THE UNIVERSITY *of* EDINBURGH

APPLIED LOGIC

- its use and implementation as a programming tool

by

David H D Warren

PhD Thesis, University of Edinburgh, 1977

1.0 CONTENTS

2.0 ABSTRACT

The first Part of the thesis explains from first principles the concept of "logic programming" and its practical application in the programming language Prolog. Prolog is a simple but powerful language which 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).

The ideas are illustrated by describing in detail one sizable Prolog program which implements a simple compiler. The advantages and practicability of using Prolog for "real" compiler implementation are discussed.

The second Part of the thesis describes techniques for implementing Prolog efficiently. In particular it is shown how to compile the patterns involved in the matching process into instructions of a low-level language. This idea has actually been implemented in a compiler (written in Prolog) from Prolog to DECsystem-10 assembly language. However 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.

It is argued 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

Logic, programming, Prolog, implementation, compiler, data structures, matching, unification, compiler specification, compiler implementation.

3.0 PREFACE

Historically, the idea of a computer preceded the idea of a programming language. In the early days, a program was seen as a means of instructing a particular machine to carry out some task. Programming languages then evolved in response to the need to make instruction of the machine easier for human programmers. With hindsight, it seems clear that this approach is back-to-front. Really we should first decide what is needed in a programming language and then tailor the machine to fit the language, not vice versa. As Dijkstra puts it [1976, p.201], instead of "the program's purpose being to instruct our computers", it should be "the computer's purpose to execute our programs".

Nevertheless, at present it still remains the case that the nature of the conventional programming language owes much to characteristics of early computers - characteristics which are arguably an historical accident and are not essential to the notion of a program. For example, would the assignment operation be so predominant in our programming languages if the first programs had been intended for execution by clerks with pen and paper? - for assignment presupposes very particular properties of the medium used for storing information.

Freeing the design of a programming language from *a priori* machine constraints, what are the qualities needed? In other words, what is the best way for the human programmer to express the information processing task he wishes to be carried out?

One obvious snap answer is to observe that all humans start out knowing at least one "programming language" - their mother tongues. How about natural language as a basis for writing programs? (This idea should not be confused with what Hobbs [1977] describes as "spreading a thin veneer of English vocabulary and perhaps some English syntax over a very ordinary programming language", as for example in the design of Cobol.)

It should be clear that the way information is expressed in natural language is very different from that of the average programming language. (For concreteness, the reader may like to compare a description in everyday English of the rules establishing a person's tax liability with, say, a PL/1 program to calculate the liability.)

Firstly, note that in natural language information can be supplied in a fairly piecemeal fashion, and generally each sentence makes sense in isolation, without a great deal of contextual knowledge. By contrast, conventional program "statements" need to be rigidly sequenced, and the meaning of each "statement" can only really be appreciated in the context of the program as a whole, or at least some sizable portion of it.

Secondly, conventional program "statements" are imperative, whereas in natural language the usual mode of expression is declarative - even when the communication is essentially an instruction, it is normal to give a declarative description of what is required, with just one imperative (often implicit) "Do it!".

Finally, it is normal for the objects referred to in a conventional program to be items associated with the machine or environment in which the program is to be executed - items such as memory locations ("variables") and files. In natural language, of course, there is no need to refer to anything other than the objects or entities directly involved in the subject at hand (such as, in our example, "salaries", "dependents", "investments", etc.).

These differences we have listed seem to be quite fundamental, and it is reasonable to argue that the conventional programming language is unsuited to the human user, precisely because the way information has to be expressed is so different from the natural one.

Of course, natural language itself has obvious disadvantages which make it impractical as a programming language (for the time being at least) - it is too vague, too prone to ambiguity and too long-winded (and parsing the language is far from trivial).

Is it possible to find a language which has the precision and conciseness needed of a programming language, but which is closer to the way we naturally communicate information? One approach meeting these criteria is the subject of this thesis. The main idea goes under the name "logic programming", and has an interesting history.

Over 2300 years ago, Aristotle and his followers initiated the study of reasoning expressed in natural language. They categorised various forms of argument which constitute valid reasoning. This science became known as logic. In the last 100 years, the more comprehensive analysis needed especially for mathematical reasoning

has led to the development of symbolic logic. Instead of working with the natural language form of the argument, logicians now use an artificial language which amounts to a convenient shorthand for natural language. (This symbolic language, like the original discipline, is referred to as "logic".) One can mechanically translate logic statements into (somewhat stilted) natural language statements. All that is needed is a suitable "interpretation" (ie. natural language phrase) for each different symbol used. Some of the symbols must have a fixed translation - these are the "logical" symbols, translated as "if", "and", "all", etc. Then any reasoning which is valid in the formal language makes correct sense when translated into natural language.

{There is (as yet) no mechanical way of doing the reverse translation - that is, from natural language into logic. However, it is not implausible that every natural language statement is equivalent in meaning to some alternative (more stilted) form, where that form results from translating a logic statement under some fixed "interpretation". If this conjecture is correct, logic does indeed constitute a comprehensive shorthand for natural language.}

In this decade, it was realised (by Kowalski and Colmerauer) that symbolic logic provides the basis for a practicable programming language. Logic statements - traditionally given solely a declarative interpretation - can also be understood imperatively (or "procedurally"). This is analogous to the way in natural language we give instructions merely by describing what we want. To turn logic into a workable programming language, it is necessary to supply

additional control information, in order to direct how the logic "program" is to be used to derive the results required. For many practical purposes, an extremely simple form of control information is sufficient. A certain subset of logic, augmented with this particularly simple form of control information, constitutes a programming language known as Prolog.

As a programming language, Prolog has all the nice properties of natural language we listed earlier - the statements are independently meaningful and can be supplied piecemeal; the language can be interpreted both declaratively and imperatively; the objects a program is "about" are independent of any execution environment. Moreover, the "symbolic" character of symbolic logic gives Prolog the essential properties which natural language lacks - conciseness, precision and a simple syntax.

However all this might only be of academic interest. What is significant and truly amazing about Prolog is that it can really be used to write useful programs, and that the efficiency of the implementation can compare quite favourably with that of conventional languages.

It is important to understand that, amongst all the possible meaningful groupings of logic statements, only a very small proportion can be considered reasonable programs, even with the most sophisticated control imaginable, still less with the simple control used in Prolog. Logic statements which form a perfectly good specification of a problem do not necessarily amount to an acceptable implementation. Logic programming <u>is</u> programming. The logic

programmer still has to formulate his task with an eye for what is practicable and reasonably efficient. However, compared with conventional languages, program and specification are much closer in nature, and may even, in favourable cases, be virtually the same.

This thesis is entitled "Applied Logic" to draw attention to the fact that the approach to logic required in logic programming is rather different from the traditional one. This difference can be likened to the distinction between pure and applied mathematics. Traditionally, logic has been studied almost exclusively for its own sake, as a subject of intrinsic intellectual interest. Here, however, we are interested in logic primarily in so far as it can be useful. We look on logic as a tool to be used in tackling information processing problems from a variety of different fields. Whereas there is a vast literature on "pure" logic, only a very simple and non-technical understanding of logic is necessary for our purposes. In a similar way, one doesn't need an awareness of erudite theorems of number theory to make simple use of arithmetic in everyday life.

The thesis consists of two separate and self-contained parts which can read independently in either order. (To achieve this independence, there is some duplication of material.)

Part I of the thesis gives a user's eye view of logic as a programming language, concentrating in particular on Prolog. We aim to show how Prolog can be used as a beautiful and practical tool for writing useful programs. Now other authors have described the advantages of Prolog in a wide variety of different applications, including large-scale programs in the fields of:-

* natural language understanding systems [Colmerauer 1975]

* algebraic symbol manipulation [Bergman & Kanoui 1975] [Bundy et al. 1976]

* computer-aided architectural design [Markusz 1977]

* drug design applications [Darvas et al. 1976,1977]

* database interrogation [Dahl & Sambuc 1976]

* plan/program synthesis [Warren 1974,1976]

So we concentrate on one particular application - compiler writing - and explain, in some detail, a sizable Prolog program. We hope this example will give the reader a better perspective of the language than can be gleaned from selected "tit-bits". Also, it happens that Prolog is remarkably well adapted to the role of "compiler-compiler" - this is remarkable in that Prolog was not developed specifically for this purpose. In our view, the compiler writing application alone is of sufficient interest to make Prolog worthy of study.

Part II of this thesis turns to the implementor's view of Prolog. We show how the logic statements of Prolog can be viewed as sequences of instructions for quite a low-level machine and, using this insight, we indicate how Prolog can be compiled into efficient code for a conventional computer.

Although Parts I and II of the thesis are written to be self-contained, each has bearing on the other:-

(1) Logic would be of little interest as a programming language were it not for the possibility of efficient implementation.

(2) The application of Prolog to compiler writing (and more generally as a systems programming language) motivates the development of more

efficient implementations, such as we describe in Part II.

(3) The method of compiling Prolog described in Part II is implemented by a compiler written in Prolog, using the techniques exemplified in Part I.

Part I of the thesis was previously published under the title "Logic Programming and Compiler Writing" as Edinburgh DAI Research Report No. xx. Part II of the thesis was previously published under the title "Implementing Prolog" in two volumes as Edinburgh DAI Research Reports Nos. 39 & 40. Some of the results were reported in a paper with Luis Pereira and Fernando Pereira entitled "Prolog - the language and its implementation compared with Lisp", given at the ACM SIGART-SIGPLAN symposium on "AI and Programming Languages", Rochester NY, August 1977.

4.0 ORIGINALITY AND ORIGIN OF THE WORK

As is noted in the text, this thesis describes work building on much previous research by other people. The main original contributions are the following:-

(1) (a) The thesis introduces the novel concept of compiling logic programs into efficient machine-oriented instructions.

(b) This idea has been developed into a detailed method for compiling Prolog programs (the "Prolog Machine").

(c) The method has been implemented in a practical and useful Prolog compiler for the DECsystem-10 computer.

(2) The compiler was implemented in Prolog - providing for the first time a demonstration of the practicability of Prolog as a language for compiler writing.

As is acknowledged elsewhere, the Prolog system described herein was implemented in collaboration with two colleagues, Fernando Pereira and Luis Pereira. However, the material covered specifically in this thesis was my work alone, apart from one exception noted below. In particular, I was responsible for designing the basic "Prolog Machine", and also did the implementation of both the compiler itself and the fundamental run-time routines supporting the "Prolog Machine Instructions". The thesis does not attempt to cover many aspects of the implementation which were essential for a practical, usable system and which in fact constituted a major part of the total man hours spent. These aspects are not covered, since no principles peculiar to Prolog are involved. Mainly this work amounted to "human engineering", provided typically in the form of "evaluable

predicates". A large portion of this work was undertaken by my colleagues, particularly the parts involving interface to the host machine's operating system (Monitor). Of the material covered in the thesis, the only part which was substantially collaborative was the garbage collector - a basic design of this component by me was "debugged" and implemented by FP.

5.0 ACKNOWLEDGEMENTS

I am indebted to the originators of Prolog and logic programming in general. The concept of the Prolog language was entirely the work of members of the Groupe d'Intelligence Artificielle, Marseille, (especially Alain Colmerauer and Philippe Roussel), and they also developed the fundamental implementation techniques.

Furthermore, the possibilities of Prolog for compiler writing were first recognised by Alain Colmerauer [1975, Chap.4], and the compilation method I describe is (necessarily) very similar in its essentials to the example he describes. The elegant assembly technique is directly based on his work.

My colleagues, Luis Pereira and Fernando Pereira, gave much encouragement and practical assistance in implementing the Prolog system referred to in the thesis. In particular, Fernando Pereira was responsible for implementing the garbage collector and routines to adjust the sizes of the main areas automatically during execution.

Thanks are due to Pavel Brazdil, Derek Brough, Keith Clark, Gottfried Eder, Chris Mellish, Fernando Pereira, Luis Pereira, Sten-Ake Tarnlund and Austin Tate for useful discussions and/or helpful comments on earlier drafts of this thesis. The original impetus for the work came from the ideas and personal encouragement of Robert Kowalski.

The work was supervised formerly by Prof. Donald Michie and Robert Kowalski, and latterly by Prof. Bernard Meltzer, with support from an SRC research studentship.

PART I - LOGIC PROGRAMMING AND COMPILER WRITING

1.0 INTRODUCTION

This Part of the thesis aims to provide an introduction to the concept of "logic programming" [Kowalski 1974] [Colmerauer 1975] [van Emden 1975] for people with experience in other programming languages. The emphasis is on those aspects which have been put to practical use in the programming language Prolog [Roussel 1975] [Pereira 1977], developed at the University of Marseille. The ideas are illustrated by discussing at length one main example, consisting of a very simple compiler written in Prolog. Although this "toy" compiler has been made as simple as possible for didactic purposes, the techniques employed are taken from a "real" implementation in Prolog of a compiler in practical use. This example has been chosen with the additional purpose of demonstrating the particular advantages of Prolog for compiler writing. The reader is expected to be broadly familiar with various conventional programming languages, but no knowledge of symbolic logic is assumed. Some acquaintance with the issues involved in writing a compiler would be an advantage.

2.0 LOGIC PROGRAMMING

The principal idea [Kowalski 1977] behind logic programming is that an algorithm can be usefully analysed into a logical component and a control component:-

"algorithm = logic + control".

Roughly speaking, the logical component defines what the algorithm does, and the control component prescribes how it is done efficiently.

The logical component can be expressed as statements of symbolic logic. For this purpose, one normally only needs to consider a restricted part of logic reduced to a standard form known as "Horn clauses". The language of this subset will now be described from a conventional programming standpoint. The notation and terminology will be that used in Prolog.

2.1 Syntax

The data objects of the language are called terms. A term is either a constant, a variable or a compound term.

The constants include integers such as:-

0 1 999

and atoms such as:-

a nil = := 'Algol-68'

The symbol for an atom can be any sequence of characters, which in general must be written in quotes unless there is no possibility of confusion with other symbols (such as variables, integers). As in conventional programming languages, constants are definite elementary

objects, and correspond to proper nouns in natural language.

Variables will be distinguished by an initial capital letter eg.

X Value A Al

If a variable is only referred to once, it does not need to be named and may be written as an "anonymous" variable indicated by a single underline character:-

_

A variable should be thought of as standing for some definite but unspecified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writeable storage location as in most programming languages. Compare instead the variable of pure Lisp, which is likewise a "stand-in" for a data object rather than a location to be assigned to.

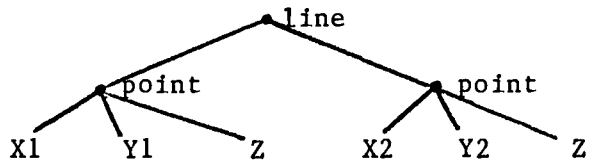
The structured data objects of the language are the compound terms. A compound term comprises a functor (called the principal functor of the term) and a sequence of one or more terms called arguments. A functor is characterised by its name, which is an atom, and its arity or number of arguments. For example the compound term whose functor is named 'point' of arity 3, with arguments X, Y and Z, is written:-

point(X,Y,Z)

Functors are generally analogous to common nouns in natural language. One may think of a functor as a record type and the arguments of a term as fields of a record. Compound terms are usefully pictured as trees. For example, the term:-

line(point(X1,Y1,Z),point(X2,Y2,Z))

would be pictured as the structure:-



Sometimes it is convenient to write a compound term using an optional infix notation, eg.

X+Y (P;Q)

instead of:-

+(X,Y) ;(P,Q)

Finally note that an atom is treated as a functor of arity 0.

Suppose we wish to give a formal definition of a data type called a "dictionary". A dictionary will be either the atom 'void', or a compound term of the form:-

dic(<term 1>,<term 2>,<term 3>,<term 4>)

where the arguments <term3> and <term4> are also dictionaries whilst <term1> and <term2> are of unrestricted type. {Here, and throughout this Part of the thesis, names in angular brackets are used as "meta-variables" to symbolise constructs of the "object language" being discussed, cf. the non-terminal symbols of a Backus-Naur form (BNF) grammar.} The required definition of the data type "dictionary" is expressed in logic by the following two statements:-

dictionary(void).

dictionary(dic(X,Y,D1,D2)) :- dictionary(D1),dictionary(D2).

Here 'dictionary(_)' is a special kind of functor called a predicate, analogous to a verb in natural language. (Predicates are distinguished from other functors only by the contexts in which they occur.) A term with a predicate as principal functor is called a

boolean term, and is analogous to a simple statement in natural language.

In general, statements of logic can be considered to be a shorthand for descriptive statements of natural language. A statement of the form:-

<P> :- <Q>, <R>, <...>

should be read as:-

<P> if <Q> and <R> and <...>

Thus the two statements above might be read as:-

"void" is a dictionary.
"dic(X,Y,D1,D2)" is a dictionary if D1 is a dictionary and D2 is a dictionary.

Any variables in a statement are interpreted as standing for arbitrary objects, so a more precise reading of the second statement would be:-

For any X, Y, D1 and D2, "dic(X,Y,D1,D2)" is a dictionary if D1 is a dictionary and D2 is a dictionary.

Note that the variables in different statements are completely independent even if they have the same name - ie. the "lexical scope" of a variable is restricted to a single statement.

The kind of logic statements we are considering are called clauses. For our purposes, a clause comprises a head and a body. The head is a boolean term and the body is a sequence of zero or more boolean terms called goals. In general a clause is written:-

<head> :- <goal 1>, <goal 2>, <...>.

If the number of goals is zero, we speak of a unit clause, and this is written:-

<head>.

2.2 Semantics

The semantics of the language we have described should be clear from its informal interpretation. However it is useful to have a precise definition. The semantics will tell us which boolean terms can be considered true according to some given clauses. Thus in the case of our clauses for 'dictionary(_)', we shall know that a term <term> is a dictionary if the boolean term:-

dictionary(<term>)

is true.

Here then is a recursive definition of what will be called the declarative semantics of clauses.

> A term is true if it is the head of some clause instance and each of the goals (if any) of that clause instance is true, where an instance of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

The unary predicate 'dictionary(_)' specified a data type. More generally, predicates are used to express relationships between objects. For example, we might use 'concatenated(<1>,<2>,<3>)' to mean that list <3> consists of the elements of list <1> followed by the elements of list <2>. Thus

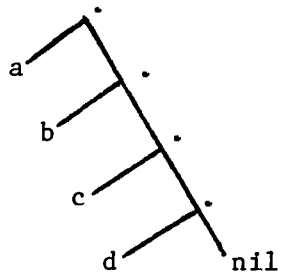
```
concatenated((a.b.c.d.nil),
             (1.2.3.nil),
             (a.b.c.d.1.2.3.nil))
```

is true, where a list is either the atom 'nil' or a term formed from the binary functor '.' whose second argument is a list, ie.

```
list(nil).
list(.(X,L)) :- list(L).
```

In general, as above, we write the functor '.(_,_)' as a

right-associative infix operator so that, for example, the first list mentioned is equivalent to the standard form '.(a,.(b,.(c,.(d,nil))))' and should be pictured as:-



The following clauses define the predicate 'concatenated(_,_,_)':-

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

The clauses may be read as:-

> The empty list concatenated with any list L is simply L.
> A non-empty list consisting of X followed by remaining elements L1 concatenated with list L2 is the list consisting of X followed by remaining elements L3 where L1 concatenated with L2 is L3.

So far we have looked on clauses as a means of <u>specifying</u> relationships between objects. This is the traditional view of the purpose of logic.

Now consider what has to be done for relationships expressed in logic to be computed efficiently. For example, given terms <term1> and <term2>, how can one find a term <term3> such that:-

concatenated(<term1>,<term2>,<term3>)

is true? The major discovery of "logic programming" is that the clauses themselves can often provide the basis of the procedures required. In such cases, it is only necessary to supply suitable control information to specify how the clauses are to be used effectively. In brief, logic has a "procedural interpretation".

The procedural interpretation treats a predicate as a procedure name, the head of a clause as a procedure entry point and a goal as a procedure call. A procedure is a set of clauses with the same head predicate. For example, the clauses for 'concatenated(_,_,_)' can be considered to be a procedure for concatenating the elements of two given lists (amongst other uses). The procedure has two entry points corresponding to whether or not the first of the two input lists is empty. One of the clauses makes a recursive call to the same procedure.

Before we go on to consider the kind of control provided in Prolog, we should observe that not all sets of clauses make equally effective procedures. Some clauses would require unrealistically sophisticated control information to be of practical use. Much of the art of logic programming is to formulate the problem in such a way that it can be solved efficiently using the control mechanisms available. This soon comes quite naturally to someone with programming experience, as really it is just what one does in any other programming language; the ingenuity required is no greater, and usually less. Indeed, as we shall see, one of the main attractions of logic programming is that often a natural specification of an algorithm and a good implementation are one and the same.

3.0 THE PROGRAMMING LANGUAGE PROLOG

3.1 Introduction

A remarkably simple form of control suffices for many practical applications of logic programming. This point was first realised at Marseille and is the basis of the programming language Prolog developed there. From now on we shall restrict our attention to Prolog.

If we think back to the declarative semantics of clauses, it is clear that the order of the goals in a clause and the order of the clauses themselves, are both irrelevant to the declarative interpretation. However these orderings are generally significant in Prolog as they constitute the main control information. In other respects a Prolog program is just a set of clauses.

When the Prolog system is executing a procedure call, the clause ordering determines the order in which the different entry points of the procedure are tried. The goal ordering fixes the order in which the procedure calls in a clause are executed. The "productive" effect of a Prolog computation arises from the process of "matching" a procedure call against a procedure entry point.

Really there are two different ways of looking at the meaning of a Prolog program. We have already discussed the declarative interpretation which Prolog inherits from logic. The alternative way is to consider, as for a conventional programming language, the sequence of steps which take place when the program is executed. This is defined by the procedural semantics of Prolog. This semantics will

tell us what happens when a goal (procedure call) is executed. The result of the execution will be to produce true instances of the goal (if there are any). Thus the procedural semantics is governed by the declarative. Here then is an exact description of the procedural semantics.

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

Let us now return to the clauses for concatenated(_,_,_)':-

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

and see how they can be used to concatenate two lists. Suppose we wish to concatenate the lists (a.b.nil) and (1.2.nil). This will be achieved by executing the goal:-

```
concatenated((a.b.nil),(1.2.nil),Z)
```

The result of the execution will be to substitute the required value for the variable Z. The goal matches only the second clause, and becomes instantiated to:-

```
concatenated((a.b.nil),(1.2.nil),(a.Z1))
```

since this is the most general common instance of the original goal and the head of the matching clause. The name given to the new variable Z1 is arbitrary. The body of the matching clause instance gives us a new goal (or recursive procedure call):-

concatenated((b.nil),(1.2.nil),Z1)

The process is repeated a second time giving rise to a further goal:-

concatenated(nil,(1.2.nil),Z2)

which this time matches only the first clause. Execution is now complete as there are no outstanding goals to be executed. The original goal has been instantiated to:-

concatenated((a.b.nil),(1.2.nil),(a.b.1.2.nil))

a true boolean term. Thus the effect of the execution is to instantiate Z to:-

(a.b.1.2.nil)

the term originally sought.

Here we have used 'concatenated(<1>,<2>,<3>)' as a procedure which takes two "inputs" <1> and <2> and returns one "output" <3>. However the procedure is much more flexible than this. For example, if <3> is also provided as input, 'concatenated(_,_,_)' acts as a procedure which checks whether <3> is the concatenation of <1> and <2>. Thus execution of the goal:-

concatenated((a.nil),(b.nil),(a.nil))

will fail whereas:-

concatenated((a.nil),(b.nil),(a.b.nil))

will succeed.

More striking is the behaviour when only <3> is provided as input. For example, consider what happens when the goal:-

concatenated(L,R,(a.b.nil))

is executed. This goal will match both clauses for 'concatenated(_,_,_)'. The first match returns an immediate result:-

L = nil
R = (a.b.nil)

Notice how the result returned consists of two "output" values. If this result is subsequently rejected, backtracking will cause the second possible match for the original goal to be considered. The match instantiates the top goal to:-

concatenated((a.L1),R,(a.b.nil))

and a new goal is produced:-

concatenated(L1,R,(b.nil))

This goal again matches both clauses. The first match produces another solution to the original goal:-

L = (a.nil)
R = (b.nil)

In this way backtracking causes the procedure to generate all possible pairs of lists L and R which, when concatenated, yield (a.b.nil).

These examples have illustrated a number of characteristic features of Prolog procedures. Firstly, when a procedure returns, the result sent back may consist of more than one value, just as, in the conventional way, more than one value may be provided as input. Furthermore, the input and output positions do not have to be fixed in advance and may vary from one call of the procedure to another. In effect, Prolog procedures can be "multi-purpose". These features will play an important part in the compiler which is the main example of Prolog to be discussed later.

3.2 The Logical Variable

The flexibility of Prolog procedures can be seen as a special case of a more general phenomenon. The variable in Prolog behaves in a particularly pleasing way, which is governed by the high-level pattern matching process of unification. Let us consider a simple but somewhat artificial example using the 'concatenated(_,_,_)' procedure. The task is to "treble" a given list to produce a list consisting of three consecutive copies of the original, eg.

(a.b.c.nil) --> (a.b.c.a.b.c.a.b.c.nil)

One way to define this is to say that the list LLL is the treble of the list L if LLL consists of L concatenated with a list LL which is the result of concatenating L with itself. ie.

```
treble(L,LLL) :-
     concatenated(L,LL,LLL),
     concatenated(L,L,LL).
```

In most list processing languages one would have to perform the second step first. That is, the doubled list LL would first be constructed and then another copy of L would be concatenated on the front. The same effect would be achieved in Prolog by expressing the two goals in the opposite order. However the Prolog clause also functions perfectly well as it stands. Let us see how this is, by executing the goal:-

```
treble((a.b.nil),X)
```

Immediately we get the pair of goals:-

```
concatenated((a.b.nil),LL,X),
     concatenated((a.b.nil),(a.b.nil),LL)
```

The first of these goals has an uninstantiated variable as its second argument, but nevertheless the execution proceeds in the familiar way,

recursing twice to hit the bottom of the recursion with the subgoal:-

concatenated(nil,LL,X2)

The net result of executing this final subgoal is that LL is left uninstantiated and the original X is instantiated to:-

X = (a.b.LL)

Thus the result of the original 'treble' goal has been partially constructed, but the value to be returned contains the uninstantiated variable LL. Execution of the goal:-

concatenated((a.b.nil),(a.b.nil),LL)

completes the picture by "filling in" the correct value of LL:-

LL = (a.b.a.b.nil)

Thus we get finally the correct result:-

X = (a.b.a.b.a.b.nil)

We refer to the variable in Prolog as the "logical" variable to draw attention to its special behaviour exemplified above. Basically there is no assignment as such in Prolog, and a variable's value, once specified, cannot be changed (except through backtracking). However the variable's value need not be fixed immediately, and may remain unspecified for as long as is required. In particular, if a variable corresponds to a component of a data structure to be output by a procedure, the value of the variable can be left unspecified when the procedure "returns". The value may then later be filled in by another procedure in the course of the normal matching process.

The logical variable has the further necessary feature that when two uninstantiated variables are matched together, they become linked as one; any value subsequently given to one variable simultaneously

instantiates the other. From a conventional programming standpoint, one can imagine a "pointer" or "reference" to one variable being assigned to the other, with subsequent "dereferencing" being carried out automatically where required.

Consider how the processing of the 'treble' example might be simulated in a conventional language (eg. Algol-68, Pop-2, Lisp); ie. what steps would correspond to execution of the goals:-

```
concatenated((a.b.nil),LL,X),
    concatenated((a.b.nil),(a.b.nil),LL)
```

in that order? The effect of the first goal would have to be simulated by creating a new list (a.b.dummy) with an arbitrary value 'dummy' as the remainder of the list. This list would be assigned to the variable X and a pointer to the location containing the arbitrary value would be assigned to LL. For the second goal, one would create the list (a.b.a.b.nil) and assign it to the location indicated by the pointer previously assigned to LL. In this way the arbitrary value 'dummy' would be overwritten to complete (a.b.a.b.a.b.nil) as the value of X. In the style of Algol-68, these steps might be written as:-

```
list dummy;
ref list LL := dummy;
ref list X := concatenate([a,b],LL);
value of LL := concatenate([a,b],[a,b]);
```

where the arguments and result of procedure 'concatenate' are of mode "ref list".

The original Prolog version achieves the same effect, but without the programmer having to bother about assignments and references. In fact it is the Prolog system which takes care of these

machine-oriented details. The Prolog programmer understands the 'treble' procedure primarily from its declarative reading; from the declarative point of view, even the order of the two goals is irrelevant, let alone the procedural details involved in execution.

Prolog programming requires a certain change of outlook on the part of the programmer, but this is soon acquired with a little practice. The programmer comes to appreciate that Prolog's logical variable provides much of the power of assignment and references, but in a higher-level, easier-to-use form. In a similar way, the disciple of "structured programming", working with a conventional language, finds that "well-structured" control primitives leave little need for goto and that the program is generally easier to understand if gotos are avoided.

3.3 An Example - Looking Up Entries In A Dictionary

To complete this introduction to Prolog, we will now consider an example which will have application in compiler writing. The example involves the data type "dictionary" introduced earlier. A dictionary will provide an efficient representation of a set of pairs of names with values. Thus the dictionary:-

dic(<name>,<value>,<dic-1>,<dic-2>)

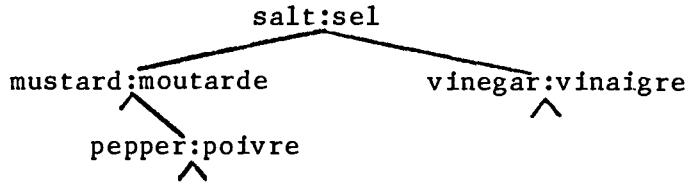
pairs <name> with <value>, together with all the pairings provided by sub-dictionaries <dic-1> and <dic-2>. We assume that the dictionary is ordered, so that all names in <dic-1> are before <name>, and all in <dic-2> are after, and both <dic-1> and <dic-2> are themselves ordered. (Thus no names can be repeated in an ordered dictionary.)

The actual ordering relation is arbitrary, but may be thought of as alphabetical order. Ordering relationships will be expressed using the familiar symbol '<' for the 2-place predicate "is before" and '>' for "is after".

As an example, the following is an (alphabetically-) ordered dictionary pairing English words with their French equivalents:-

```
dic(salt,sel,
    dic(mustard,moutarde,
        void,
        dic(pepper,poivre,void,void)),
    dic(vinegar,vinaigre,void,void))
```

This term is more easily visualised as the tree structure:-



Because our dictionaries are ordered, it is possible to find quickly the value (if any) associated with a given name, without searching through the entire dictionary. So let us now write a Prolog procedure to "look-up" a name in a dictionary and find its paired value. The predicate defined will be

```
lookup(<1>,<2>,<3>)
```

meaning "name <1> is paired with value <3> in dictionary <2>". Given a dictionary:-

```
dic(<name>,<value>,<dic-1>,<dic-2>)
```

we clearly have to distinguish three cases . If the name sought is <name> itself, then the required value is simply <value>, ie.

```
lookup(Name,dic(Name,Value,_,_),Value).
```

Note the use of two "anonymous" variables for the components of the

dictionary which are not relevant to this case. In the other two cases, we have to look for the required name in one of the two sub-dictionaries of the initial dictionary. If the name sought is before <name>, then we must look in the first sub-dictionary, ie.

```
lookup(Name,dic(Namel,_,Before,_),Value) :-
     Name < Namel, lookup(Name,Before,Value).
```

A similar clause deals with the case where the name sought is after <name>, ie.

```
lookup(Name,dic(Namel,_,_,After),Value) :-
     Name > Namel, lookup(Name,After,Value).
```

We have explained these clauses in a procedural way, having in mind the particular goal of looking up a given name in a given dictionary to find an unknown value. The control information built into the Prolog clauses reflects this aim. ie. The order of the clauses, and the order of the goals in the body of each clause, is chosen to be appropriate for the type of goal in mind. Thus, of the three clauses, it is natural to consider the first clause first, since it may give an immediate result without further recursive procedure calls. Again, in the last two clauses, it is sensible to make the test comparing the order of names as the first goal in each clause, since then the recursive call of 'lookup' will only be made on the appropriate sub-dictionary.

Note that the control information is not strictly essential; if a different clause and goal ordering were used, valid results would still eventually be obtained, but the 'lookup' procedure would not go "straight" to the required result - without the right control, the procedure would perform an extremely wasteful exploration of

irrelevant parts of the dictionary.

How can one be so sure that valid results will be obtained whatever the control information? The reason is that the clauses for 'lookup' have a proper declarative interpretation, and the Prolog execution mechanism is guaranteed only to produce answers which accord with the declarative interpretation. Although we explained the clauses "procedurally", they can be understood entirely declaratively as simple statements about dictionaries. For example, the third clause might be read as:-

"If a name Name has a value Value in a dictionary called After, and Name1 is a name which is ordered earlier than Name, then Name has value Value in any dictionary of the form 'dic(Name1,_,_,After)' ".

Of course, the statement would still be true if the condition on the order of Name and Name1 were omitted. As it stands, the statement is true, but less general than it might be. However if attention is restricted to ordered dictionaries, the three clauses for 'lookup' are sufficiently general to cover all possible instances of the 'lookup' relationship. It is generally desirable in Prolog programming to make the logical statements comprising the program no more general than is necessary to give just the truths required. In this way, the Prolog system is prevented from considering irrelevant alternatives. This principle could be thought of as a further form of control information - the system's attention is directed (in fact, restricted) to a small but adequate subset of all the correct statements which could be made.

4.0 A SIMPLE COMPILER WRITTEN IN PROLOG

4.1 Overview

Let us now look at how Prolog can be applied to the task of writing a compiler. We shall only consider a simplified example. Imagine we require a compiler to translate from a small Algol-like language to the machine language of a typical one-accumulator computer. The source language has assignment, IF, WHILE, READ and WRITE statements plus a selection of arithmetic and comparison operators restricted to integers. (A BNF grammar of the language appears later in sub-section /.7/.) The target language instructions:-

(1) arithmetic &c. literal op.	(2) arithmetic &c. memory op.	(3) control transfer	(4) input-output &c.
ADDC	ADD	JUMPEQ	READ
SUBC	SUB	JUMPNE	WRITE
MULC	MUL	JUMPLT	HALT
DIVC	DIV	JUMPGT	
LOADC	LOAD	JUMPLE	
	STORE	JUMPGE	
		JUMP	

each have one (explicit) operand which either (1) is an integer constant, or (2) is the address of a storage location, or (3) is the address of a point in the program, or (4) is to be ignored. Most of the instructions also have a second implicit operand which is either the accumulator or its contents. In addition, there is a pseudo-instruction BLOCK which reserves a number of storage locations as specified by its integer operand.

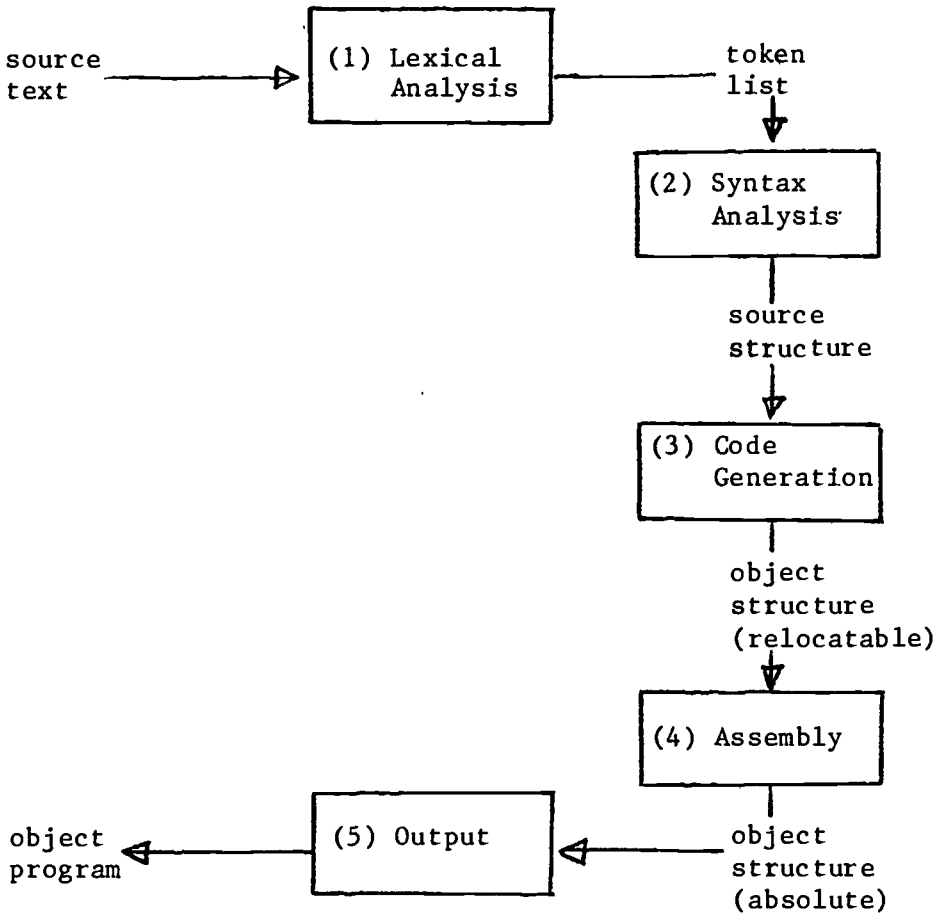
As an illustration of the compiler's function, here is a simple source program (to compute factorials):-

```
READ VALUE;
COUNT := 1;
RESULT := 1;
WHILE COUNT < VALUE DO
     (COUNT := COUNT+1;
      RESULT := RESULT*COUNT);
WRITE RESULT
```

and the following is the straightforward translation into machine language which the compiler will produce. (The columns headed symbol are not part of the compiler's output and are merely comments for the reader.)

symbol:	address	instruction	operand	:symbol
	0	READ	0	
	1	STORE	21	VALUE
	2	LOADC	1	
	3	STORE	19	COUNT
	4	LOADC	1	
	5	STORE	20	RESULT
LABEL1	6	LOAD	19	COUNT
	7	SUB	21	VALUE
	8	JUMPGE	16	LABEL2
	9	LOAD	19	COUNT
	10	ADDC	1	
	11	STORE	19	COUNT
	12	LOAD	20	RESULT
	13	MUL	19	COUNT.
	14	STORE	20	RESULT
	15	JUMP	6	LABEL1
LABEL2	16	LOAD	20	RESULT
	17	WRITE	0	
	18	HALT	0	
COUNT	19	BLOCK	3	
RESULT	20			
VALUE	21			

Compilation will be performed in five stages of which we shall only look at the middle three.



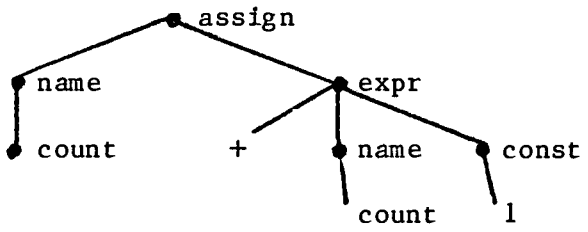
The first stage, Lexical Analysis, involves grouping the characters of the source text into a list of basic symbols called "tokens" (represented by Prolog atoms and integers). This stage is relatively uninteresting and will not be discussed further. The second stage, Syntax Analysis, is responsible for parsing the token list. Essentially, the effect of the analysis is to recognise the abstract program structure encoded in the characters of the source text and give this structure a name. The name will be a Prolog term. For example, the name of the statement:-

```
COUNT := COUNT+1
```

will be:-

```
assign(name(count),expr(+,name(count),const(1)))
```

which can also be pictured as the tree:-



Since the Syntax Analysis stage is not our main topic, the discussion of it will be postponed to a later section.

The third stage, Code Generation, produces the basic structure of the object program, but machine addresses are left in a "symbolic" form. These addresses are computed and filled in by the fourth stage, Assembly.

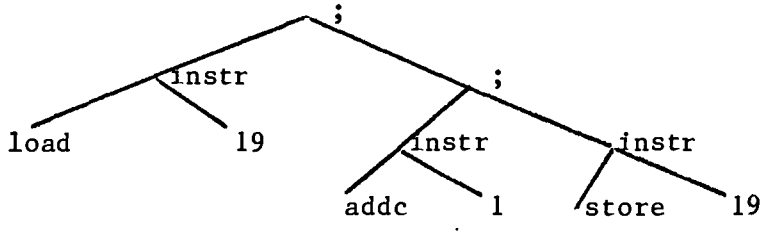
We shall not go into the less interesting final stage of outputting an actual object program (as a bit string say). The result of the Assembly stage will be a Prolog term which names the object program structure. For example, the name for:-

```
LOAD    19
ADDC    1
STORE   19
```

will be:-

```
(instr(load,19); instr(addc,1); instr(store,19))
```

where the binary functor ';' has been written as a right-associating infix operator, ie. the term can be pictured as:-



Note that the ';' functor is only used to indicate sequencing, and

that the same sequence can be expressed by different terms, eg.

(a;(b;c)) and ((a;b);c)

4.2 Compiling The Assignment Statement

Consider first the problem of compiling the assignment statement:-

<name> := <expression>

The code for this will have the form:-

<expression code>
STORE <address>

where <expression code> is the code to evaluate the arithmetic expression <expression> yielding a result in the accumulator. The STORE instruction stores this result at <address>, the address of the location named <name>.

We want to make this semi-formal specification precise by translating it into a Prolog clause. Now the Prolog term which names the source form is:-

assign(name(X),Expr)

where X and Expr are Prolog variables which correspond to the BNF non-terminals <name> and <expression> in the semi-formal specification. Similarly, a Prolog term naming the target form is:-

(Exprcode; instr(store,Addr))

where Exprcode and Addr are Prolog variables corresponding to <expression code> and <address>. We have to define the relationship between X,Expr,Exprcode and Addr. Suppose the source language names are to be mapped into machine addresses in accordance with a

dictionary D. Then one necessary condition is expressed by the Prolog goal:-

```
lookup(X,D,Addr)
```

The condition relating Expr and Exprcode may be expressed by the goal:-

```
encodeexpr(Expr,D,Exprcode)
```

where the meaning of the predicate 'encodeexpr(<1>,<2>,<3>)' is "<3> is the code for the expression <1> conforming to dictionary <2>". If 'encodestatement(<1>,<2>,<3>)' is a similar predicate meaning "<3> is the code for the statement <1> conforming to dictionary <2>", then the complete Prolog clause we require is:-

```
encodestatement(assign(name(X),Expr),D,
        (Exprcode;
         instr(store,Addr))
):-
   lookup(X,D,Addr),
   encodeexpr(Expr,D,Exprcode).
```

All we have done so far is to make precise the informal rule for compiling an assignment statement. Now the resulting clause is not only an exact statement of the rule, but will also actually be the part of the compiler responsible for implementing the rule. The clause represents one case of the procedure 'encodestatement(<1>,<2>,<3>)' which takes as input a statement <1> and a dictionary <2> and produces as output object code <3>.

If we regard the clause as just a statement of a rule, the ordering of the two goals in the body of the clause is irrelevant. Now usually the order is very important when we want also to use the clause as part of a practical procedure. However in this case, as for many of the other clauses which make up the compiler, it will become

clear that the clause will function perfectly well whichever order is chosen.

4.3 Compiling Arithmetic Expressions

We already know the clauses for 'lookup', so let us move on to the clauses for 'encodeexpr'. For reasons which will become clearer later, 'encodeexpr' is defined in terms of another predicate:-

```
encodeexpr(Expr,D,Code) :-
      encodesubexpr(Expr,0,D,Code).
```

The extra (integer) argument of 'encodesubexpr' provides information about the context in which the expression occurs, and is zero unless the expression is a sub-expression of another expression. Let us now look at the clauses for 'encodesubexpr' and see how they embody rules for translating the different types of arithmetic expression.

If the expression is just a constant <const> then the instruction:-

```
LOADC <const>
```

has the desired effect of loading the constant into the accumulator. Similarly, if the expression is a location named <name> then the instruction:-

```
LOAD <addr>
```

loads the current value of the location, where <addr> is the location's address. These two rules are expressed in Prolog by the clauses:-

```
encodesubexpr(const(C),_,_, instr(loadc,C) ).

encodesubexpr(name(X),_,D, instr(load,Addr) ):-
   lookup(X,D,Addr).
```

The final possibility is a composite expression of the form:-

<expression 1><operator><expression 2>

If <expression 2> is simply a constant or location name, the code generated for the composite expression takes the form:-

```
<expression 1 code>
<instruction>
```

where <expression 1 code> is the translation of <expression 1> and <instruction> is the appropriate machine instruction which applies <operator> to the value in the accumulator and operand <expression 2>. For example:-

<expression>+7

translates to:-

```
<expression code>
ADDC 7
```

The clauses which express this more generally are:-

```
encodesubexpr(expr(Op,Expr1,Expr2),N,D,
        (Expr1code;
         Instruction)
):-
   apply(Op,Expr2,D,Instruction),
   encodesubexpr(Expr1,N,D,Expr1code).

apply(Op,const(C),_, instr(Opcode,C) ):-
   literalop(Op,Opcode).

apply(Op,name(X),D, instr(Opcode,Addr) ):-
   memoryop(Op,Opcode),
   lookup(X,D,Addr).

literalop(+,addc).      memoryop(+,add).
literalop(-,subc).      memoryop(-,sub).
literalop(*,mulc).      memoryop(*,mul).
literalop(/,divc).      memoryop(/,div).
```

Notice how the information residing in the clauses for 'literalop' and 'memoryop' would conventionally be treated as tables of data rather than procedures.

{The following covers the more general case where <expression 2> is composite, and may be skipped on first reading.

In this more general case, the code will have to be of the form:-

```
<expression 2 code>
STORE <temporary>
<expression 1 code>
<op-code><temporary>
```

where <expression 2 code> evaluates <expression 2> and the result is stored temporarily at address <temporary>. <expression 1> is then evaluated and the instruction of type <op-code> applies <operator> to the pair of values respectively contained in the accumulator and previously stored at location <temporary>. Note that if <expression 1 code> itself requires temporary storage locations, these must all be different from <temporary>. These requirements are met by the clause:-

```
encodesubexpr(expr(Op,Expr1,Expr2),N,D,
          (Expr2code;
           instr(store,Addr);
           Expr1code;
           instr(Opcode,Addr))
):-
   complex(Expr2),
   lookup(N,D,Addr),
   encodesubexpr(Expr2,N,D,Expr2code),
   N1 is N+1,
   encodesubexpr(Expr1,N1,D,Expr1code),
   memoryop(Op,Opcode).

complex(expr(_,_,_)).
```

(Here the goal 'N1 is N+1' means "N1 is the value of the arithmetic expression N+1".) The procedure's extra argument N is an integer which is used as a name to be looked up in the dictionary D. In this way the compiler uses integers as

"private" names for the temporary storage locations it requires. In other respects, temporaries are treated just like any other locations defined in the actual source program, and are recorded in the same dictionary. Notice how any temporaries required for the evaluation of <expression 1> are named by the integers N+1, N+2, etc., and thus are distinct from the temporary named by the integer N which is used to preserve the previously calculated value of <expression 2> while <expression 1> is being evaluated.}

4.4 Compiling The Other Statement Types

Now let us consider a statement type which is, in itself, slightly more complex to compile - the IF statement:-

IF <test> THEN <then> ELSE <else>

The code for this will take the form:-

```
        <test code>
        <then code>
        JUMP <label 2>
<label 1>:
        <else code>
<label 2>:
```

where <test code> causes a jump to <label 1> if the test proves false. As in an assembly language program, we have used labels to indicate the instructions whose addresses are <label 1> and <label 2>.

The Prolog formulation of this is:-

```
encodestatement(if(Test,Then,Else),D,
        (Testcode;
         Thencode;
         instr(jump,L2);
      label(L1);
         Elsecode;
      label(L2))
):-
   encodetest(Test,D,L1,Testcode),
   encodestatement(Then,D,Thencode),
   encodestatement(Else,D,Elsecode).
```

Notice that the clause does not fix the addresses L1 and L2, but merely indicates constraints on their values through labelling the object code. One can think of the output from the procedure 'encodestatement' as being relocatable code. The output term will contain free variables L1 and L2 whose values will not be fixed until stage 4 of compilation - the Assembly stage. This is an example of the use of the logical variable to delay specifying certain parts of a data structure.

The clauses for 'encodetest' are as follows:-

```
encodetest(test(Op,Arg1,Arg2),D,Label,
        (Exprcode;
         instr(Jumpif,Label))
):-
   encodeexpr(expr(-,Arg1,Arg2),D,Exprcode),
   unlessop(Op,Jumpif).

unlessop(=,jumpne).
unlessop(<,jumpge).
unlessop(>,jumple).
unlessop(\=,jumpeq).
unlessop(=<,jumpgt).
unlessop(>=,jumplt).
```

The test is effected by computing the difference of the two operands to be compared, and then applying a conditional jump instruction. 'Label' is the address to jump to if the test fails. The meaning of the clauses should be clear by analogy with cases previously discussed.

The clauses for translating the remaining statement types are as follows:-

```
encodestatement(while(Test,Do),D,
      (label(L1);
         Testcode;
         Docode;
         instr(jump,L1);
      label(L2))
):-
   encodetest(Test,D,L2,Testcode),
   encodestatement(Do,D,Docode).

encodestatement(read(name(X)),D, instr(read,Addr) ):-
   lookup(X,D,Addr).

encodestatement(write(Expr),D,
        (Exprcode;
         instr(write,0))
):-
   encodeexpr(Expr,D,Exprcode).

encodestatement((S1;S2),D, (Code1;Code2) ):-
   encodestatement(S1,D,Code1),
   encodestatement(S2,D,Code2).
```

Notice how the "serial" statement:-

```
<statement 1>;<statement 2>
```

is treated as just another statement type.

4.5 Constructing The Dictionary

Now that we have considered all the elements of the source language, it remains to describe how a program as a whole is compiled. Many of the clauses already stated have referred to a common dictionary D. So far we have tacitly assumed that this dictionary (or symbol table) has been constructed in advance and supplied as "input" to each procedure which translates source language constructs. Now it happens that, with a little care, we can arrange for the dictionary to be built up in the course of the main translation process (stage 3).

The clauses for 'lookup' not only do the job of consulting existing dictionary entries, but will also serve to insert new entries as required. In fact 'lookup' is a good example of a "multi-purpose" procedure. Its very useful and rather remarkable behaviour depends on the full flexibility of the logical variable.

Let us first restate the 'lookup' clauses (with a slight change in the first clause, to be discussed shortly):-

```
lookup(Name,dic(Name,Value,_,_),Value):-!.
lookup(Name,dic(Name1,_,Before,_),Value):-
   Name < Name1, lookup(Name,Before,Value).
lookup(Name,dic(Name1,_,_,After),Value):-
   Name > Name1, lookup(Name,After,Value).
```

To see how the 'lookup' procedure can be used to create a dictionary, consider the effect of executing the goals:-

```
lookup(salt,D,X1),
lookup(mustard,D,X2),
lookup(vinegar,D,X3),
lookup(pepper,D,X4),
lookup(salt,D,X5)
```

in that order, assuming all the variables are initially uninstantiated, even the dictionary argument D. One can interpret these goals as saying : "construct a dictionary D such that 'salt' is paired with X1 and 'mustard' is paired with X2 and ...". The first goal is immediately solved by the first clause for 'lookup' giving:-

```
D = dic(salt,X1,D1,D2)
```

and leaving X1 uninstantiated. Thus variable D is now instantiated to a partially specified dictionary. The second of the original goals is executed next. Execution proceeds initially as for a normal call of 'lookup' and produces the recursive call:-

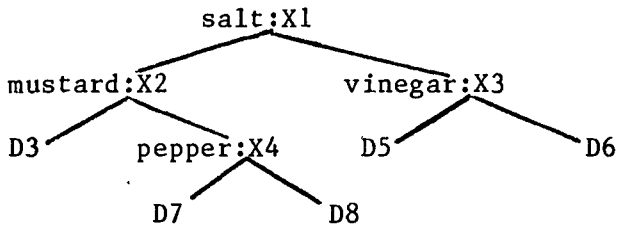
```
lookup(mustard,D1,X2)
```

Now, since D1 is uninstantiated, this goal is solved immediately,

giving:-

D = dic(salt,X1,dic(mustard,X2,D3,D4),D2)

In this way 'lookup' is inserting new entries in a partially specified dictionary. By the time of executing the fifth of the original goals, D is instantiated to a dictionary which may be pictured as:-



The effect of the fifth goal is to leave the dictionary unaltered; the only result is the instantiation:-

X5 = X1

Thus both values paired with "salt" are guaranteed to be the same.

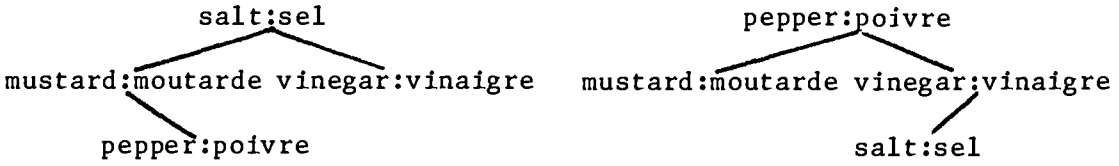
We have seen that:-

(a) the dictionary can be built up as we go along, starting from a free variable, and with free variables as the terminal nodes of the dictionary at every stage;

(b) the values which are paired with the names in the dictionary can be left unspecified til later - their places are taken by variables, and different variables representing the same value will be identified where necessary.

As used in the compiler, the 'lookup' procedure builds up a dictionary associating storage location names with free variables representing their addresses. These addresses are only filled in during the Assembly stage of compilation.

We shall now consider the meaning of, and reason for, the extra '!' in the first clause of 'lookup'. A fundamental reason for the change is that an ordered dictionary for a given set of pairings is not unique. For example, the two ordered dictionaries diagrammed below embody the same set of associations:-



In theory, the 'lookup' procedure could choose to build either of these, or any other equivalent dictionary. This is reflected in the fact that a 'lookup' goal such as:-

```
lookup(salt,D,X1)
```

with an uninstantiated variable as second argument will match not only the first clause for 'lookup' but also either of the other two. These alternative matches in principle allow different but equivalent forms of dictionary to be constructed.

Obviously we wish to limit the choice to just one of these equivalent forms. Moreover, the generation of alternative forms may be highly inefficient, if not impossible. This is because a match of, say,

```
lookup(salt,D,X1)
```

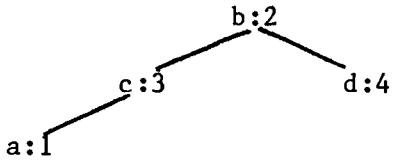
against the second clause gives rise to the goal:-

```
salt < Name1
```

with Name1 uninstantiated. Now in theory this goal should generate any name which is ordered later than 'salt'. In practice, it is highly undesirable to execute a goal with such a large set of alternative solutions, and the actual implementation of '<' may well be such as to

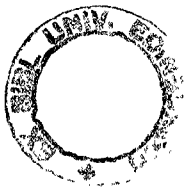
make the goal impossible to execute.

The alert reader will also have noticed that the declarative meaning of the clauses for 'lookup' does not guarantee a dictionary of the type we require - that is, an ordered dictionary (with no name repeated). For example, if <dic> is the dictionary pictured as:-



then 'lookup(a,<dic>,1)' is true, but <dic> is not ordered. Strictly speaking, a check should be made somewhere in the compiler that the dictionary created and used during compilation is indeed ordered. This check is tiresome and in practice unnecessary.

All of these various potential drawbacks to the "creative" use of 'lookup' are circumvented by inserting the cut operator '!' as a pseudo-"goal" in the first clause. The cut operator is an additional control device provided by Prolog, which should be ignored when reading a clause declaratively*. {* With certain usages of cut, there is no meaningful declarative reading for the "clause"; however this does not apply to any of the clauses in this Part of the thesis.} When a "cut" pseudo-goal has been executed, if backtracking should later return to that point, the effect is to immediately fail the "parent" goal, ie. the goal which activated the clause containing the cut. In other words, the cut operation commits the Prolog system to all choices made since the parent goal was invoked. For our 'lookup' procedure, the cut means "if a match is obtained against the first clause, don't ever try any of the subsequent clauses".



Given Prolog's procedural semantics, it is not difficult to see how the qualification expressed by the cut symbol ensures that 'lookup' constructs a unique ordered dictionary starting from an initially uninstantiated variable. The dictionary is "unique" except that the terminal nodes are free variables which really represent unspecified sub-dictionaries. All of these variables must finally be instantiated to 'void' in order to obtain the smallest possible dictionary meeting the required conditions.

4.6 Compiling The Whole Program, And The Assembly Stage

The translation of a complete source program (or rather, its abstract structure) into an object program (structure) with absolute addresses is expressed by the following clause:-

```
compile(Source,
        (Code;
         instr(halt,0);
         block(L))
):-
   encodestatement(Source,D,Code),
   assemble(Code,0,N0),
   N1 is N0+1,
   allocate(D,N1,N),
   L is N-N1.
```

(A goal, such as 'L is N-N1' above, of the form '<var> is <expr>' means that <var> is the value of the arithmetic expression <expr>.) The result of compiling the program Source is a sequence of instructions Code followed by a HALT instruction and then a block of storage for the variables used in Source. Unlike most of the compiler clauses described so far, the particular order of the goals in this clause is essential control information.

Stage 3 of compilation (Relocatable Code Generation) is represented by the goal:-

encodestatement(Source,D,Code)

Observe that when this goal is invoked, dictionary D is completely unspecified, ie. D is still a free variable. So stage 3 really returns two outputs - the code and the dictionary.

Strictly speaking, for logical soundness, the clause for 'compile' should contain an extra goal, say:-

ordereddictionary(D)

to check that D is indeed an ordered dictionary. We may imagine this goal being inserted after the 'encodestatement' goal. However, as noted previously, this check can be dispensed with in practice.

At the end of stage 3, Code still contains many free variables - representing the yet to be specified addresses of writeable locations and labelled instructions. Thus stage 3 makes extensive use of the full power of the logical variable to delay fixing of addresses until stage 4. The goal:-

assemble(Code,0,N0)

computes the addresses of labelled instructions and returns N0, the address of the end of Code. N1 is therefore the address of the start of the block of storage locations. The goal:-

allocate(D,N1,N)

is responsible for laying out the storage required for the source language symbols contained in dictionary D. It fills in the corresponding addresses and returns N, the address of the end of the storage block. Finally the length L of the storage block is

calculated from N and N1.

The procedure for 'assemble' is neat and simple:-

```
assemble((Code1;Code2),N0,N):-
   assemble(Code1,N0,N1),
   assemble(Code2,N1,N).
assemble(instr(_,_),N0,N):- N is N0+1.
assemble(label(N),N,N).
```

Note that 'assemble(<1>,<2>,<3>)' means that <2> is the start address and <3> the end address of the sequence of instructions <1>.

The procedure for 'allocate' has a similar character:-

```
allocate(void,N,N):-!.
allocate(dic(Name,N1,Before,After),N0,N):-
   allocate(Before,N0,N1),
   N2 is N1+1,
   allocate(After,N2,N).
```

Observe that the layout of the source symbols will be in dictionary order.

Note that the dictionary input to 'allocate' from 'encodestatement' is incomplete in the sense that the terminal nodes are still variables. The 'allocate' procedure in fact chooses the smallest possible dictionary, ie. the one which contains only symbols actually occurring in the source program. If it chose otherwise, the object program would still be correct but would contain extra unused storage locations. The proper choice is achieved by placing the clause for the 'void' case first, with a cut '!' to prevent any possibility of backtracking considering other alternatives, cf. the use of cut in 'lookup'.

We have now looked at all the clauses needed to perform the Code Generation and Assembly stages of the compiler. Except where otherwise noted, the particular order in which these clauses are stated is unimportant, ie. the performance will be virtually the same whichever order is chosen.

4.7 Syntax Analysis

We shall now show very briefly how the parser, or Syntax Analysis stage of compilation, is programmed in Prolog. Van Emden [1975, Section 6] gives a much fuller introduction to the basic method we use for writing parsers in logic. The theory of this method is described by Colmerauer [1975], from whom the technique originated. The clauses we require are closely related to the following BNF grammar of the source language:-

```
<program>          ::=  <statements>
<statements>       ::=  <statement>|
                        <statement>;<statements>
<statement>        ::=  <name>:=<expr>|
                        IF <test> THEN <statement> ELSE <statement>|
                        WHILE <test> DO <statement>|
                        READ <name>|
                        WRITE <expr>|
                        (<statements>)
<test>             ::=  <expr><comparison op><expr>
<expr>             ::=  <expr><op 2><expr 1>|
                        <expr 1>
<expr 1>           ::=  <expr 1><op 1><expr 0>|
                        <expr 0>
<expr 0>           ::=  <name>|
                        <integer>|
                        (<expr>)
<comparison op>    ::=  = | < | > | =< | >= | \=
<op 2>             ::=  * | /
<op 1>             ::=  + | -
```

The essential idea behind the translation into Prolog is that a BNF non-terminal becomes a predicate of three arguments:-

<non-terminal>(<start>,<end>,<name>)

meaning "the token list <start> commences with a phrase of type <non-terminal> ending at a point where the list of remaining tokens is <end>; the structure of the phrase is identified by <name>". Now because the grammar contains some left recursive rules, and for other efficiency reasons, parts of the grammar are rewritten to facilitate left-to-right top-down parsing. Some of the resulting predicates have to be given an additional argument which is the name of the preceding phrase. For example:-

restexpr(<n>,<start>,<end>,<name0>,<name>)

means that the token list <start> commences with the remainder of an arithmetic expression of precedence <n> ending at <end> and the whole expression is named <name> if the preceding subexpression is named <name0>. Here then is the Prolog translation of the BNF grammar:-

```
program(Z0,Z,X) :- statements(Z0,Z,X).

statements(Z0,Z,X) :- statement(Z0,Z1,X0), reststatements(Z1,Z,X0,X).

reststatements((';'.Z0),Z,X0,(X0;X)) :- statements(Z0,Z,X).
reststatements(Z,Z,X,X).

statement((V.':='.Z0),Z,assign(name(V),Expr)) :-
   atom(V), expr(Z0,Z,Expr).
statement((if.Z0),Z,if(Test,Then,Else)) :-
   test(Z0,(then.Z1),Test),
   statement(Z1,(else.Z2),Then),
   statement(Z2,Z,Else).
statement((while.Z0),Z,while(Test,Do)) :-
   test(Z0,(do.Z1),Test),
   statement(Z1,Z,Do).
statement((read.V.Z),Z,read(name(V))) :- atom(V).
statement((write.Z0),Z,write(Expr)) :- expr(Z0,Z,Expr).
statement(('('.Z0),Z,S) :- statements(Z0,(')'.Z),S).

test(Z0,Z,test(Op,X1,X2)) :-
   expr(Z0,(Op.Z1),X1), comparisonop(Op),
   expr(Z1,Z,X2).

expr(Z0,Z,X) :- subexpr(2,Z0,Z,X).

subexpr(N,Z0,Z,X) :- N>0, N1 is N-1,
   subexpr(N1,Z0,Z1,X0),
   restexpr(N,Z1,Z,X0,X).
subexpr(0,(X.Z),Z,name(X)) :- atom(X).
subexpr(0,(X.Z),Z,const(X)) :- integer(X).
subexpr(0,('('.Z0),Z,X) :- subexpr(2,Z0,(')'.Z),X).

restexpr(N,(Op.Z0),Z,X1,X) :- op(N,Op), N1 is N-1,
   subexpr(N1,Z0,Z1,X2),
   restexpr(N,Z1,Z,expr(Op,X1,X2),X).
restexpr(N,Z,Z,X,X).

comparisonop(=).
comparisonop(<).
comparisonop(>).
comparisonop(=<).
comparisonop(>=).
comparisonop(\=).

op(2,*).          op(1,+).
op(2,/).          op(1,-).
```

5.0 THE ADVANTAGES OF PROLOG FOR COMPILER WRITING

This section summarises the particular advantages of Prolog as a language for writing compilers. Many of the advantages should be clear from the main example discussed above. It is important to take into account, not just the compiler which is the end product, but also the work which must go into initially designing and building it and into subsequently "maintaining" it.

So let us review how one might set about constructing a compiler. Initially, the picture is just of a black box with source programs as input and correctly translated object programs as output. The first consideration is to decide how the output is related to the input. It is natural to examine the structure of the source language and to devise for each element of the language a rule for translating it into target language code. These rules form a specification of the compiler's function. The final and generally more laborious stage of compiler construction involves implementing procedures which efficiently carry out the translation process in accordance with the specification.

The major advantage of implementation in Prolog is that it is possible for the final stage to be almost trivial. For a compiler such as the sample one discussed, it is not a great exaggeration to say that

"the specification _is_ the implementation".

Thus the procedures which make up the compiler consist of clauses, each of which can generally be interpreted as a rule describing a possible translation of some particular construct of the source

language into the target language. The burden of the implementation stage reduces to ensuring that the specification can be used as an efficient implementation. This requires the addition of suitable control information (ie. choosing the ordering of clauses and goals) and may involve some rewriting of parts of the specification to allow an efficient procedural interpretation.

The closeness of implementation and specification brings many benefits:-

* The implementation is more readable and may be virtually self-documenting.

* The correctness (or otherwise) of the implementation is more easily apparent and the scope for error is greatly reduced. As long as each clause is a valid rule for translating the source language, one can be confident that the compiler will not generate erroneous code.

* Compiler modifications and source language extensions are more readily incorporated, since the compiler consists of small independently-meaningful units (clauses) which are directly related to the structure of the source language.

There are a number of conventional programming language features which would normally have to be used in a compiler implementation, but which do not appear explicitly in a Prolog implementation. These include assignment, references (pointers), operations for creating data structures, operations for selecting from data structures, conditional or test instructions, and the goto instruction. Of course, all these features are being used implicitly, behind the scenes, by the Prolog system. In effect, the Prolog system assumes

much of the responsibility for "coding up" the implementation. This relieves the programmer of tedious details and protects him against errors commonly associated with the low-level features mentioned, eg.

* referring to a non-existent component of a data structure;
* attempting to use the value of a variable before it has been assigned;
* attempting to use a value which is obsolete, such as a "dangling reference" to storage which has been de-allocated;
* overwriting a value or part of a data structure which is still needed elsewhere in the program;
* omitting to test for a special case before dropping through to the else clause of a conditional.

In a conventional language, errors such as these typically produce bugs which are difficult to trace and eradicate. At best the program will halt immediately with some error message, which may or may not help the programmer to pinpoint the bug. More usually, the bug will only manifest itself later in the processing, by which time the root cause will be difficult to determine.

Such situations cannot arise with the basic Prolog language covered here, since none of the low-level features mentioned is present in the language. Moreover, the (procedural) semantics of Prolog is totally defined; a syntactically correct program is guaranteed to be legal, and is incapable of performing, or even attempting to perform, any invalid or undefined operation. If there is a "bug" in a Prolog program, it merely means that the program, while being perfectly legal, doesn't do exactly what the programmer

intended. The actual behaviour is entirely predictable and therefore the "bug" is normally found relatively easily. A totally defined semantics is of great practical significance and is almost unique among programming languages.

To summarise, Prolog has the following advantages as a compiler-writing tool:-

* less time and effort is required;
* there is less likelihood of error;
* the resulting implementation is easier to "maintain" and modify.

6.0 THE PRACTICABILITY OF PROLOG FOR COMPILER WRITING

Granted that Prolog is a very congenial language for compiler writing, the question naturally arises whether an implementation in Prolog can perform well enough to be practically useful. The answer obviously depends on how efficiently Prolog itself is implemented.

The first Prolog system was an interpreter written in Fortran at the University of Marseille [Battani & Meloni 1973]. This proved to be surprisingly fast. More recently, building on the techniques developed at Marseille, two colleagues and I have implemented a Prolog compiler [Warren et al. 1977] [Warren 1977] for the DECsystem-10 machine.

The machine code generated by this compiler is reasonably efficient and is not so very different from that which might be produced by a compiler for a conventional list- or record-processing language. The principal effect of the compilation is to translate the head of each clause into instructions which will do the work of matching against any goal. Of the two terms involved in the matching, it is the clause head which is compiled, since this is uninstantiated prior to the matching, unlike the goal. Because the variables in the head are uninstantiated prior to the matching, their first occurrences can be compiled into simple assignment operations.

The code generated for a compound term has to distinguish between two cases. If the subterm matches against a variable, a new data structure must be constructed and assigned to the variable. This case is handled by an out-of-line subroutine. The other case concerns

matching against a non-variable. This is performed essentially by in-line code. It comprises a test for matching functors (record types), followed by the compiled form of each of the subterms of the compound term. This code will be responsible for accessing subcomponents of the matching data structure.

Many Prolog procedures consist of a number of clauses giving a definition by cases - each clause accounts for a different possible form of the input. This characteristic is particularly evident in compiler writing as illustrated above. For example the clauses for 'encodestatement' each match a different statement type. Here, and more generally, it is natural to place the principal input as first argument of the predicate. Our Prolog compiler capitalises on this fact by compiling in a fast "switch" or "computed goto", branching on the form (principal functor) of the first argument. Thus instead of trying each clause in turn, the code automatically selects only appropriate clauses (often just one).

As far as the general efficiency of Prolog is concerned, space economy is more likely to be a limitation than speed. From our discussion of the (basic) language it is clear that the responsibility for storage management falls entirely on the system and not on the programmer. In meeting this responsibility, the Prolog compiler employs a certain degree of sophistication.

In particular, it automatically classifies variables into two types ("local" and "global") with storage allocated from different areas analogous to the "stack" and "heap" of Algol-68. Local storage is recovered automatically by a conventional stack mechanism when a

procedure returns, provided the procedure has no more alternative multiple results to generate through backtracking. In addition, a second stack mechanism associated with backtracking ultimately recovers all storage, both local and global. Thus a garbage collector is not an essential part of the system, although one is provided. This is in contrast to the situation for the "heap" of Algol-68 and similar languages, where storage can only be recovered by the potentially very expensive process of garbage collection.

Although the automatic storage management of basic Prolog is quite effective, it is not adequate on its own for really large tasks. For example, it is currently unrealistic to expect a compiler written in basic Prolog to compile a sizable program in one step, as, unaided by the user, the storage requirements would exhaust main memory. A technique which can be adopted at present is to compile small units of the program, eg. "lines", "blocks" (or in the case of Prolog itself "clauses"), using the "pure" methods we have described; the compiler as a whole consists of a number of "pure" procedures linked together using more ad hoc (and conventional) methods. The ad hoc parts are written using extensions beyond the basic Prolog language, which are outside the scope of this Part of the thesis. The essential feature of the "impure" code is that use is made of further storage areas and external files, all of which have to be managed directly by the programmer. This approach to compiler writing in Prolog enables one to produce a practical implementation, large parts of which are written in the basic Prolog language, with all the advantages discussed above.

Given the theme of this Part of the thesis, it should come as no surprise that the Prolog compiler is itself written in Prolog, using the very principles which are the subject of this Part of the thesis. Data on this "bootstrapped" compiler may therefore give some idea of the kind of performance attainable with Prolog as the implementation language. Note that the compiler does not attempt any sophisticated optimisation.

The compiler generates about 2 machine instructions (= 2 machine words, of 36 bits each) per source symbol (ie. constant, functor or variable). It takes typically around 10.6 seconds to generate 1000 words of code. The amount of "short-term" (Prolog-controlled) working storage required during compilation is rarely more than 5K words, ie. this is a normal bound on the amount of storage required to compile any one clause. {The remaining "long-term" (programmer-controlled) working storage is needed primarily for a global symbol table, the size of which depends on the number of different functors in the program being compiled.} The total code of the compiler itself (which is not overlayed) is about 25K words.

Briefly, the performance indicated by these figures is reasonably acceptable, although naturally it falls short of what can be attained in a low-level language with efficiency as the only objective. Nevertheless, the performance is not out of line with that of certain other items of software on the DECsystem-10 (eg. the manufacturer's assembler, "MACRO").

Prolog is a promising language for software implementation where the main priority is to have a correctly working system available quickly, or where the system specification is liable to change. Better performance can certainly be obtained from an implementation in a lower-level language; for this, a preliminary Prolog formulation can serve as a very useful prototype. It is likely that most of the improvement will be attributable to a few relatively simple but heavily used procedures (eg. lexical analysis, dictionary lookup), and so a mixed language approach may be an attractive possibility. An alternative view (which I favour) is to look for more sophisticated ways of compiling special types of Prolog procedure, guided probably by extra pragmatic information provided by the Prolog programmer.

PART II - IMPLEMENTING PROLOG

1.0 INTRODUCTION

This Part of the thesis 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 /2./. The sample programs listed in Appendix /5./ and referred to in Section /8.1/ may be useful.

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

1.1 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 suggest Prolog as a promising query language for a relational database (cf. [van Emden 1976] and Zloof's "Query by Example" [1974]). These characteristics have already been exploited in interesting applications of Prolog in the practical areas of computer-aided architectural design [Markusz 1977] and drug design [Darvas et al. 1976,1977].

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 rplaca and rplacd 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 run 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 procedural 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.

1.2 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.

Since the Marseille Fortran implementation, other authors have implemented Prolog interpreters of essentially similar designs. Peter Szeredi [1977] has a very practical CDL implementation, with nice debugging aids, running on various machines including ICL 1900 and S/4. Maurice Bruynooghe [1976] has written an interpreter in Pascal. He gives a good introduction to the fundamentals of Prolog implementation and describes a space saving technique using a "heap". Grant Roberts [1977] has a very efficient interpreter written in IBM 370 assembler, which also has good "human engineering".

The main subject of this Part of the thesis 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 Part of the thesis 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 <u>all</u> 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 /4.9/ and Appendix /2./.

Note: This Part of the thesis 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".

2.0 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 /5./ and discussed in Section /8.1/.

2.1 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).
```

2.2 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 declarative semantics 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 true if it is the head of some clause instance and each of the goals (if any) of that clause instance is true, where an instance of a 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 determinately if execution is complete and no alternative clauses exist for any of the goals invoked during the execution (including the original goal).

2.3 The Cut Operation

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 commits 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.

3.0 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.

3.1 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 >

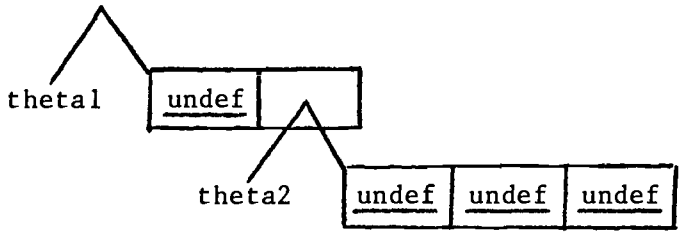
where the frame is a vector of constructed terms representing the values of the variables in the source term. Each variable is given a number which indicates the position in the frame of its value. (We shall also say the variable is bound to that value.) If the variable is unbound, its value is a special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

Thus if we are given source terms:-

theta1 = tree(X1,a,X2)

theta2 = tree(Y1,Y2,Y3)

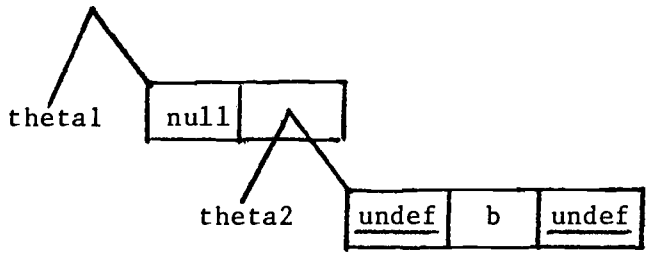
then the constructed term pictured as:-



represents the term:-

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

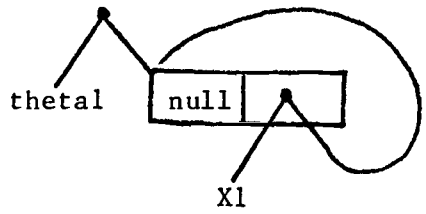
If the source term is a constant then there is no need to provide a frame, so we shall treat constants as being both source terms and constructed terms. Thus the constructed term pictured as:-



represents the term:-

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

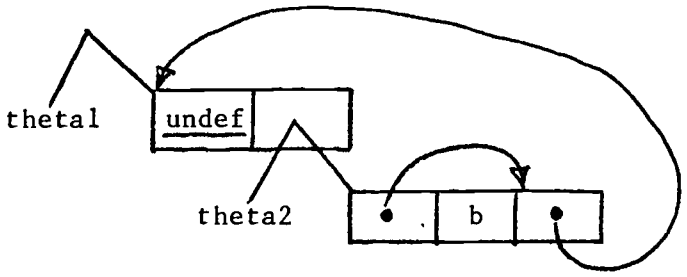
Notice also that the source part of a constructed term may be a variable so that if, for example, X2 in theta1 is bound to X1 and X1 is in turn bound to 'null', then:-



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.

3.2 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 'undef'. Unification is also responsible for setting every variable cell in the new environment to 'undef' 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.

3.3 Implementing The Cut Operation

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.

3.4 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 /4.9/), 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.

4.0 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./.

4.1 The Main Data Areas

Each clause of a Prolog source program 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.

4.2 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.

Note: 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.

4.3 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 inner variable literal or the address of a skeleton literal, atom literal or integer literal. 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.

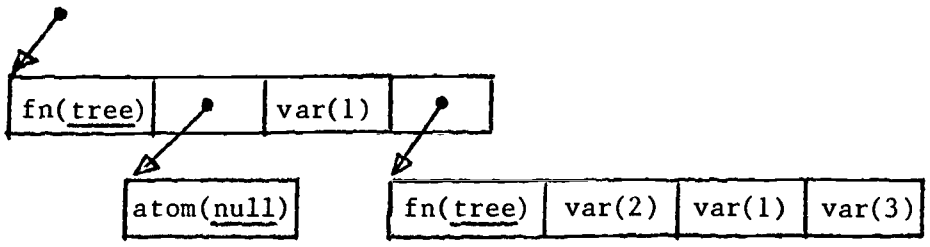
We shall write '[S]' for the address of a structure S. Thus the address of the literal corresponding to the skeleton:-

```
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 outer literals to represent the arguments of a goal. An outer literal is either the address of an atom integer

or skeleton literal, or is a local literal, a global literal or a void 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.

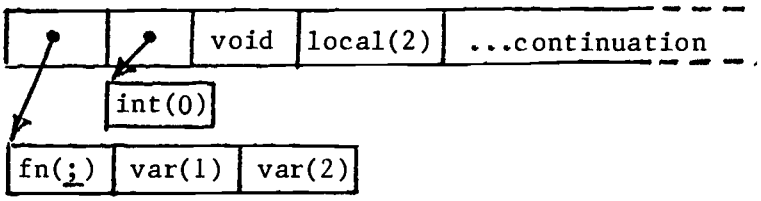
Thus the arguments of the second goal in the clause:-

```
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.

4.4 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.

4.5 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 'undef'. Otherwise the result is the final construct examined.

4.6 Unification Of Constructs

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 fail 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 senior reference is assigned to the cell of the more junior reference. A cell in the global stack is always more senior than any cell in the local stack. 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

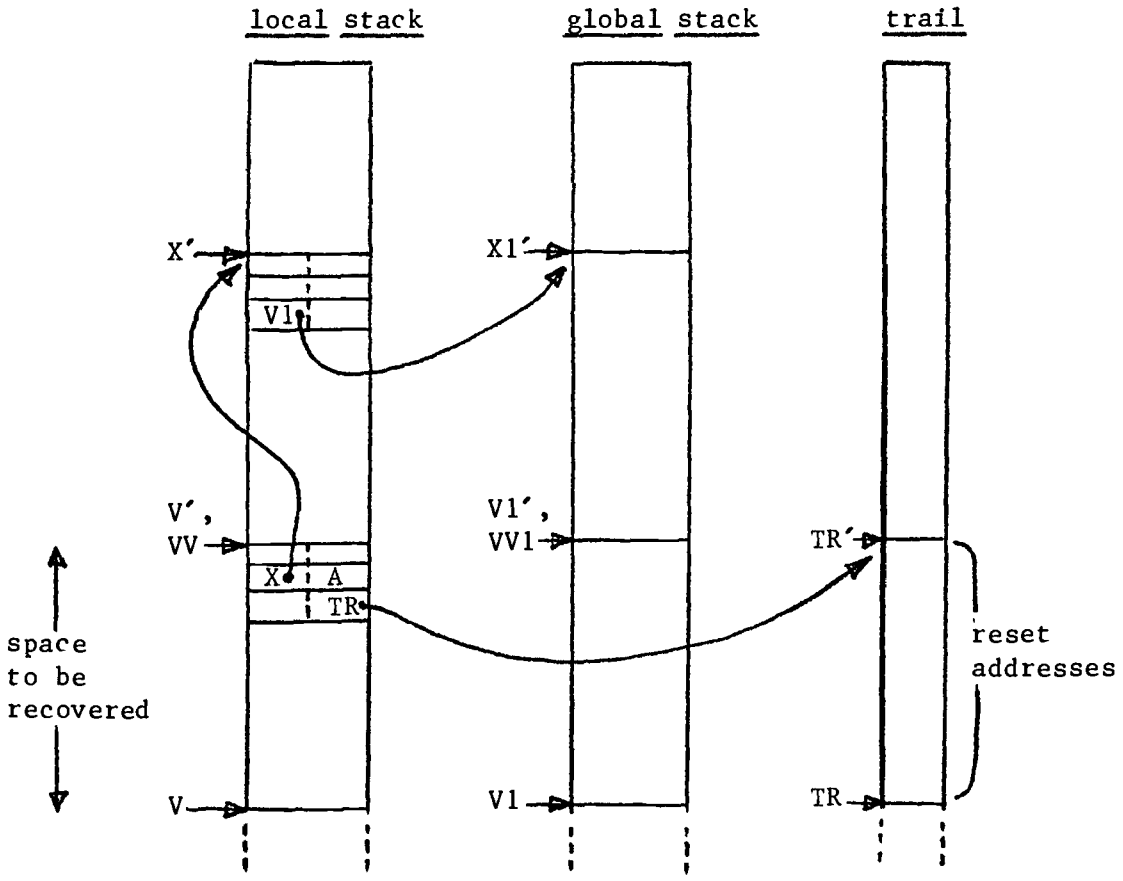
reference" is a well known nightmare where a location is left containing the address of a part of storage which has been de-allocated from its original use.) The rules also play an important role for efficiency in tending to prevent long chains of references being built up. In typical Prolog programs it is quite rare for dereferencing beyond the first step to be necessary, if the above scheme is applied.

{The reader may find it interesting to compare our treatment of variable-variable unification with Rem's algorithm described by Dijkstra [1976,pp.161-167]. In fact, our treatment was devised before reading Dijkstra's account. It is interesting that the same "seniority" concept is used for (primarily) quite different reasons.}

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.

4.7 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'.



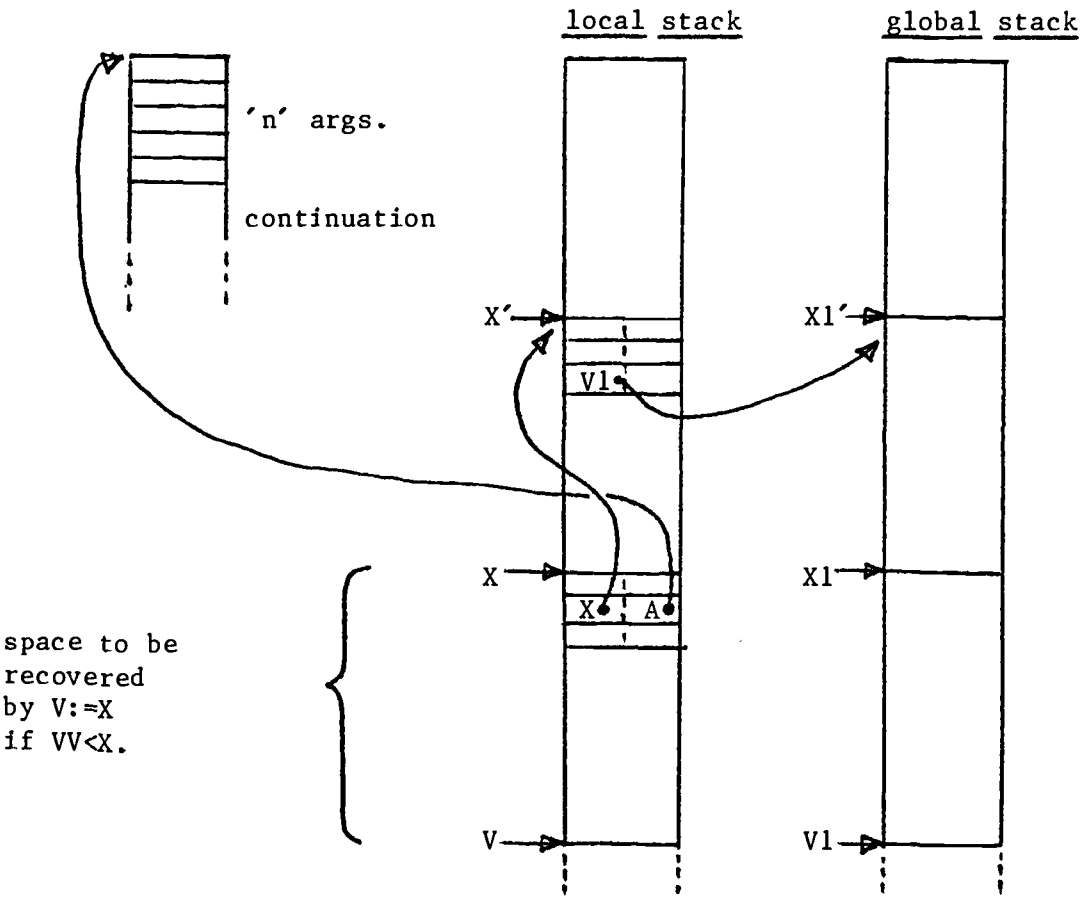
The PLM keeps an up-to-date record of the environment to backtrack to in registers VV and VV1. VV contains the address of the local frame, VV1 the address of the global frame. Note that VV1 is strictly superfluous since it merely shadows the contents of the V1 field in the local frame indicated by VV. 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.

4.8 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.



4.9 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 '*undef*'. 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 *void variables*.

(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 (*level*

number) 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 argument number 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 'undef', 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.

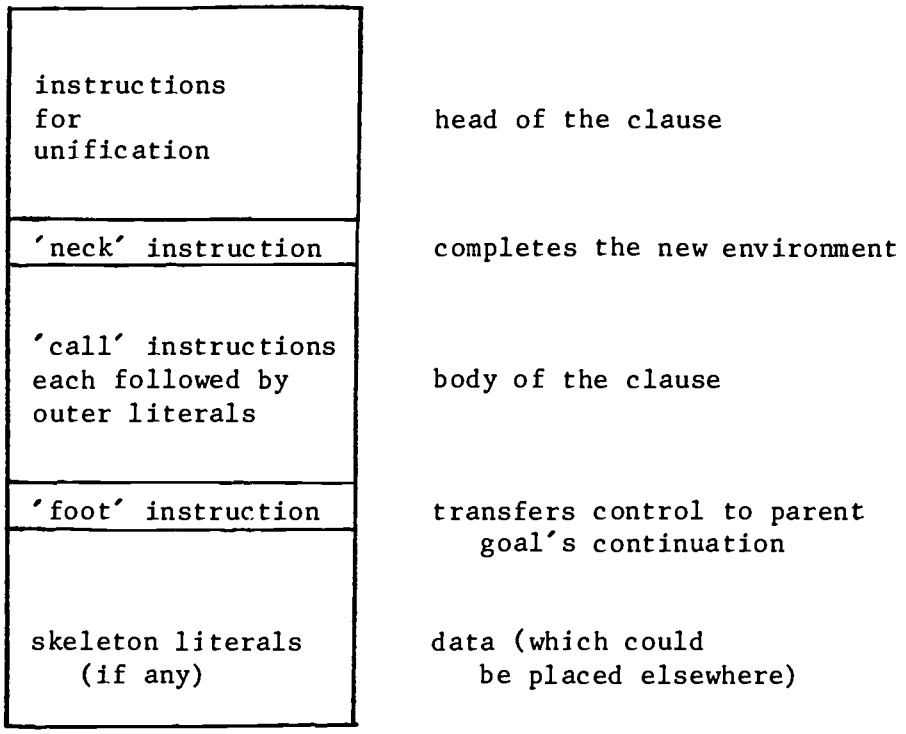
(6) By bringing together information from the different clauses in a procedure one can optimise the selection of potentially matching clauses and/or share part of the work involved in unifying with each clause head, and in addition provide a means of detecting the important case where the selection of 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:-



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)          uvar1(N,E,I)
uref(N,E,I)          uref1(N,E,I)
```

'uvar' or 'uvar1' is used if it is the first occurrence, 'uref' or 'uref1' otherwise. 'uvar' corresponds to level 1 and 'uvar1' to level 2, and similarly for all other pairs of instructions named 'name' and 'name1'. '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)           uatom1(N,I)
uint(N,I)            uint1(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 'undef'. 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.

4.10 Examples Of Prolog Machine Code

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

4.10.1

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)	
	uref1(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)
          uvar1(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)
```

4.10.2

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)
```

4.11 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.

4.12 More Examples Of Prolog Machine Code

4.12.1

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)
```

4.12.2

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(
	uvar1(0,global,0)	X,
	uvar1(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).
```

4.12.3

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
clause1:	uskeld(0,or)	rewrite(or(
	uvar1(0,global,0)	X,
	init(1,3)	
	uskell(1,label2)	and(Y,Z)),
	uskel(1,label3)	and(
	ifdone(label1)	
	uskell(0,label4)	or(X,Y),
	uskell(1,label5)	or(X,Z))
label1:	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)	

5.0 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 one 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(X1)
```

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 'undef', 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 all 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.

6.0 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.

6.1 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 indexed 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 and 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 local variable which is bound to the term passed as first argument in the call. If the variable at position 0 in the head is global, 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 reference code 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 non-reference code 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 independent of the number of clauses in the section.

In more detail, instruction 'switch' computes a key 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 'fail' replacing all occurrences of the 'Next' address. If control is transferred to 'fail', 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 'error'. If control is transferred to 'error' a diagnostic message is given followed by deep backtracking. If the restriction is '-', the non-reference code is replaced by the instruction 'goto(error)'. Thus the procedure code checks that the type of the first argument is consistent with any mode declaration.

6.1.1 Example -

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(ref1,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)
```

6.2 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:-

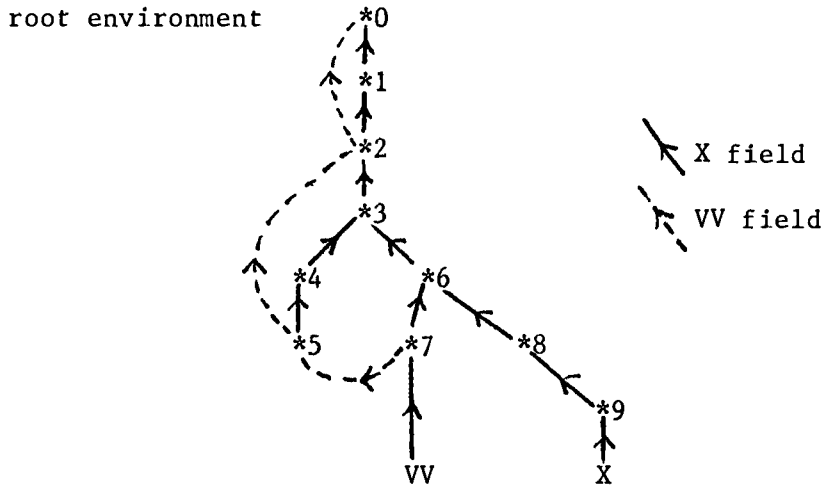
(1) An extra GC bit must be made available in (or associated with) each global cell. This bit will be set during the trace and mark phase if the cell is to be retained.

(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 Vl 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.
```

7.0 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 their 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 practicable 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.

8.0 PERFORMANCE

8.1 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.

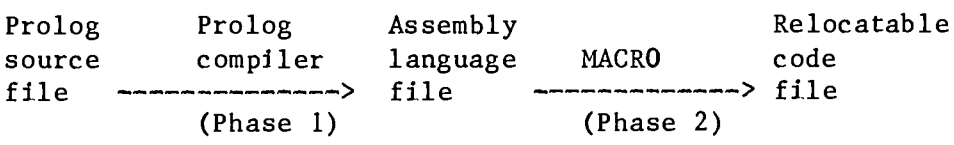
We shall now summarise the results of these benchmark tests and other less direct performance data. Firstly, comparing Prolog implementations, one can say that compilation has improved running speed by a factor of (typically) 15 to 20 relative to the Marseille interpreter. The improvement is greater 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 largely "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:-



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.

8.2 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 all 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 goto

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.

9.0 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 implementation.

APPENDICES

1.0 PLM REGISTERS, DATA AREAS AND DATA STRUCTURES

Here we summarise the state of the PLM during unification. Recall that the machine is attempting to match the head of the current clause against the current goal. A failure to unify will cause backtracking to the latest choice point where the parent goal will be reconsidered.

Registers

V top of local stack = local frame for current clause

V1 top of global stack = global frame for current clause

X local frame for current goal

X1 global frame for current goal

VV local frame for latest choice point

VV1 global frame for latest choice point

TR pushdown list pointer for the trail

PC current instruction

A arguments and continuation of current goal

B a skeleton involved in unification

Y the global frame corresponding to B

Other registers used in the DEC10 implementation

FL failure label, but only when VV=V

T construct passed as argument to a unification routine

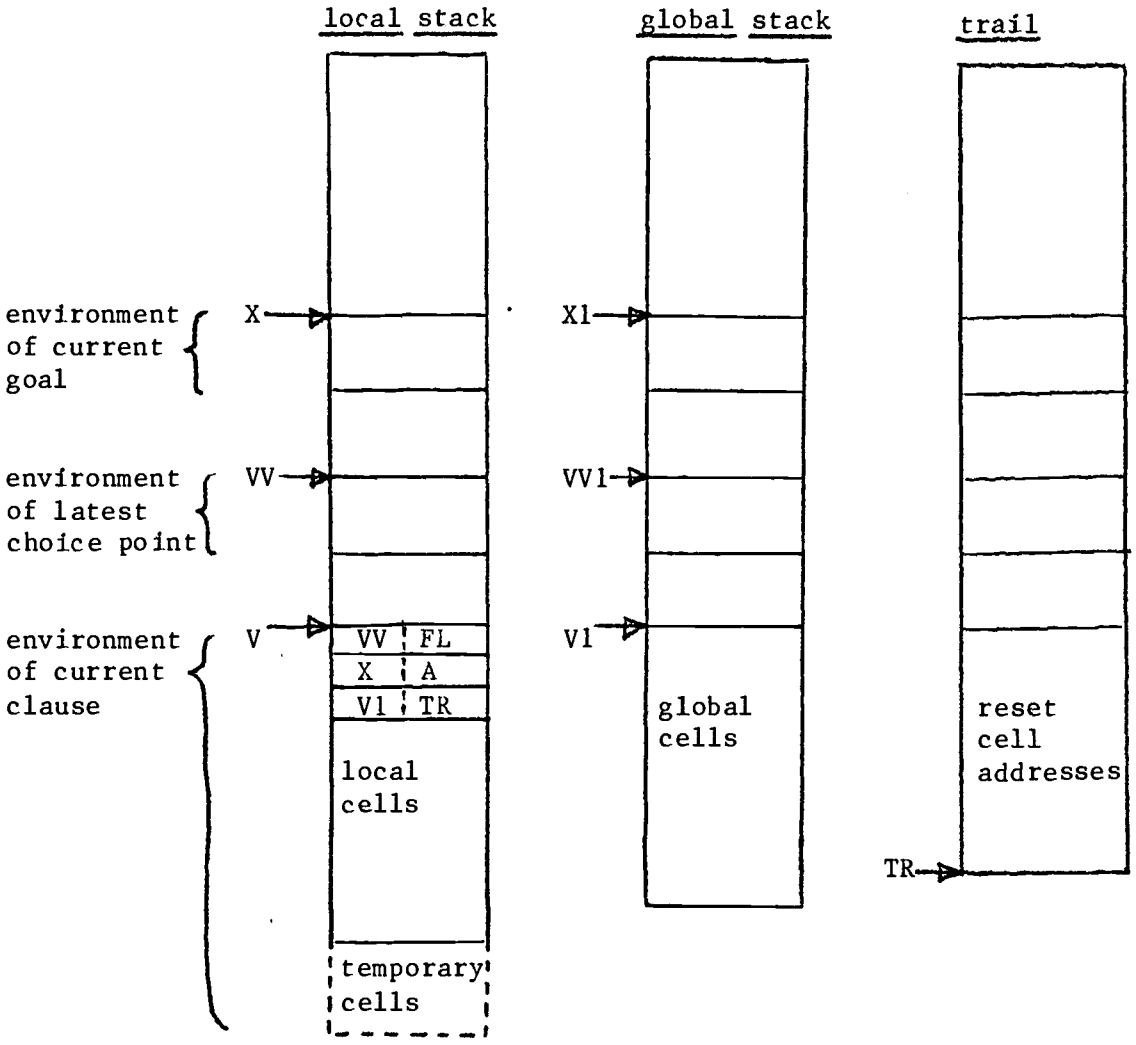
B1 construct passed as argument to a unification routine

C return address for a runtime routine

R1 temporary results

R2 temporary results

Data areas and environment layout



Fields of an environment

A parent goal's arguments and continuation

X parent goal's local frame

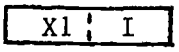
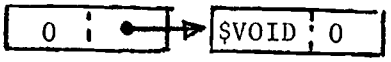
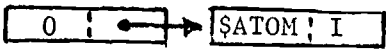
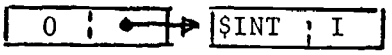
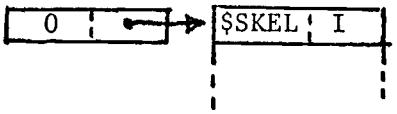
V1 global frame corresponding to this local frame

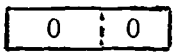
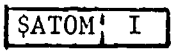
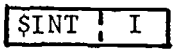
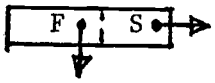
TR state of TR when parent goal was invoked

FL failure label, if any, for parent goal; ie. an alternative clause

VV local frame for the choice point prior to the parent goal

Representations for source and constructed terms

Source term (literal)	DEC10 form
var(I)	
local(I)	
global(I)	
void	
[atom(I)]	
[int(I)]	
[fn(I),...]	

Constructed term (cell value)	DEC10 form
undef	
ref(L)	
atom(I)	
int(I)	
mol(S,F)	

2.0 PLM INSTRUCTIONS AND LITERALS

2.1 Summary

literals

```
var(I)                  local(I)
atom(I)                 global(I)
int(I)                  void
fn(I)
```

unification

```
uvar(N,F,I)             uvarl(N,F,I)
uref(N,F,I)             urefl(N,F,I)
uatom(N,I)              uatoml(N,I)
uint(N,I)               uintl(N,I)
uskel(N,S)              uskell(N,S)
uskeld(N,I)
uskelc(N,S)             init(I,J)
                        localinit(I,J)
```

control transfer

```
ifdone(L)
call(L)
try(L)
trylast(L)
```

"red tape"

```
enter                   cut(I)
neck(I,J)               neckcut(I,J)
foot(N)                 neckcutfoot(J,N)
neckfoot(J,N)           fail
```

extra instructions for clause indexing

```
gsect                   switch(N)
ssect(L,C)              case(L)
ssectlast(L)
endssect                ifatom(I,L)
                        ifint(I,L)
ugvar(I)                iffn(I,L)

tryatom(I,C)            goto(L)
tryint(I,C)             notlast
tryskel(S,C)
trylastatom(I,C)
trylastint(I,C)
trylastskel(S,C)
```

2.2 var(I)

Use: An occurrence of a variable in a skeleton. I is the number of the global variable.

Example: 'var(2)' for:-

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

Effect: Serves as a pointer to a construct which is the value of the global variable.

DEC10 form:

```
            WD i(Y)          ;where i=I.
```

2.3 atom(I)

Use: An occurrence of an atom in a skeleton or goal is represented by the address of a literal 'atom(I)' where I identifies the atom.

Example: '[atom(nil)]' for:-

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

Effect: The address of the atom literal serves as a pointer to a construct representing the atom.

DEC10 form:

```
            WD label

label:      XWD $ATOM,i      ;where i = functor number of atom.
```

2.4 int(I)

Use: An occurrence of an integer in a skeleton or goal is represented by the address of a literal 'int(I)' where I is the value of the integer.

Example: '[int(29)]' for:-

```
leapyear(X) :- duration(february,X,29).
                                    **
```

Effect: The address of the integer literal serves as a pointer to a construct representing the integer.

DEC10 form:

```
            WD label

label:  XWD $INT,i        ;where i=I.
```

2.5 fn(I)

Use: An occurrence of a skeleton term in a goal or in a non-mode '+' position in the head of a clause is represented by the address of a skeleton literal, which commences with a functor literal 'fn(I)' where I identifies the functor of the skeleton.

Example: '[fn(cons),var(0),var(2)]' for:-

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

Effect: The address of the skeleton literal serves as the skeleton component of the molecule which represents the subterm.

DEC10 form:

```
            WD label

label:  XWD $SKEL,i       ;where i = skeleton's functor number.
        ...               ;inner literals
```

2.6 local(I)

Use: An occurrence of a local variable as an argument of a goal. I is the number of the local variable.

Example: 'local(0)' for:-

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

Effect: Serves as a pointer to a construct which is the value of the local variable.

DEC10 form:

```
WD i(X)          ;where i=I+3.
```

2.7 global(I)

Use: An occurrence of a global variable as an argument of a goal. I is the number of the global variable.

Example: 'global(1)' for:-

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

Effect: Serves as a pointer to a construct which is the value of the global variable.

DEC10 form:

```
WD i(X1)         ;where i=I.
```

2.8 void

Use: An occurrence of a void variable (ie. the variable occurs nowhere else) as an argument of a goal.

Example: 'void' for:-

```
employed(X) :- employs(Y,X).
                        *
```

Effect: Any instruction which attempts to unify against this outer literal behaves as a (successful) no-operation.

DEC10 form:

```
            WD label

label:      XWD $VOID,0
```

2.9 uvar(N,F,I)

Use: Argument N in the head of a clause is the first occurrence of a variable of type F (local or global), number I. (A temporary variable will have F=local.)

Example: 'uvar(1,global,2)' for:-

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

Effect: The outer literal representing argument N of the current goal is accessed via register A and the dereferenced result is assigned to cell I in frame F of the current environment, unless the result is a local reference and F is global. In the latter case, a reference to cell I in frame E is assigned to the incoming reference, and the assignment is trailed if necessary.

DEC10 form:

```
        MOVE T,@n(A)      ;where n=N.
        TLNN T,$1MA
        JSP C,$UVAR
        MOVEM T,i(reg)    ;where i=I+3 and reg=V if F=local
                          ;or    i=I and reg=V1 if F=global.
```

If N<9 and fastcode is not required, this is condensed to:-

```
        JSP C,routine
        MOVEM T,i(reg)

routine: MOVE T,@n(A)
        TLNN T,$1MA
        JSP C1,...
        JRST 0(C)
```

2.10 uvarl(N,F,I)

Use: Argument N of a skeleton at level l in the head of a clause is the first occurrence of a variable of type F (local or global), number I. The instruction is not needed if the skeleton is in a mode '-' position.

Example: 'uvarl(1,local,0)' for:-

```
:-mode reverse(+,+,?).
reverse(cons(X,L1),L2,L3) :- reverse(L1,cons(X,L2),L3).
                **
```

Effect: The inner literal representing argument N of the matching skeleton is accessed via register B and the dereferenced result is assigned to cell I in frame F of the current environment. Note: if the result is a reference, it must refer to a global cell, which will therefore be at least as senior as the cell assigned.

DEC10 form:

```
        MOVE T,@n(B)      ;where n=N+1.
        TLNN T,$1MA
        JSP C,$UVAR1
        MOVEM T,i(reg)    ;where i=I+3 and reg=V if F=local
                          ;or    i=I and reg=V1 if F=global.
```

If N<5 and fastcode is not required, this is condensed to:-

```
        JSP C,routine
        MOVEM T,i(reg)

routine: MOVE T,@n(B)
        TLNN T,$1MA
        JSP C1,...
        JRST 0(C)
```

2.11 uref(N,F,I)

Use: Argument N in the head of a clause is a subsequent occurrence of a variable of type F (local or global), number I. (A temporary variable will have F=local.)

Example: 'uref(2,local,0)' for:-

```
reverse(nil,L,L).
            *
```

Effect: The outer literal representing argument N of the current goal is accessed via register A and the dereferenced result is unified with the dereferenced value of cell I in frame F of the current environment.

DEC10 form:

```
        MOVE B,@n(A)      ;where n=N.
        MOVE B1,i(reg)    ;where i,reg are as for 'uvar'.
        JSP C,$UREF
```

If N<5 this is condensed to:-

```
        MOVE B1,i(reg)
        JSP C,routine

routine: MOVE B,@n(A)
         ...
```

2.12 urefl(N,F,I)

Use: Argument N of a skeleton at level 1 in the head of a clause is a subsequent occurrence of a variable of type F (local or global), number I. The instruction is not needed if the skeleton is in a mode '-' position.

Example: 'urefl(0,global,0)' for:-

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

Effect: The inner literal representing argument N of the matching skeleton is accessed via register B and the dereferenced result is unified with the dereferenced value of cell I in frame F of the current environment.

DEC10 form:

```
            MOVE T,@n(B)      ;where n=N+1.
            MOVE B1,i(reg)    ;where i,reg are as for 'uvar'.
            JSP C,$UREF1
```

If N<3 this is condensed to:-

```
            MOVE B1,i(reg)
            JSP C,routine

routine:    MOVE T,@n(B)
            ...
```

2.13 uatom(N,I)

Use: Argument N in the head of a clause is an atom, identified by I.

Example: 'uatom(1,september)' for:-

```
month(9,september).
        *********
```

Effect: The outer literal representing argument N of the current goal is accessed via register A and the dereferenced result is unified with atom I.

DEC10 form:

```
            MOVE T,@n(A)      ;where n=N.
            JSP C,$UATOM
            XWD $ATOM,i       ;where i = functor number of atom.

$UATOM:     TLNN T,$1MAS
            JRST ...
            CAME T,0(C)
            JRST $FAIL
            JRST 1(C)
```

If N<8 this is condensed to:-

```
            JSP C,routine
            XWD $ATOM,i

routine:    MOVE T,@n(A)
            TLNN T,$1MAS
            JSP C1,...
            CAME T,0(C)
            JRST $FAIL
            JRST 1(C)
```

2.14 uatoml(N,I)

Use: Argument N of a skeleton at level l in the head of a clause is an atom, identified by I.

Example: 'uatoml(1,nil)' for:-

```
singleton(cons(X,nil)).
                ***
```

Effect: The inner literal representing argument N of the matching skeleton is accessed via register B and the dereferenced result is unified with atom I.

DEC10 form:

```
        MOVE T,@n(B)      ;where n=N+1.
        JSP C,$UATOM
        XWD $ATOM,i       ;where i = functor number of atom.
```

If N<5 this is condensed to:-

```
        JSP C,routine
        XWD $ATOM,i

routine: MOVE T,@n(B)
        ...
```

2.15 uint(N,I)

Use: Argument N in the head of a clause is an integer, value I.

Example: 'uint(0,9)' for:-

```
month(9,september).
      *
```

Effect: The outer literal representing argument N of the current goal is accessed via register A and the dereferenced result is unified with integer I.

DEC10 form:

```
            MOVE T,@n(A)      ;where n=N.
            JSP C,$UATOM
            XWD $INT,i        ;where i = value of the integer.
```

If N<8 this is condensed to:-

```
            JSP C,routine
            XWD $INT,i

routine:    MOVE T,@n(A)
            TLNN T,$1MAS
            JSP C1,...
            CAME T,0(C)
            JRST $FAIL
            JRST 1(C)
```

2.16 uintl(N,I)

Use: Argument N of a skeleton at level l in the head of a clause is an integer, value I.

Example: 'uintl(1,2)' for:-

```
differentiate(square(X),X,*(X,2)).
                                  *
```

Effect: The inner literal representing argument N of the matching skeleton is accessed via register B and the dereferenced result is unified with nteger I.

DEC10 form:

```
        MOVE T,@n(B)      ;where n=N+1.
        JSP C,$UATOM
        XWD $INT,i        ;where i = value of the integer.
```

If N<5 this is condensed to:-

```
        JSP C,routine
        XWD $INT,i

routine: MOVE T,@n(B)
         ...
```

2.17 uskel(N,S)

Use: Argument N in the head of a clause is a skeleton term for which S is the address of a corresponding skeleton literal. (Not used for a mode '+' or mode '-' position.)

Example: 'uskel(2,[fn(cons),var(0),var(2)])' for:-

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

Effect: The outer literal representing argument N of the current goal is accessed via register A and dereferenced. If the result is a reference, a molecule is assigned to the cell referenced, the assignment is trailed if necessary and register Y is set to 'undef'. The molecule is constructed from S and the address of the current global frame given by register V1. If the result of the dereferencing is not a reference, a failure occurs unless the result is a molecule with the same functor as S. In the latter case register B is set to the address of the skeleton part of the matching molecule and register Y to the address of its (global) frame.

DEC10 form:

```
            MOVE B,@n(A)      ;where n=N.
            JSP C,$USK
            WD address        ;of literal S.

$USK:       HLRZ Y,B          ;load type of B into Y.
            CAIGE Y,$MOLS     ;if B isn't a molecule
            JRST @table(Y)    ;  switch on Y.
            MOVE R1,0(B)      ;load functor of B.
            CAME R1,@0(C)     ;if different from functor of S
            JRST $FAIL        ;  then fail.
            JRST 1(C)         ;return to in-line code.
```

If N<5 this is condensed to:-

```
            JSP C,routine
            WD address

routine:    MOVE B,@n(A)
            ...
```

2.18 uskel1(N,S)

Use: Argument N of a skeleton at level 1 in the head of a clause is another skeleton term for which S is the address of a corresponding skeleton literal.

Example: 'uskel1(0,[fn(int),var(0)])' for:-

```
expr(cons(int(N),S),S,N).
      ******
```

Effect: The inner literal representing argument N of the matching skeleton is accessed via register B and the dereferenced result is unified with the molecule formed from S and the global frame address in register Y.

DEC10 form:

```
          MOVE T,@n(B)      ;where n=N+1.
          JSP C,$USK1
          WD address        ;of literal S.

$USK1:    HLRZ R1,T
          CAIGE R1,$MOLS
          JRST @table(R1)
          MOVE R2,@0(C)
          CAME R2,0(T)
          JRST $FAIL
```

If N<3 this is condensed to:-

```
          JSP C,routine
          WD address

routine:  MOVE T,@n(B)
          ...
```

2.19 uskeld(N,I)

Use: Argument N in the head of a clause is a skeleton term, and this position has mode '+'. I identifies the functor of the skeleton term.

Example: 'uskeld(0,cons)' for:-

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

Effect: cf. 'uskel'. The result of dereferencing the matching outer literal is guaranteed to be a non-reference. A failure occurs unless it is a molecule with functor as indicated by I. Register B is set to the address of the skeleton part of the molecule and register Y to the address of the (global) frame.

DEC10 form:

```
            MOVE B,@n(A)      ;where n=N.
            JSP C,$USKD
            XWD $SKEL,i       ;cf. fn(I).

$USKD:      HLRZ Y,B          ;load type of B into Y.
            CAIGE Y,$MOLS     ;if B isn't a molecule
            JRST @table(Y)    ;  switch on Y.
            MOVE R1,0(B)      ;load functor of B.
            CAME R1,0(C)      ;if different from fn(I)
            JRST $FAIL        ;  then fail.
            JRST 1(C)         ;return to in-line code.
```

If N<5 this is condensed to:-

```
            JSP C,routine
            XWD $SKEL,i

routine:    MOVE B,@n(A)
```

2.20 uskelc(N,S)

Use: Argument N in the head of a clause is a skeleton term, and this position has mode '-'. S is the address of a corresponding skeleton literal.

Example: 'uskelc(2,[fn(cons),var(0),var(1)])' for:-

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

Effect: cf. 'uskel'. The result of dereferencing the matching outer literal is guaranteed to be a reference. A molecule formed from S with global frame address from V1 is assigned to the cell referenced and the assignment is trailed if necessary.

DEC10 form:

```
            MOVE B,@n(A)      ;where n=N.
            JSP C,$USKC
            WD address        ;of literal S.

    $USKC:  JUMPE B,UNDO      ;if B=undef goto UNDO.
    CONTINUE:                 CAILE B,$MAXREF ;if B is not a
 reference
            JRST 1(C)         ;  then it's a void so return.
            SKIPN R1,0(B)     ;if B is fully dereferenced
            JRST ASSIGN       ;  then goto ASSIGN.
            ...               ;else continue dereferencing.

    UNDO:   MOVEI B,@-2(C)    ;undo initial dereference step.
    ASSIGN: ...               ;proceed with assignment.
```

If N<8 this is condensed to:-

```
            JSP C,routine     ;call special subroutine.
            WD address        ;address of skeleton literal S.

    routine: SKIPE B,@n(A)    ;deref.arg.N into B _unless undef
            JRST CONTINUE     ;  goto CONTINUE.
            MOVEI B,@n(A)     ;load addr.of undef cell into B.
            JRST ASSIGN       ;goto ASSIGN
```

2.21 init(I,J)

Use: The instuction is used (a) following a 'uskel' or 'uskelc', or (b) preceding a 'uskell' which is an argument of a 'uskeld' instruction, or (c) preceding a 'neck'. I to J-1 inclusive are the numbers of global variables having their first occurrences in, respectively, (a) the level 1 skeleton or (b) the level 2 skeleton or (c) the body of the clause concerned. The instuction is omitted if there are no such variables (ie. I=J).

Example: The three different cases are illustrated by the use of 'init(1,2)' for each of:-

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

(b)  :-mode lookup(?,+,?).
     lookup(X,tree(_,pair(X,Y),_),Y).
                   *

(c)  member(X,L) :- concatenate(L1,cons(X,L2),L).
                 *
```

Effect: The cells for the global variables I through J-1 are initialised to 'undef'.

DEC10 form:

```
              SETZM n(V1)       ;for each n from I
              ...               ;            to J-1
```

If J-I > 2 this is condensed to:-

```
              MOVEI R1,j(V1)    ;where j=J.
              JSP C,routine

     routine: SETZM -i(R1)      ;where i=I.
              SETZM 1-i(R1)
              ...
              SETZM -1(R1)
              JRST 0(C)
```

2.22 localinit(I,J)

Use: Precedes a 'neck' instruction. The local variables which have their first occurrences within the body of the clause are numbered from I to J-1 inclusive. The instruction is omitted if there are no such variables (ie. I=J). Note if both an 'init' and a 'localinit' precede a 'neck' instruction, the order of the two is not important.

Example: 'localinit(1,2)' for:-

```
member(X,L) :- concatenate(L1,cons(X,L2),L).
               *
```

Effect: The cells for the local variables I through J-1 are initialised to 'undef'.

DEC10 form:

```
            SETZM n(V)        ;for each n from I+3
            ...               ;           to J+2
```

If J-I > 2 this is condensed to:-

```
            MOVEI R1,j(V)     ;where j=J+3.
            JSP C,routine

routine:    SETZM -i(R1)      ;where i=I+3.
            SETZM 1-i(R1)
            ...
            SETZM -1(R1)
            JRST 0(C)
```

2.23 ifdone(L)

Use: Precedes the instructions for the arguments of a level 1 skeleton (not occurring in a mode '+' or mode '-' position). L is the address following the last argument instruction.

Example: 'ifdone(labell)' for (cf. Section $$):-

```
member(X,cons(X,L)).
             *
```

Effect: If register Y contains 'undef', indicating that the skeleton has matched against a reference, control is transferred to label L, thereby skipping the argument instructions.

DEC10 form:

```
JUMPE Y,label     ;where label=L.
```

2.24 call(L)

Use: Corresponds to the predicate of a goal in the body of a clause. L is the address of the procedure code for the predicate.

Example: 'call(reverse)' for:-

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

Effect: The address of the outer literals and continuation which follows the 'call' instruction is assigned to register A and control is transferred to L.

DEC10 form:

```
JSP A,label       ;where label=L.
```

2.25 try(L)

Use: (a) In unindexed procedure code, each clause in the procedure is represented by an instruction 'try(L)' where 'L' is the address of the clause's code. These instructions are ordered as the corresponding clauses in the source program.
(b) The 'try' instruction is also used in indexed procedure code.

Effect: The address of the following instruction is stored in the FL field of the current environment and control is transferred to 'L'. (In our DEC10 implementation, the address is saved in register FL and is only stored in the FL field if and when the 'neck' instruction is reached.)

DEC10 form:

```
JSP FL,label      ;where label=L.
```

2.26 trylast(L)

Use: (a) In unindexed procedure code, it replaces the 'try(L)' instruction for the last clause in the procedure.
(b) The instruction is also used in indexed procedure code.

Effect: Registers VV and VV1 are reset to the values they held at the time the current goal was invoked. Control is transferred to 'L'.

DEC10 form:

```
HLRZ VV,0(V)      ;VV:=VV field of current env.
HLRZ VV1,2(VV)    ;VV1:=V1 field of the VV env.
JRST label        ;where label=L.
```

2.27 enter

Use: The first instruction in the procedure code for a predicate. It is executed immediately after a 'call' instruction.

Effect: The instruction is responsible for initialising the control information in a new environment. The VV,X,A,V1,TR fields in the local frame are set from the VV,X,A,V1,TR registers. Registers VV and VV1 are then set to the values of registers V and V1 respectively.

DEC10 form:

```
            JSP C,$ENTER

$ENTER:  HRLZM VV,0(V)      ;VV field set.
         HRLI A,(X)
         MOVEM A,1(V)       ;X,A fields set.
         HLRZM TR,R1
         HRLI R1,(V1)
         MOVEM R1,2(V)      ;V1,TR fields set.
         MOVEI VV,(V)       ;VV:=V.
         MOVEI VV1,(V1)     ;VV1:=V1.
         MOVEM TR,$TR0      ;save TR in location $TR0.
         JRST 0(C)          ;return.
```

2.28 neck(I,J)

Use: Precedes the body of a non-unit clause having I local variables (excluding temporaries) and J global variables.

Example: 'neck(1,1)' for:-

```
rterm(T,N,N,wd(Atom)) :- flagatom(T,Atom).
                         **
```

Effect: Registers X and X1 are set from registers V and V1 respectively. The contents of registers V and V1 are then incremented by the sizes of (the non-temporary part of) the local frame and of the global frame respectively. Both stacks are checked to ensure a sufficient margin of free space.

DEC10 form:

```
            JSP C,$NECK
            WD i(V)           ;where i=I+3.
            WD j(V1)          ;where j=J.

$NECK:      HRRM FL,0(V)      ;set FL field in local frame.
            MOVEI X,(V)       ;X:=V.
            MOVEI X1,(V1)     ;X1:=V1.
            MOVEI V,@0(C)     ;V:=V+i.
            MOVEI V1,@1(C)    ;V1:=V1+j.
            CAMLE V,$VMAX     ;if insufficient local freespace
            JSP R1,..         ;  call subroutine.
            CAMLE V1,$V1MAX   ;if insufficient global freespace
            JSP R1,...        ;  call subroutine.
            JRST 2(C)         ;return to in-line code.
```

If J=0 this is condensed to:-

```
            JSP C,$NECK1
            WD i(V)

$NECK1:     HRRM FL,0(V)
            MOVEI X,(V)
            MOVEI V,@0(C)
            CAMLE V,$VMAX
            JSP R1,...
            JRST 1(C)
```

If J=0 and I<5 this is further condensed to:-

```
            JSP C,routine

routine:    HRRM FL,0(V)
            MOVEI X,(V)
            MOVEI V,i(V)      ;where i=I+3.
            CAMLE V,$VMAX
            JSP R1,...
            JRST 0(C)
```

2.29 foot(N)

Use: At the end of a non-unit clause for a predicate of arity N.

Example: 'foot(3)' for:-

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

Effect: If the register VV indicates a point on the local stack earlier than register X, V is assigned the contents of X. Thus a determinate exit from the current procedure results in all the local storage used during the process being recovered. A, X and X1 are reset from the corresponding field in the parent local frame pointed to by X, and control is transferred to the parent's continuation.

DEC10 form:

```
            JSP C,$FOOT
            WD n(A)             ;where n=N.

$FOOT:      CAILE X,(VV)        ;if X>VV
            MOVEI V,(X)         ;  then V:=X.
            MOVE A,1(V)         ;reset A from parent.
            HLRZM A,X           ;reset X from parent.
            HLRZ X1,2(X)        ;reset X1 from parent.
            JRST @0(C)          ;goto parent's continuation.
```

If N<9 this is condensed to:-

```
            JRST routine

routine:    CAILE X,(VV)
            MOVEI V,(X)
            MOVE A,1(X)
            HLRZM A,X
            HLRZ X1,2(X)
            JRST n(A)           ;where n=N.
```

2.30 neckfoot(J,N)

Use: The last instruction for a unit clause. It replaces a 'neck(0,J)' followed by a 'foot(N)' where 'J' is the number of global variables and N is the arity of the predicate of the clause. (Note that a unit clause has no non-temporary local variables.)

Example: 'neckfoot(0,3)' for:-

```
concatenate(nil,L,L).
                    *
```

Effect: The instruction combines the effect of the 'neck' and 'foot' instructions it replaces. However considerable computation is saved; registers A, X and X1 do not have to be modified. Registers V1 and (if a non-determinate exit) V are incremented to take account of the new global and local frames, and control is transferred to the parent's continuation.

DEC10 form:

```
            MOVEI V1,j(V1)    ;where j=J.
            JSP C,$NKFT
            WD n(A)           ;where n=N.

$NKFT:      CAILE V,(VV)      ;if V>VV (ie. determinate exit)
            JRST BELOW        ;  then skip to BELOW.
            HRRM FL,0(V)      ;set FL field in local frame.
            MOVEI V,3(V)      ;V:=V+3.
            CAMLE V,$VMAX     ;if insufficient local freespace
            JSP R1,...        ;  call subroutine.
BELOW:      CAMLE V1,$V1MAX   ;if insufficient global freespace
            JSP R1,...        ;  call subroutine.
            JRST @0(C)        ;goto parent's continuation.
```

If J=0 this is condensed to:-

```
            CAIG V,(VV)       ;if V =< VV (ie. non-determinate)
            JSP C,$NKFT0      ;  then call subroutine $NKFT0
            JRST n(A)         ;  else goto parent's continuation.

$NKFT0:     HRRM FL,0(V)
            MOVEI V,3(V)
            CAMLE V,$VMAX
            JSP R1,...
            JRST @0(C)
```

If J=0 and N<9 this is further condensed to:-

```
            JRST routine

routine:    CAIG V,(VV)
            JSP C,$NKFT0
            JRST n(A)
```

2.31 cut(I)

Use: Corresponds to an occurrence of the cut symbol. I is the number of local variables (excluding temporaries) in the clause, as for the instruction 'neck(I,J)'.

Example: 'cut(2)' for:-

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

Effect: Any remaining local frames created since the environment of the current clause are discarded by resetting register V to point at the end of the current local frame. Registers VV and VV1 are reset to the backtrack environment of the parent. The portion of the trail created since the parent goal is "tidied up" by discarding references to variables if they don't belong to environments before the backtrack environment.

DEC10 form:

```
            MOVEI V,i(V)      ;V:=V+i where i=I+3.
            JSP C,$CUT

$CUT:       CAILE X,(VV)      ;if no alternatives to cut
            JRST 0(C)         ;  then return.
            HLRZ VV,0(X)      ;reset VV from parent.
            HLRZ VV1,2(VV)    ;reset VV1 from parent.
            HRRE P,2(X)       ;P:= lh of TR for parent.
            ADD P,$TRTOP      ;P:= rh of TR for parent.
            CAIN P,(TR)       ;if no change to TR
            JRST 0(C)         ;  then return.
            MOVEI P1,(TR)     ;P1:=rh of TR for parent.
            MOVEI R1,(TR)     ;
            SUBI R1,(P)       ;R1:=delta=increase in trail size.
            HRLI R1,(R1)      ;reset TR to its original value:-
            SUB TR,R1         ;  TR:=TR-(delta,delta).
CYCLE:      MOVE R1,1(P)      ;load one of the new trail entries.
            CAIL R1,(VV)      ;if refers after VV
            JRST CONTINUE     ;  then continue with next entry.
            CAIGE R1,(V1)     ;if refers after V1 (ie. is local)
            CAIGE R1,(VV1)    ;or before VV1
            PUSH TR,(R1)      ;  then restore it to trail.
CONTINUE:   CAIE P1,1(P)      ;if more trail entries to consider
            AOJA P,CYCLE      ;  then P:=P+1, goto CYCLE.
            JRST 0(C)         ;return.
```

If I<10 this is condensed to:-

```
            JSP C,routine

routine:    MOVEI V,i(V)
            JRST $CUT
```

2.32 neckcut(I,J)

Use: Corresponds to a cut symbol which is the first "goal" in the body of a non-unit clause. It replaces a 'neck(I,J)' followed by a 'cut(I)' where 'I' and 'J' are the numbers of local and global variables respectively.

Example: 'neckcut(0,0)' for:-

```
divide(X,0,Y) :-!, error('division by 0').
                ***
```

Effect: The instruction combines the effects of the corresponding 'neck' and 'cut' instructions in a straightforward way.

DEC10 form:

```
            JSP C,$NCUT
            WD i(V)          ;where i=I+3.
            WD j(V1)         ;where j=J.
```

If J=0 this is condensed to:-

```
            JSP C,$NCUT1
            WD i(V)
```

If J=0 and I<5 this is further condensed to:-

```
            JSP C,routine

   routine: MOVEI X,(V)
            MOVEI V,i(V)     ;where i=I+3.
            JRST ...
```

2.33 neckcutfoot(J,N)

Use: Corresponds to a cut symbol which is the only "goal" in the body of a clause. It replaces instructions 'neck(0,J)' followed by 'cut(0)' followed by 'foot(N)' where 'J' is the number of global variables and N is the arity of the predicate of the clause.

Example: 'neckcutfoot(0,2)' for:-

```
factorial(0,1):-!.
                ****
```

Effect: Combines the effect of the three instructions it replaces. As with 'neckfoot', considerable computation is saved since registers A, X and X1 do not have to be modified. Register V1 is incremented to take account of the new global frame, registers VV and VV1 are reset to their states prior to invoking the parent goal and trail entries are discarded where possible. Finally control is transferred to the parent's continuation.

DEC10 form:

```
            MOVEI V1,j(V1)    ;where j=J.
            JSP C,$NCTF
            WD n(A)           ;where n=N.

$NCTF:      CAMLE V1,$V1MAX   ;if insufficient local freespace
            JSP R1,...        ;  call subroutine.
            CAILE V,(VV)      ;if V>VV (already determinate)
            JRST @0(C)        ;  then goto parent's continuation.
$NCTF0:     HLRZ VV,0(V)      ;reset VV.
            HLRZ VV1,2(VV)    ;reset VV1.
            .                 ;
            .                 ;perform rest of cut.
            .                 ;
            JRST @0(C)        ;goto parent's continuation.
```

If J=0 this is condensed to:-

```
            CAIG V,(VV)       ;if V =< VV (not already determinate)
            JSP C,$NCTF0      ;  then call subroutine $NCTF0.
            JRST n(A)         ;goto parent's continuation.
```

2.34 fail

Use: Corresponds to a goal 'fail' in the body of a clause. This goal is defined to be unsolvable and instigates (deep) backtracking.

Example: 'fail' for:-

```
unknown(X) :- known(X),!,fail.
                          ****
```

Effect: Registers V, V1, A, X and X1 are reset to the values they had prior to the most recent goal which is non-determinate (ie. one for which there are still further choices available). Register TR is also restored to its earlier value by popping entries off the trail and resetting the cells referenced to 'undef'. Finally control is transferred to the clause which is the next choice for the earlier goal.

DEC10 form:

```
            JRST $EFAIL

$EFAIL:  MOVEI V,(VV)      ;V:=VV
         MOVEI V1,(VV1)    ;V1:=VV1
         HRRZ FL,0(V)      ;FL:=next clause for earlier goal.
         MOVE A,1(V)       ;reset A
         HLRZM A,X         ;reset X
         HLRZ X1,2(X)      ;reset X1
         HRRE R1,2(V)      ;R1:=lh of earlier TR.
         ADD R1,$TRTOP     ;R1:=rh of earlier TR.
         CAIN R1,(TR)      ;if no change to TR
         JRST EXIT         ;  then goto EXIT.
CYCLE:   POP TR,R2         ;pop an entry off the trail.
         SETZM (R2)        ;set cell refd. to 'undef'.
         CAIE R1,(TR)      ;if more trail entries to consider
         JRST CYCLE        ;  then goto CYCLE.
EXIT:    MOVEM TR,$TRO     ;save TR in location $TRO
         JRST @FL          ;goto next clause.
```

Note in passing that a failure in a unification instruction causes control to be transferred to a routine $FAIL which instigates shallow backtracking:-

```
$FAIL:   CAIE V,(VV)       ;if V = VV (no other choices)
         JRST $EFAIL       ;  then deep backtracking.
         CAMN TR,$TRO      ;if no trail entries from this unifn.,
         JRST @FL          ;  then goto next clause.
CYCLE1:  POP TR,R2         ;pop an entry off the trail.
         SETZM (R2)        ;set the cell refd. to 'undef'.
         CAME TR,$TRO      ;if more trail entries to consider
         JRST CYCLE1       ;  then goto CYCLE1.
         JRST @FL          ;goto next clause.
```

2.35 gsect

Use: Precedes a general section of clauses having a variable at position 0 in the head.

Effect: The outer literal representing argument 0 of the current goal is accessed via register A and the dereferenced result is assigned to cell 0 in the current local frame.

DEC10 form:

```
            JSP C,$GS

$GS:        MOVE B,@0(A)      ;B := arg. 0
            HLRZM B,Y         ;Y := type of arg. 0
            CAIG Y,$SKEL      ;if arg. 0 is not a molecule
            JRST @table(Y)    ;  then switch on type.
            MOVEM B,3(V)      ;local cell 0 := arg. 0
            JRST 0(C)         ;return.
```

In practice the code is optimised by
(1) coalescing the code for 'enter' immediately followed by 'gsect',
(2) 'ssect' initialises local cell 0 as a side effect so that 'gsect' doesn't have to be called if no clauses in the special section are entered,
(3) 'endssect' performs the work of 'gsect' if the matching term is a reference so 'gsect' only needs to handle the non-reference case.

2.36 ssect(L,C)

Use: Precedes a special section of clauses having a non-variable at position 0 in the head. L is the address of the reference code for the section and C is the address of the section which follows.

Effect: If the dereferenced value of argument 0 in the current goal is a reference, control is transferred to L. Otherwise register FL is set to C and control passes to the non-reference code which follows the 'ssect' instruction.

DEC10 form:

```
        JSP C,$SS
        WD refcode          ;where refcode=L
        WD nextsection      ;where nextsection=C

$SS:    MOVE B,3(V)         ;B := arg.0 from local cell 0
        HLRZM B,Y           ;Y := type of arg. 0
        JUMPE Y,@0(C)       ;if arg.0 is a ref goto refcode.
        MOVE FL,1(C)        ;FL := nextsection
        MOVEM B,R2          ;R2 := arg. 0
        CAIL Y,$MOLS        ;if arg.0 is a molecule
        MOVE R2,0(B)        ;  then R2 := functor of arg.0
        JRST 2(C)           ;return to non-reference code.
```

The above is an optimisation, used only if it is not the first section in the procedure. 'enter' immediately followed by 'ssect' is treated as a special case. Register R2 is set to the address of the atom, integer or functor literal for argument 0. If argument 0 is a reference, this is trailed once and for all to avoid repeated "trailing" for each of the clauses in the section.

2.37 ssectlast(L)

Use: Precedes a special section which is the last section of a procedure. L is the address of the reference code for the section.

Effect: If the dereferenced value of argument 0 in the current goal is a reference, control is transferred to L. Otherwise registers VV and VV1 are reset to the values they held at the time the current goal was invoked and control passes to the reference code which follows the 'ssectlast' instruction.

DEC10 form:

```
            JSP C,$SS1
            WD refcode          ;where refcode=L

$SS1:       MOVE B,3(V)         ;B := arg.0 from local cell 0.
            HLRZM B,Y           ;Y := type of arg.0
            JUMPE Y,@0(C)       ;if arg.0 is a ref. goto refcode.
            HLRZ VV,0(V)        ;VV := VV field of current env.
            HLRZ VV1,2(VV)      ;VV1 := V1 field of the VV env.
            MOVEM B,R2          ;R2 := arg.0
            CAIL Y,$MOLS        ;if arg.0 is a molecule
            MOVE R2,0(B)        ;  then R2 := functor of arg.0.
            JRST 1(C)           ;return to non-reference code.
```

2.38 endssect

Use: Terminates the reference code at the end of a special section.

Effect: The reference passed as argument 0 is recovered from the trail and stored in local cell 0. The following 'gsect' instruction is skipped.

DEC10 form:

```
            JSP C,$ENDRC

$ENDRC:     POP TR,R1           ;pop last trail entry into R1.
            MOVEM TR,$TR0       ;TR0 := TR.
            SOS 2(V)            ;correct TR field of current env.
            SETZM (R1)          ;set cell referenced to undef.
            MOVEM R1,3(V)       ;local cell 0 := the reference.
            JRST 1(C)           ;return, skipping one instruction.
```

2.39 switch(N)

Use: Precedes the non-reference code in a special section if there is a sufficient number of clauses in the section (currently 5 or more). N is the number of 'case' instructions which follow and is a power of 2 chosen depending on the number of clauses in the section.

Effect: A key, determined by the principal functor of argument 0 of the current goal, is "anded" with N-1 to give a value M. Control is then transferred to the (M+1)th. 'case' instruction.

DEC10 form:

```
            MOVEI R1,(R2)     ;R1 := key
            ANDI R1,n-1       ;R1 := key/(N-1)
            JRST @NEXT(R1)    ;goto case (R1)
    NEXT:
```

2.40 case(L)

Use: A 'switch(N)' instruction is followed by N 'case' instructions. The parameter L is the address of the code for the subset of the section's clauses corresponding to that case.

Effect: Control is transferred to address L by the preceding 'switch' instruction.

DEC10 form:

```
            WD label          ;where label=L.
```

2.41 ifatom

Use: In the non-reference code of a special section, the clause(s) for the atom identified by I is indicated by address L.

Effect: If argument 0 of the current goal is atom I, control is transferred to address L.

DEC10 form:

```
CAMN R2,atom      ;where atom = addr. of atom I literal.
JRST label        ;where label=L.
```

2.42 ifint(I,L)

Use: In the non-reference code of a special section, the clause(s) for integer I is indicated by address L.

Effect: If argument 0 of the current goal is integer I, control is transferred to address L.

DEC10 form:

```
CAMN R2,int       ;where int = integer I literal
JRST label        ;where label=L.
```

2.43 iffn(I,L)

Use: In the non-reference code of a special section, the clause(s) for the functor identified by I is indicated by address L.

Effect: If the principal functor of argument 0 of the current goal is functor I, registers B and Y are set according to this molecule and control is transferred to address L.

DEC10 form:

```
            CAMN R2,functor  ;where functor = addr of fn I literal.
            JRST label       ;where label=L.
```

Note that in the actual implementation, registers B and Y are set by 'ssect' or else by the following preceding the CAMN:-

```
            JSP C,$RLDSK

$RLDSK:     MOVE B,@0(A)     ;B := arg.0
            HLRZ Y,B         ;Y := type of arg.0
            CAIL Y,MOLS      ;if arg.0 is a molecule
            JRST 0(C)        ;  then return.
            JUMPE Y,DEREF    ;if arg.0 is a ref. goto DEREF.
            MOVEI B,@0(A)    ;B := skel. literal in the goal.
            HRLI B,(X1)      ;lh. of B := X1.
            MOVEI Y,(X1)     ;Y := X1.
            JRST 0(C)        ;return.

DEREF:      MOVE B,0(B)      ;B := deref B.
            HLRZ Y,B         ;Y := type of B.
            JUMPE Y,DEREF    ;if B is a ref. goto DEREF.
            JRST 0(C)        ;return
```

2.44 goto(L)

Use: (1) Following a sequence of 'if' instructions or a sequence of 'try' instructions in a special section, L is the address of the following section.

Effect: Control is transferred to address L.

DEC10 form:

```
        JRST label      ;where label=L.
```

2.45 notlast

Use: If there is more than one clause for a particular functor in a special section, the 'try' instructions are preceded by a 'notlast' instruction.

Effect: Registers VV and VV1 are reset from V and V1 respectively to indicate the current environment.

DEC10 form:

```
        JSP C,$NLAST

$NLAST: MOVEI VV,(V)      ;VV:=V
        MOVEI VV1,(V1)    ;VV1:=V1
        MOVEM             TR,$TR0 ;TR0:=TR
        JRST              0(C) ;return
```

2.46 ugvar(I)

Use: If argument 0 of the head of a clause is a global variable (and the procedure code is to be indexed), this term is represented by 'ugvar(I)' where I is the number of the global variable.

Effect: The construct which has been assigned to local cell 0 (by the corresponding 'gsect' instruction) is also assigned to global cell I unless the construct is a local reference. In the latter case global cell I is initialised to undef and a reference to global cell I is assigned to the local reference. This assignment is trailed if necessary.

DEC10 form:

```
            JSP C,$GTER1
            MOVEM T,i(V1)       ;global cell I := T

$GTER1:     MOVE T,3(V)         ;T := local cell 0
            CAIG T,MAXREF       ;if T not a reference
            CAIGE T,@0(C)       ;or T < global cell I
            JRST 0(C)           ;  then return.
            MOVEI R1,@0(C)      ;R1 := addr. of global cell I
            SETZM (R1)          ;global cell I := undef
            MOVEM R1,(T)        ;cell T := R1
            CAIGE T,(VV)        ;if T < VV
            PUSH TR,T           ;  then push T onto the trail
            JRST 1(C)           ;return, skipping 1 instr.
```

2.47 tryatom(I,C)

Use: In the reference code of a special section, a clause with atom I as argument 0 of the head is represented by the instruction 'tryatom(I,C)'. C is the address of the clause's code.

Effect: The atom is assigned to the matching reference and the assignment is trailed. Register FL is set to the address of the following instruction and control is transferred to C.

DEC10 form:

```
            JSP C,$RVAT
            XWD clause,atom  ;clause=C,atom= atom I.

$RVAT:      MOVEI FL,1(C)    ;FL := next PLM instr.
            MOVE R1,0(C)     ;R1 := (clause,atom)
            HLRZM R1,C       ;C := clause.
            MOVE R1,0(R1)    ;R1 := atom.
            MOVEM R1,@0(TR)  ;trailed ref. := atom.
            JRST 0(C)        ;goto clause.
```

Note that the matching reference has already been trailed by 'ssect'.

2.48 tryint(I,C)

Exactly analagous to 'tryatom'. The DEC10 form uses routine $RVAT also.

2.49 tryskel(S,C)

Use: In the reference code of a special section, a clause with a skeleton as argument 0 in the head is represented by 'tryskel(S,C)'. S is the address of the skeleton literal and C of the clause's code.

Effect: A molecule is formed from S with current global frame address from register V1 and assigned to the matching reference. The assignment is trailed. Register Y is set to 'undef' and register FL to the address of the following instruction. Control is transferred to C.

DEC10 form:

```
            JSP C,$RVSK
            XWD clause,skeleton ;clause=C,skeleton=S.

$RVSK:      MOVEI Y,0           ;Y := undef.
            MOVEI FL,1(C)       ;FL := next PLM instr.
            MOVE R1,0(C)        ;R1 := (clause,skeleton).
            HLRZM R1,C          ;C := clause.
            HRLI R1,(V1)        ;R1 := (V1,skeleton).
            MOVEM R1,@0(TR)     ;trailed ref. := (V1,skel.).
            JRST 0(C)           ;goto clause.
```

Note that the matching reference has already been trailed by 'ssect'.

2.50 trylastatom(I,C)

Use: If the final clause in a procedure has atom I as argument 0 of its head, the clause is represented in the reference code by the instruction 'trylastatom(I,C)', where C is the address of the clause's code. No 'endssect' is needed at the end of the section.

Effect: The atom is assigned to the matching reference but the assignment need not be trailed. Registers VV and VV1 are reset to indicate the previous backtrack point. Control is transferred to C.

DEC10 form:

```
            JSP C,$RVAT1
            XWD clause,atom   ;clause=C, atom = atom I.

$RVAT1:     HLRZ VV,0(V)      ;VV := VV field of current env.
            HLRZ VV1,2(VV)    ;VV1 := V1 field of the VV env.
            MOVE R1,0(C)      ;R1 := (clause,atom).
            HLRZM R1,C        ;C := clause.
            MOVE R1,0(R1)     ;R1 := atom.
            MOVEM R1,@0(TR)   ;trailed ref. := atom.
            JRST 0(C)         ;goto clause.
```

Note that the matching reference has already been trailed by 'ssect'.

2.51 trylastint(I,C)

Exactly analogous to 'trylastatom'. The DEC10 form uses routine $RVAT1 also.

2.52 trylastskel(S,C)

Use: If the final clause in a procedure has a skeleton as argument 0 of its head, the clause is represented in the reference code by an instruction 'trylastskel(S,C)', where S is the address of the skeleton literal and C is the address of the clause's code. No 'endssect' is needed at the end of the section.

Effect: A molecule is formed from S with the current global frame address from register V1 and assigned to the matching reference. The assignment need not be trailed. Register Y is set to 'undef'. Registers VV and VV1 are reset to indicate the previous backtrack point. Control is then transferred to C.

DEC10 form:

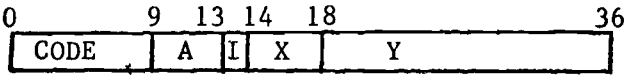
```
            JSP C,$RVSK1
            XWD clause,skeleton ;clause=C,skeleton=S.

$RVSK1:     MOVEI Y,0           ;Y := undef.
            HLRZ VV,0(V)        ;VV := VV field of current env.
            HLRZ VV1,2(VV)      ;VV1 := V1 field of the VV env.
            MOVE R1,0(C)        ;R1 := (clause,skeleton).
            HLRZM R1,C          ;C := clause.
            HRLI R1,(V1)        ;R1 := (V1,skeleton).
            MOVEM R1,@0(TR)     ;trailed ref. := (V1,skeleton).
            JRST 0(C)           ;goto clause.
```

Note that the matching reference has already been trailed by 'ssect'.

3.0 SYNOPSIS OF THE DEC SYSTEM 10

The machine has 36 bit words which can accommodate two 18 bit addresses. Addresses 0 to 15 refer to fast registers which are used as accumulators and (for 1 to 15 only) as index registers. Signed integers are represented as "2s complement" bit patterns. The instruction format is:-



where CODE = instruction code,
A = accumulator address,
I = indirection bit,
X = index register address,
Y = main address.

An instruction with I=1 is written symbolically in the form:-

CODE A,@Y(X)

If I=0 the '@' is omitted. If A=0 it can be omitted along with the comma. If X=0 it can be omitted along with the brackets. If Y=0 it can be omitted.

A fundamental mechanism is the "effective address calculation" which is the first step in the execution of each and every instruction. It computes an effective address E depending on I, X and Y. If X is nonzero, the contents of index register X is added to Y to produce a modified address M (modulo 2 to the power 18). If I=0 then simply E=M. If I=1, the addressing is indirect and E is derived by treating the I, X and Y fields of the word stored at M in exactly the same way. The process continues until a referenced location has I=0 and then E is calculated according to the X and Y fields of this location.

Two instructions, PUSH and POP, access pushdown lists which are stored in main memory. A pushdown list is referenced via a pushdown list pointer held in an accumulator. The right half of this word is the address of the current last item in the list. The left half (normally) contains the negative quantity M-L where M is the maximum size of the list and L is the current size.

The instructions referred to in this paper are summarised below. In all cases A is the accumulator address and E is the effective address computed as above. We write:-

(X) for "the contents of location X",
X.L for "the left half of location X"
X.R for "the right half of location X",
(X,Y) for "the word with left half X and right half Y",
sign X for "-1 if the top bit of X is 1 or 0 otherwise",
Y:=X for "location Y is assigned the value X".
skip for "skip the next instruction"

Instruction	Effect
MOVE A,E	A:=(E)
MOVEI A,E	A:=E
MOVEM A,E	E:=(A)
SETZM E	E:=0
ADD A,E	A:=(A)+(E)
SUB A,E	A:=(A)-(E)
SUBI A,E	A:=(A)-E
AOS E	E:=(E)+1
SOS E	E:=(E)-1
HLRZ A,E	A:=(0,(E.L))
HRRZ A,E	A:=(0,(E.R))
HLRZM A,E	E:=(0,(A.L))
HRLZM A,E	E:=((A.R),0)
HRLI A,E	A.L:=E
HRRM A,E	E.R:=(A.R)
HRRE A,E	A:=(sign (E.R), E.R)
CAIE A,E	if (A) = (0,E) then skip
CAIN A,E	if (A) ≠ (0,E) then skip
CAME A,E	if (A) = (E) then skip
CAMN A,E	if (A) ≠ (E) then skip
CAIG A,E	if (A) > (0,E) then skip
CAILE A,E	if (A) =< (0,E) then skip

```
CAIGE A,E       if (A) >= (0,E) then skip
CAMLE A,E       if (A) =< (E) then skip

SKIPE A,E       if A ≠ 0 then A:=(E), if (E)=0 then skip
SKIPN A,E       if A ≠ 0 then A:=(E), if (E)≠0 then $skip
TLNN A,E        if (A.L)/e ≠ 0 then skip

JRST E          goto E
JSP A,E         A:=(flags,address of next instruction), goto E
JUMPE A,E       if (A)=0 then goto E
AOJA A,E        A:=(A)+1, goto E

PUSH A,E        A:=(A)+(1,1); (A.R):=(E);
                if (A.L)=0 then interrupt
POP A,E         E:=((A.R)); A:=(A)-(1,1);
                if (A.L)=0 then interrupt

WD E            a non-executable address word with CODE=0
XWD X,Y         a non-executable data word containing (X,Y)
```

Constant Symbols

```
$VOID=1
$SKEL=2
$ATOM=4
$INT=5
$MOLS=16
$MAXREF=777777base8
$1MA=777764base8
$1MAS=777766base8
```

4.0 TIMING DATA FOR PLM INSTRUCTIONS ON DEC10

The times given below are the minimum times to complete the PLM instruction successfully. Cases where a failure to match occurs are not counted. Certain infrequent but faster special cases are also discounted (for example matching against a void).

The times relate to a KI10 processor and have been calculated from the data given on pages D-4 and D-5 of [DEC 1974]. An extra 1.02 microseconds has been allowed for each indirection and 0.89 microseconds for a control transfer or test instruction. All other factors (such as indexing) have been ignored.

Instruction	microsecs.	Remarks
uvar,uvarl	4.85	v. atom, integer or molecule
uref,urefl	15.88	undef v. molecule
uatom,uatoml, uint,uintl	8.68	v. atom or integer
uskel	12.22	v. molecule
uskell	26.49 + 28.26 per argument	v. molecule, with mol. v. ref. for each arg.
uskeld	11.20	v.molecule
uskelc	15.60	v. reference
init,localinit	.95	per cell initialised
ifdone	1.45	
call	1.34	
try	1.34	
trylast	3.75	
enter	9.67	
neck	12.77	general case
foot	7.55	
neckfoot	2.74	no globals, determinate exit
cut	14.32 + 9.24 per trail entry examined	general case, assumes each trail entry retained
neckcut	15.53 + 9.24 per trail entry examined	no globals
neckcutfoot	12.10 + 9.24 per trail entry examined	no globals
fail	13.14 + 5.85 per trail entry examined	
"shallow" fail	6.08 + 6.66 per trail entry examined	

5.0 BENCHMARK TESTS

Times in milliseconds

Procedure	Data	Prolog-10	Lisp	Pop-2	Prolog-M	Prolog-10I
nreverse	list30	53.7	34.6	203	1156	1160
qsort	list50	75.0	43.8	134	1272	1344
deriv	times10	3.00	5.21	11.2	86.4	76.2
	divide10	2.94	7.71	15.9	90.6	84.4
	log10	1.92	2.19	8.56	61.6	49.2
	ops8	2.24	2.94	5.25	61.2	63.7
serialise	palin25	40.2	19.76	-	711	602
dbquery	-	185	-	300	9970	8888

Time ratios

Procedure	Data	Prolog-10	Lisp	Pop-2	Prolog-M	Prolog-10I
nreverse	list30	1	.64	3.8	22	22
qsort	list50	1	.58	1.8	17	18
deriv	times10	1	1.7	3.7	29	25
	divide10	1	2.6	5.4	31	29
	log10	1	1.1	4.5	32	26
	ops8	1	1.3	2.3	27	28
serialise	palin25	1	.49	-	18	15
dbquery	-	1	-	1.6	54	48

Notes

The above table, giving average figures for actual CPU time on a DECsystem-10 (KI processor), compares compiled Prolog (our implementation, "Prolog-10"), compiled Lisp (Stanford with the NOUUO

option), compiled Pop-2 and interpreted Prolog (both the Marseille Fortran implementation, "Prolog-M", and our implementation in Prolog, "Prolog-10I"). The data was obtained by timing (via "control-T") a large number of iterations of each test. The figures include garbage collection times for Lisp and Pop-2. No garbage collection was needed for Prolog since the stack mechanism recovers storage after each iteration. Test iterations were achieved in the following ways:-

Prolog

```
tests(N) :- read(_),from(1,N,I),test,fail.
tests(N) :- read(_),test.

from(I,I,I):-!.
from(L,N,I) :- N1 is (L+N)/2, from(L,N1,I).
from(L,N,I) :- L1 is (L+N)/2+1, from(L1,N,I).
```

Lisp

```
(DEFPROP TESTS (LAMBDA (N)
   (PROG (I RESULT)
      (READ)
      (SETQ I 0)
LAB   (SETQ RESULT TEST)
      (COND (LESSP I N) (GO LAB))
      (READ)
      (RETURN RESULT)))
EXPR)
```

Pop-2

```
FUNCTION TESTS N;
   VARS I RESULT;
   ERASE(ITEMREAD());
   FORALL I 1 1 N;
      TEST -> RESULT
   CLOSE;
   ERASE(ITEMREAD());
   RESULT
END
```

The dummy "reads" serve to interrupt the execution of each test so that "control-T" timings can be taken. The Prolog form of each benchmark test is listed below, together with the Lisp and Pop-2 versions selected for comparison. Note that in the Prolog examples a

more convenient syntactic form is used for lists. Thus '[]' stands for the empty list and '[X,..L]' denotes a list whose head is X and tail is L. A list of two elements 'a' followed by 'b' is written '[a,b]'. Apart from the syntax, such lists are treated no differently from other terms. The timing data would be exactly the same if, say, 'nil' and 'cons(X,L)' were used.

5.1 reverse

```
list30 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                                21,22,23,24,25,26,27,28,29,30]
```

Prolog : nreverse(list30,X)

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

nreverse([X,..L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

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

Lisp : (NREVERSE list30)

```
(DEFPROP NREVERSE (LAMBDA (L)
   (COND ((NULL L) NIL)
      (T (CONCATENATE (NREVERSE (CDR L)) (CONS (CAR L) NIL)))))
EXPR)

(DEFPROP CONCATENATE (LAMBDA (L1 L2)
   (COND ((NULL L1) L2)
      (T (CONS (CAR L1) (CONCATENATE (CDR L1 L2))))))
EXPR)
```

Pop-2 : NREVERSE(list30)

```
FUNCTION NREVERSE LIST;
   IF NULL(LIST) THEN NIL
   ELSE CONCATENATE(NREVERSE(TL(LIST)),HD(LIST)::NIL)
   CLOSE
END;

FUNCTION CONCATENATE LIST1 LIST2;
   IF NULL(LIST1) THEN LIST2
   ELSE HD(LIST1)::CONCATENATE(TL(LIST1),LIST2)
   CLOSE
END;
```

5.2 qsort

```
list50 = [27,74,17,33,94,18,46,83,65, 2,
          32,53,28,85,99,47,28,82, 6,11,
          55,29,39,81,90,37,10, 0,66,51,
           7,21,85,27,31,63,75, 4,95,99,
          11,28,61,74,18,92,40,53,59, 8]
```

Prolog : qsort(list50,X,[])

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

qsort([X,..L],R,R0) :-
   partition(L,X,L1,L2),
   qsort(L2,R1,R0),
   qsort(L1,R,[X,..R1]).
qsort([],R,R).

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

Lisp : (QSORT list50 NIL)

```
(DEFPROP QSORT (LAMBDA (L R)
   (COND ((NULL L) R)
      (T ((LAMBDA (P)
           (QSORT (CAR P) (CONS (CAR L) (QSORT (CDR P) R))))
         (PARTITION (CDR L) (CAR L))))))
EXPR)

(DEFPROP PARTITION (LAMBDA (L X)
   (COND ((NULL L) (CONS NIL NIL))
      (T ((LAMBDA (P)
         (COND ((LESSP (CAR L) X)
               (CONS (CONS (CAR L) (CAR P)) (CDR P)))
            (T (CONS (CAR P) (CONS (CAR L) (CDR P))))))
         (PARTITION (CDR L) X)))))
EXPR)
```

Pop-2 : QSORT(list50)

```
FUNCTION QSORT LIST;
VARS Y Z Q QQV QQW QQS;
   0;
L2:IF NULL(LIST) OR NULL(TL(LIST)) THEN GOTO SPLIT CLOSE;
   NIL->QQS; NIL->Y; NIL->Z;
   HD(LIST)->QQW;
L1:HD(LIST)->QQV; TL(LIST)->LIST;
   IF QQW>QQV THEN QQV::QQS->QQS
   ELSEIF QQW<QQV THEN QQV::Z->Z
   ELSE QQV::Y->Y
   CLOSE;
   IF NULL(LIST) THEN Z;Y;1; QQS->LIST; GOTO L2 ELSE GOTO L1 CLOSE;
SPLIT: ->Q; IF Q=0 THEN LIST EXIT
   ->Y;
   IF Q=1 THEN ->Z; LIST<>Y;2; Z->LIST; GOTO L2 CLOSE;
   Y<>LIST->LIST;
   GOTO SPLIT
END;
```

5.3 deriv

```
times10 = (((((((x*x)*x)*x)*x)*x)*x)*x)*x
divide10 = (((((((x/x)/x)/x)/x)/x)/x)/x)/x
log10 = log(log(log(log(log(log(log(log(log(log(x)))))))))
ops8 = (x+1)*(x^2+2)*(x^3+3)
```

Prolog : d(expr,x,Y)

```
:-mode d(+,+,-).
:-op(300,xfy,^).

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

Lisp : (DERIV expr (QUOTE X))

```
(DEFPROP DERIV (LAMBDA (E X)
   (COND ((ATOM E) (COND ((EQ E X) 1) (T 0)))
      ((OR (EQ (CAR E) (QUOTE PLUS)) (EQ (CAR E) (QUOTE DIFFERENCE)))
         (LIST (CAR E) (DERIV (CADR E) X) (DERIV (CADDR E) X)))
      ((EQ (CAR E) (QUOTE TIMES))
         (LIST (QUOTE PLUS)
            (LIST (CAR E) (CADDR E) (DERIV (CADR E) X))
            (LIST (CAR E) (CADR E) (DERIV (CADDR E) X))))
      ((EQ (CAR E) (QUOTE QUOTIENT))
         (LIST (CAR E)
            (LIST (QUOTE DIFFERENCE)
               (LIST (QUOTE TIMES) (CADDR E) (DERIV (CADR E) X))
               (LIST (QUOTE TIMES) (CADR E) (DERIV (CADDR E) X)))
            (LIST (QUOTE TIMES) (CADDR E) (CADDR E))))
      ((AND (EQ (CAR E) (QUOTE EXPT)) (NUMBERP (CADDR E)))
         (LIST (QUOTE TIMES)
            (LIST (QUOTE TIMES) (CADDR E)
               (LIST (CAR E) (CADR E) (SUB1 (CADDR E))))
            (DERIV (CADR E) X)))
      ((EQ (CAR E) (QUOTE MINUS))
         (LIST (CAR E) (DERIV (CADR E) X)))
      ((EQ (CAR E) (QUOTE EXP))
         (LIST (QUOTE TIMES) E (DERIV (CADR E) X)))
      ((EQ (CAR E) (QUOTE LOG))
         (LIST (QUOTE QUOTIENT) (DERIV (CADR E) X) (CADR E)))
      (T NIL)))
EXPR)
```

Pop-2 : DERIV(expr,X)

```
VARS SUM1 SUM2 DESTSUM OPERATION 4 ++;
RECORDFNS("SUM",[0 0])->SUM1->SUM2->DESTSUM->NONOP ++;
VARS DIFC1 DIFC2 DESTDIFC OPERATION 4 --;
RECORDFNS("DIFC",[0 0])->DIFC1->DIFC2->DESTDIFC->NONOP --;
VARS PROD1 PROD2 DESTPROD OPERATION 3 **;
RECORDFNS("PROD",[0 0])->PROD1->PROD2->DESTPROD->NONOP **;
VARS QUOT1 QUOT2 DESTQUOT OPERATION 3 ///;
RECORDFNS("QUOT",[0 0])->QUOT1->QUOT2->DESTQUOT->NONOP ///;
VARS POWR1 POWR2 DESTPOWR OPERATION 2 ~~;
RECORDFNS("POWR",[0 0])->POWR1->POWR2->DESTPOWR->NONOP ~~;
VARS MINUS1 DESTMINUS MINUS;
RECORDFNS("MINUS",[0])->MINUS1->DESTMINUS->MINUS;
VARS EXPF1 DESTEXPF EXPF;
RECORDFNS("EXPF",[0])->EXPF1->DESTEXPF->EXPF;
VARS LOGF1 DESTLOGF LOGF;
RECORDFNS("LOGF",[0])->LOGF1->DESTLOGF->LOGF;

FUNCTION DERIV E X;
   IF E.ISNUMBER  THEN 0
   ELSEIF E.ISWORD THEN IF E=X THEN 1 ELSE 0 CLOSE
   ELSEIF E.DATAWORD="SUM" THEN DERIV(SUM1(E),X)++DERIV(SUM2(E),X)
   ELSEIF E.DATAWORD="DIFC" THEN DERIV(DIFC1(E),X)--DERIV(DIFC2(E),X)
   ELSEIF E.DATAWORD="PROD" THEN
      DERIV(PROD1(E),X)**PROD2(E)++PROD1(E)**DERIV(PROD2(E),X)
   ELSEIF E.DATAWORD="QUOT" THEN
      (DERIV(QUOT1(E),X)**QUOT2(E)--QUOT1(E)**DERIV(QUOT2(E),X))
         ///QUOT2(E)~~2
   ELSEIF E.DATAWORD="POWR" AND POWR2(E).ISNUMBER THEN
      DERIV(POWR1(E),X)**POWR2(E)**POWR1(E)~~(POWR2(E)-1)
   ELSEIF E.DATAWORD="MINUS" THEN MINUS(DERIV(MINUS1(E),X))
   ELSEIF E.DATAWORD="EXPF" THEN E**DERIV(EXPF1(E),X)
   ELSEIF E.DATAWORD="LOGF" THEN DERIV(LOGF1(E),X)///LOGF1(E)
   ELSE "ERROR"
   CLOSE
END;
```

5.4 serialise

palin25 = "ABLE WAS I ERE I SAW ELBA"

ie. a list of 25 numbers representing the character codes.

Result = [2,3,6,4,1,9,2,8,1,5,1,4,7,4,1,5,1,8,2,9,1,4,6,3,2]

Prolog : serialise(palin25,X)

```
:-mode serialise(+,-).
:-mode pairlists(+,-,-).
:-mode arrange(+,-).
:-mode split(+,+,-,-).
:-mode before(+,+).
:-mode numbered(+,+,-).

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

pairlists([X,..L],[Y,..R],[pair(X,Y),..A]) :- pairlists(L,R,A).
pairlists([],[],[]).

arrange([X,..L],tree(T1,X,T2)) :-
   split(L,X,L1,L2),
   arrange(L1,T1),
   arrange(L2,T2).
arrange([],void).

split([X,..L],X,L1,L2) :-!, split(L,X,L1,L2).
split([X,..L],Y,[X,..L1],L2) :- before(X,Y),!, split(L,Y,L1,L2).
split([X,..L],Y,L1,[X,..L2]) :- before(Y,X),!, split(L,Y,L1,L2).
split([],_,[],[]).

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

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

Lisp : (SERIALISE palin25)

```
(DEFPROP SERIALISE (LAMBDA (L)
   (PROG (R)
      (SETQ R (DUPLICATE L))
      (NUMBERTREE 1 (ARRANGE (CELLS R)))
      (RETURN R)))
EXPR)
```

```
(DEFPROP DUPLICATE (LAMBDA (L)
   (COND ((NULL L) NIL)
      (T (CONS (CAR L) (DUPLICATE (CDR L))))))
EXPR)

(DEFPROP CELLS (LAMBDA (L)
   (COND ((NULL L) NIL)
      (T (CONS L (CELLS (CDR L))))))
EXPR)

(DEFPROP ARRANGE (LAMBDA (L)
   (COND ((NULL L) NIL)
      (T (CONS (CONS (CAR L) (MIDDLEPART (CAAR L) (CDR L)))
               (CONS (ARRANGE (LOWERPART (CAAR L) (CDR L)))
                     (ARRANGE (UPPERPART (CAAR L) (CDR L))))))))
EXPR)

(DEFPROP MIDDLEPART (LAMBDA (X L)
   (COND ((NULL L) NIL)
      ((EQ (CAAR L) X) (CONS (CAR L) (MIDDLEPART X (CDR L))))
      (T (MIDDLEPART X (CDR L)))))
EXPR)

(DEFPROP LOWERPART (LAMBDA (X L)
   (COND ((NULL L) NIL)
      ((LESSP (CAAR L) X) (CONS (CAR L) (LOWERPART X (CDR L))))
      (T (LOWERPART X (CDR L)))))
EXPR)

(DEFPROP UPPERPART (LAMBDA (X L)
   (COND ((NULL L) NIL)
      ((GREATERP (CAAR L) X) (CONS (CAR L) (UPPERPART X (CDR L))))
      (T (UPPERPART X (CDR L)))))
EXPR)

(DEFPROP NUMBERTREE (LAMBDA (N TREE)
   (COND ((NULL TREE) N)
      (T (NUMBERTREE
            (NUMBERLIST
               (NUMBERTREE N
                  (CADR TREE))
               (CAR TREE))
            (CDDR TREE)))))
EXPR)

(DEFPROP NUMBERLIST (LAMBDA (N LO)
   (PROG (L)
      (SETQ L LO)
LOOP  (RPLACA (CAR L) N)
      (SETQ L (CDR L))
      (COND ((NOT (NULL L)) (GO LOOP)))
      (RETURN (ADD1 N))))
EXPR)
```

5.5 query

The solutions to a database query to find countries of similar population density are

```
[indonesia, 223, pakistan,   219]
[uk,        650, w_germany,  645]
[italy,     477, philippines,461]
[france,    246, china,      244]
[ethiopia,   77, mexico,      76]
```

Prolog : query([C1,D1,C2,D2])

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

density(C,D) :- pop(C,P), area(C,A), D is (P*100)/A.

/* populations in 100000s,     areas in 1000s of sq. miles. */

pop(china,       8250).        area(china,       3380).
pop(india,       5863).        area(india,       1139).
pop(ussr,        2521).        area(ussr,        8708).
pop(usa,         2119).        area(usa,         3609).
pop(indonesia,   1276).        area(indonesia,    570).
pop(japan,       1097).        area(japan,        148).
pop(brazil,      1042).        area(brazil,      3288).
pop(bangladesh,   750).        area(bangladesh,    55).
pop(pakistan,     682).        area(pakistan,     311).
pop(w_germany,    620).        area(w_germany,     96).
pop(nigeria,      613).        area(nigeria,      373).
pop(mexico,       581).        area(mexico,       764).
pop(uk,           559).        area(uk,            86).
pop(italy,        554).        area(italy,        116).
pop(france,       525).        area(france,       213).
pop(philippines,  415).        area(philippines,   90).
pop(thailand,     410).        area(thailand,     200).
pop(turkey,       383).        area(turkey,       296).
pop(egypt,        364).        area(egypt,        386).
pop(spain,        352).        area(spain,        190).
pop(poland,       337).        area(poland,       121).
pop(s_korea,      335).        area(s_korea,       37).
pop(iran,         320).        area(iran,         628).
pop(ethiopia,     272).        area(ethiopia,     350).
pop(argentina,    251).        area(argentina,   1080).
```

Pop-2 : QUERY(N)

[N is the number of times the test is to be iterated. The strips COUNTRY, POPULATION, AREA are initialised with the appropriate data.]

```
VARS COUNTRY POPULATION AREA;
INIT(25)->COUNTRY;
INIT(25)->POPULATION;
INIT(25)->AREA;

FUNCTION DENSITY I; SUBSCR(I,POPULATION)*100/SUBSCR(I,AREA) END;

FUNCTION QUERY N;
VARS I C1 C2 D1 D2;
   ERASE(ITEMREAD());
   N+1->N;
   FORALL I 1 1 N;
      IF I=N THEN ERASE(ITEMREAD()) CLOSE;
      FORALL C1 1 1 25;
         DENSITY(C1)->D1;
         FORALL C2 1 1 25;
            DENSITY(C2)->D2;
            IF D1>D2 AND 20*D1<21*D2 AND I=N
               THEN PR([% SUBSCR(C1,COUNTRY),D1,
                          SUBSCR(C2,COUNTRY),D2 %]);NL(1)
            CLOSE
         CLOSE
      CLOSE
   CLOSE
END;
```

6.0 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.

Darvas F, Futo I and Szeredi P [1976]
Some applications of theorem-proving based machine intelligence in QSAR (quantitative structure-activity research).
Procs. QSAR conf., Suhl, E Germany. 1976.

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

DEC [1974]
DECsystem-10 System Reference Manual (3rd. edition).
Digital Equipment Corporation, Maynard, Mass. Aug 1974.

Dijkstra E W [1976]
A Discipline of Programming.
Prentice-Hall. 1976.

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.

Hobbs J R [1977]
What the nature of natural language tells us about how to
make natural language like programming languages more natural.
Procs. ACM SIGART-SIGPLAN Symp. on
AI and Programming Langs.,pp.85-93. Aug 1977.

Kowalski R A [1974]
Logic for problem solving.
DCL Memo 75, Dept of AI, Edinburgh. Mar 74.

Kowalski R A [1977]
Algorithm = Logic + Control.
Dept. of Computing & Control,Imperial College,London. 1977.

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

Markusz Z [1977]
Designing variants of flats.
Procs. IFIP conf. 1977.

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

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

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

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

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

Roussel P [1975]
Prolog : 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.

Szeredi P [1977]
Prolog - a very high level language based on predicate logic.
Procs.2nd.Hungarian Conf.on Computer Science,Budapest. Jun 1977.

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, Pereira L M and Pereira F [1977]
Prolog - the language and its implementation compared with Lisp.
Procs. ACM SIGART-SIGPLAN Symp. on
AI and Programming Languages. Aug 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.