requirement need not be much different from the other two. (For example, it is doubtful whether a Lisp or Pop-2 implementation of the Prolog compiler would use less storage.) The difference between Prolog and the other two is likely to be of less practical significance on a virtual memory machine. The extra storage required by Prolog typically represents groups of "dead" environments which are not in active use, and which are also adjacent in memory by virtue of the stack regime. Therefore they can generally be paged out.

From the coding of PLM instructions detailed in Appendix 2, we see that the compiled code is relatively compact at about two words per source symbol. For the record, the "high-segment" sizes of our compiler and interpreter are respectively 25K words and 14K words. These sizes represent the total sharable code including essential run-time system.

10.2 Discussion

The above results show that Prolog speed compares quite well with other languages such as Lisp and Pop-2. Also the performance of the compiler suggests that software implemented in Prolog can reach an acceptable standard of efficiency.

Now on the face of it, a language such as pure Lisp offers simpler and more obviously machine-oriented facilities. How is it that Prolog is not considerably slower?

The first point to notice is that Prolog extras - the full flexibility of unification with the logical variable and backtracking - lead to very little overhead when not used, provided the program is compiled. For example, consider the code generated for the 'concatenate' procedure (cf. Appendix 5.1) and assume it is called, as for the corresponding Lisp function, with two arguments ground (ie. terms containing no variables) and a variable as third argument. All unification on the first two arguments of 'concatenate' reduces to simple type checks and direct assignments. Unification on the third argument is somewhat more costly, as it is creating the new output list (cf. the "conses" performed by the Lisp procedure). If indexed procedure code is generated, the Prolog machine readily detects that it is executing a determinate procedure and there are no significant overheads attributable to "backtracking" -- the trail is never accessed and all local storage is automatically recovered on procedure exit. In short, the procedure is executed in much the same manner as one would expect for a conventional language.

Despite this, it is still surprising that Lisp is not several times faster than Prolog. Lisp has only the one record type and, more importantly, it does 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 are effectively hardware instructions on the DEC10.

In analysing the reasons for Prolog's relative speed, we are led to the following, perhaps unexpected, conclusions:-

(1) Specifying operations on structured data by "pattern matching" is likely to lead to a better implementation than use of conventional selector and constructor functions.

(2) On a suitable machine, the "structure-sharing" representation for structured data can result in faster execution than the standard "literal" representation. To be more specific, it allows a "cons" to be effected faster than in Lisp.

To illustrate the reasons for these conclusions, let us compare (a) an extract from the definition of evalquote given in the Lisp 1.5 Manual [McCarthy et al. 1962] with (b) the clause which is its Prolog counterpart. We shall write the Prolog functor corresponding to cons as an infix operator '.' :-

```
(a)  apply[fn;x;a] =
       ...
       eq[car[fn];LABEL] -> apply[caddr[fn];x;
                                  cons[cons[cadr[fn];caddr[fn]];a]]
       ...


(b)  apply(label.Name.Form._,X,A,Result) :-
       apply(Form,X,(Name.Form).A,Result).
```

As an aside to our main argument, we may first of all observe that "pattern matching" makes it much easier to visualise what is happening. The pattern matching version also invites a better implementation. No location corresponding to the variable 'fn' needs to be set aside and initialised. It is only the form and subcomponents of this argument which are of interest. The decomposition is performed initially once and for all by pattern matching. In contrast, a straightforward implementation of the Lisp version will duplicate much of the work of decomposition. The double occurrence of caddr is the most noticeable cause, but we should also remember that caddr and cadr share a common step.

A more technical consideration is that pattern matching encourages better use of index registers. A pointer to the structured object is loaded just once into an index register and held there while all the required subcomponents are extracted. Unless the Lisp implementation is quite sophisticated it will be repeatedly reloading the value of 'fn', and subcomponents thereof. A related issue concerns run-time type checks needed in languages like Pop-2. (Lisp manages to avoid such checks for the reasons noted above.) An unsophisticated implementation of selector functions will have to perform a type check before each application of a selector. With pattern matching, one type check suffices for all the components extracted from an object.

Finally, for procedures such as 'apply' above, pattern matching also encourages the implementation to integrate type checking with case selection, building in computed gotos where appropriate.

To summarise, not only is pattern matching more convenient for the user, it also leads the implementor directly to an efficient implementation:-

(1) Procedure call and argument passing are no longer just "red tape" - they provide the context in which virtually all the "productive" computation is performed.

(2) No location needs to be set up for an argument unless it is explicitly referred to by name.

(3) One can select <u>all</u> the required components of a compound object in one efficient process using a common index register.

(4) Type checking is performed once and for all at the earliest opportunity.

(5) It is easier for the implementation to replace a sequence of tests with a computed <u>goto</u>.

Hoare [1973] has proposed a more limited form of "pattern matching" for an Algol-like language and has advanced similar arguments for its clarity and efficiency.

Let us now consider the impact of structure-sharing on efficiency. Ironically, this technique was first devised by Boyer and Moore as a means of saving space. However we shall argue that it is even more important for its contribution to Prolog's speed.

Clearly the direct representation of a compound data object, as used in Lisp implementations and for source terms in Prolog, would enable somewhat faster access to components. However, the representation in our DEC10 implementation of a source term variable by an indexed address word means that each argument of a constructed term can likewise be accessed in just one machine instruction.

(Further dereferencing is sometimes needed, but this is comparatively rare in practice.) Thus the only significant accessing overhead for structure-shared objects is the necessity for preliminary loading of the frame component of a molecule into an index register. The great advantage of structure-sharing lies in the supreme speed with which complex new objects are created, and also the ease with which they can be discarded when no longer needed.

To see this, let us return to our evalquote example. The Lisp version has to perform two "conses" to construct the third argument of the call to apply. Each "cons" involves:-

(1) grabbing a new free cell, after checking that the free list is not exhausted;

(2) copying each component into the list cell obtained;

(3) saving the address of the new cell.

If, as Prolog, Lisp allowed more than one record size, step (1) would have to be a lot more complex.

In contrast, Prolog has to perform absolutely no work to construct the third argument of the call to 'apply'! ie. No executable code is generated for the term '(Name.Form).A'. Well, this is slightly misleading since the analogous computation will in fact occur during the next invocation of 'apply', when unification creates a new molecule to bind to the next generation of 'A'. However, creating this molecule merely involves bringing together two existing pointers as the halves of the word to be stored in 'A's cell.

The difference between the two methods can be summarised as follows. Languages like Lisp assemble the information to construct a new object on a stack (local storage), and then copy the information into special records individually obtained from heap storage. Prolog leaves the information in situ on the stack(s) and relies on structure-sharing for later procedures to locate the information as needed. Prolog is substituting extra indirection, which is very fast, for the relatively slow operations of copying and heap management. The Prolog cost of constructing new objects from a set of skeletons in a clause is, at worst, proportional to V, the number of distinct variables in the skeletons. The cost for conventional methods is at least proportional to S, the total number of symbols in those skeletons. V can't be any greater than S, and is often much smaller. The smaller V is, the more advantageous the Prolog method.

Another point to notice is that each Lisp cell "consed" up must ultimately be reclaimed by the expensive process of garbage collection. In tight situations, a garbage collecting system can "thrash", spending nearly all its time on garbage collection and little on useful work. It is for this reason that systems programmers prefer not to rely on garbage collectors. With Prolog, the user can usually rely on the stack mechanism associated with backtracking to recover all storage at negligible cost. This advantage is, again, even greater if one considers the complexities of garbage collection in other languages admitting more than one size of record.

A final point is that the stack regime leads to better exploitation of virtual memory, since, as noted above, it avoids the random memory accesses inevitably associated with "heap" management.

11.0 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, from both the user's and the implementor's point of view. This is especially so where many user-defined record types are allowed.

For "symbol processing" applications where a transparent and easy-to-use language is required, Prolog has significant advantages over languages such as Lisp and Pop-2. Firstly the Prolog program is generally easier to understand, mainly because it is formulated in smaller units which have a natural declarative reading. Secondly Prolog allows a wider range of problems to be solved without resort to machine- or implementation-oriented concepts. The logical variable and "iteration through backtracking" go a long way towards removing any need for assignment in a program. Finally our implementation shows that these advantages can be obtained with little or no loss of efficiency. In fact in many cases the distinctive features of Prolog actually promote better implentation.

12.0 ACKNOWLEDGEMENTS

I am indebted to members of the Groupe d'Intelligence Artificielle, Marseille, especially Alain Colmerauer and Philippe Roussel. They were entirely responsible for conceiving the Prolog language and developed the fundamental implementation techniques.

My colleagues, Luis Pereira and Fernando Pereira, have provided great encouragement throughout this project, and took on much of the hard work which is needed to make an implementation practically usable. In particular, Fernando Pereira was responsible for implementing the garbage collector and routines to adjust the sizes of the main areas automatically during execution.

Gottfried Eder was an early "guinea-pig" user, and happily survived.

Keith Clark and Sten-Ake Tarnlund spurred me to write this report and many people gave helpful comments on earlier drafts.

The work was inspired by the ideas of Robert Kowalski, and was directed and encouraged by Prof. Bernard Meltzer with support from an SRC research studentship and SRC grant B/RG 9972.

13.0 REFERENCES

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

Bergman M and Kanoui H [1975]
Sycophante: Systeme de calcul formel et d'integration symbolique sur ordinateur.
Groupe d'Intelligence Artificielle, Marseille-Luminy. Oct 1975.

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

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

Bundy A, Luger G, Stone M and Welham R [1976]
MECHO: year one.
DAI Report 22, Dept. of AI, Edinburgh. Apr 1976.

Burstall R M, Collins J S, Popplestone R J [1971]
Programming in Pop-2.
Edinburgh University Press. 1971.

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

Dahl V and Sambuc R [1976]
Un systeme de banque de donnees en logique du premier ordre, en vue de sa consultation en langue naturelle.
Groupe d'Intelligence Artificielle,Marseille-Luminy. Sep 1976.

DEC [1974]
DECsystem-10 System Reference 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.

van Emden M H [1976]
Deductive information retrieval on virtual relational databases.
Report CS-76-42, Dept. of Computer Science,
University of Waterloo, Canada. Aug 1976.

Hoare C A R [1973]
Recursive data structures.
Stanford AI Memo 223, Calif. Nov 1975.

Kowalski R A [1974]
Logic for problem solving.

DCL Memo 75, Dept of AI, Edinburgh. Mar 74.

Lichtman B M [1975]
Features of very high-level programming with Prolog.
MSc dissertation, Dept.of Computing and Control,
Imperial College, London. Sep 1975.

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

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 : Manuel de reference et d'utilisation.
Groupe d'Intelligence Artificielle, Marseille-Luminy. Sep 1975.

Sussman G J and Winograd T [1970]
MICRO-PLANNER reference manual.
AI Memo 203, MIT Project MAC. Jul 1970.

Warren D H D [1974]
Warplan : a system for generating plans.
DCL Memo 76, Dept. of AI, Edinburgh. Jun 1974.

Warren D H D [1976]
Generating conditional plans and programs.
Procs. AISB Conf., pp.344-354, Edinburgh. Jul. 1976.

Warren D H D [1977]
Compiler writing and logic programming.
forthcoming report, Dept. of AI, Edinburgh. 1977.

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

Zloof M [1974]
Query by Example.
RC 4917 (21862), IBM Thomas J Watson Research Centre,
Yorktown Heights, New York 10598. 1974.