An Implementation of D. H. Warren's PROLOG Machine on a Vax 11/780

R. W. Coffin
J. W. Roach

TR 86-17

An Implementation of D. H. Warren's PROLOG Machine on a Vax 11/780

by

R. W. Coffin
J. W. Roach

Virginia Tech
Department of Computer Science

TR-86-17
August 1986

ABSTRACT

This report describes, in considerable detail, the implementation of a Prolog compiler on a VAX. This implementation is modelled after Warren's Abstract Machine (WAM) for Prolog. The report clarifies and gives examples of the most important concepts in Warren's dissertation. In this regard, this report serves as a tutorial for others who wish to implement a Prolog compiler.

VT Logic Programming Internal Report #1
November 1984

Contents

List of Figures

1 Introduction

Prolog is a non-procedural predicate logic programming language. It is a dynamic language in the sense that new data structures (lists) can be constructed and then asserted (added) to the list of rules that comprise the program, as the program executes. Similarly, rules may be deleted at run-time. This facility is particularly natural in the versions of Prolog developed at Virginia Polytechnic Institute and State University [Roach and Fowler, 1981]: there is no syntactic distinction between program and data since both are modelled after the LISP s-expression.

Prolog is most easily implemented using an interpreter, in part because of this run-time flexibility. Indeed, most implementations are interpretive, including several developed at Virginia Tech. Since Prolog interpreters tend to be time and space inefficient, however, there is motivation to develop a compiler. In 1977, David Warren published his doctoral dissertation describing in some detail a Prolog compiler that he developed for a DECsystem-10.

The Warren report [Warren, 1977] has served as the design document for our Prolog compiler, written in the C language, for a VAX 11/780. This report has several objectives:

1. to clarify the most important aspects of the Warren report,
2. to explain how the Tech compiler differs and why, and
3. to serve as a maintenance manual for the Tech compiler.

This report assumes a knowledge of Prolog, especially from a procedural (as opposed to declarative) point of view.

1.1 Background

Warren's 1977 Prolog compiler implementation marks a watershed in making Prolog a viable language. He illuminated many of the problems involved and contributed greatly toward their solution. We followed his ideas closely in developing our compiler.

The dissertation offered by Warren is an extremely concise but sufficiently complete description of his implementation. We found several concepts and implementation details that needed altering, in part because our host machine, the VAX 11/780, has a very different architecture than Warren's DEC10 machine. The system Warren describes in the report represents a minimal Prolog. There were many gaps that had to be filled. For example, the very fundamental NOT operator is not treated. Also, there is no provision for failing a goal for which the user has defined no rules. Nor does he describe failure of a goal for which rules may have a different number of arguments.

We had to flesh out the details not described in Warren because we wanted to build a system suitable both for general student use and for supporting research in various areas of artificial intelligence.

For one thing, a host of student programs developed for use with the interpreters had to run on the compiler. We wished to provide a facility for adding and deleting rules to the database as the program ran, a common feature in the student programs. Additionally, we required that the compiler cooperate closely with an interpretive LISP system.

1.2 Warren's Contribution

There are three major deficiencies with Prolog interpreters. 1) The fundamental unification operation is general purpose and therefore slow. In fact, 90% of execution time is spent in the unification process. 2) There are storage management problems. This was particularly severe with the earliest interpreters: they tended to use the entire memory space available. 3) The management of the Prolog search, which requires a stack, is not fully optimized.

Building a compiler requires, above all, a solution to the unification and space utilization problems. It was precisely in these areas that Warren had the most to contribute. His use of "mode declarations" and his classification scheme for Prolog variables, described next, will give an idea of how he solved these problems.

The term "mode" is new, but the idea is a familiar one to Prolog programmers. In designing a Prolog procedure, the programmer usually decides which arguments are to receive data and which are to return answers. For example, we append two given lists to find a third. The first two variables of such an append rule could be considered "input variables" (they receive the initial lists), and the third (it constructs and returns an answer) could be considered an "output variable". This distinction is precisely what identifies the mode of a variable. The fact that the rule can be run "backwards" to answer the question "what two lists appended together produce a given list?", is an interesting, and often valuable feature that Prolog offers by default. However, if the user can communicate to Prolog his guarantee that he will always use his procedure in a consistent way, whether "backwards", "forwards", or otherwise, ("my append will always receive two known lists to produce an answer"), then the compiler can produce code that executes more rapidly and is more compact.

Another idea introduced by Warren is transparent to the user. Warren distinguishes between "local" and "global" variables. Of course, all variables in a Prolog rule (or clause) are local to the clause in which they are defined. Warren's idea, however, is unrelated to the scope of a variable in the conventional sense. A global variable is simply one that for at least one of its appearances in the clause is nested within parentheses or brackets. (Precise definitions will be given later.) For

example, the variables of dotted pairs are global. (Dotted pairs are commonly represented as $(?x.?y)$, $[?x|?y]$, or $cons(?x, ?y)$, depending on the Prolog being used.) Broadly speaking, other variables are local. Warren observes that the values to which global variables have instantiated ("unified") must be remembered on the stack[1] From this observation stems a very fruitful idea: the use of separate local and global stacks[1] to hold variable unifications. In this way, it is possible to recover local stack space well before it would be possible with a single stack.

1.3 An Overview of the Compiler

There are two main parts to the operation of the compiler. First, there is a clause scanning routine that handles such tasks as the recognition of local and global variables, the incorporation of mode declaration information, and the consolidation of procedures (those clauses with the same "name").

The second step deals with the generation of an array (the "code array") of executable instructions and data ("literals") that encodes the Prolog source into an equivalent but more useful representation. The instructions correspond to the intermediate language produced by many compilers, though for our implementation, this constitutes the final form. In general, each instruction corresponds to a single source symbol. In fact, the sequencing of the instructions corresponds closely to a left to right reading of the original source clauses.

In most cases, the complexity of an operation performed by an instruction is constant. For this reason, as Warren notes [page 51, Volume 1], it is natural to think of these instructions as the primitive machine instructions for a Prolog Machine or PLM. The function of the run-time module, then, is to act as the central control for the execution (interpretation) of these PLM instructions, or macros. The latter term emphasizes the fact that they could be replaced by in-line native machine code. (In practice, some routine calls would be necessary for the more complex unification routines.)

The function of the PLM macros corresponds to the two main features of any Prolog implementation: unification and execution flow control (the Prolog search). This report similarly divides into an explanation of these two aspects of the Prolog Machine. Greater emphasis is given to the conceptually more challenging of the two problems: the management of the flow of control.

Since one of the purposes of paper is to clarify Warren's presentation, there are many references to his 1977 report. Assume that page numbers in the references to his report refer to Volume 1.

[1]The stack is the data structure used to remember the current unifications of variables and to manage Prolog's built-in search mechanism. longer than the values for local variables.

1.4 The VAX 11/780 and the DECsystem 10

As mentioned above, our compiler was designed for DEC's VAX 11/780 minicomputer, whereas Warren used the DECsystem 10. Parts of the Warren report deal with how he used to his advantage the rather unique architecture of his machine. It is a quite different machine than the VAX, offering a 36-bit word and 18-bit addresses. The form of much of the literal data and stack data used in the PLM appears as $< pointer, pointer >$ or $< type, pointer >$. Warren took advantage of the DEC10 word size to hold a pair of addresses. Also, the "address word" feature of the DEC10, where a word is used to store an index register address and an offset, is used to advantage in Warren's implementation, especially during dereferencing (finding current variable bindings). In this way, the appropriate value can be found in one instruction, which speeds his unification routines considerably.

Clearly, the DEC10 architecture is well suited to the operation of the PLM. The VAX offers a 32-bit word (a "longword"), and uses the entire longword for addresses. Thus, a similar scheme for storing data will use a great deal more space: a pair of longwords versus a single DEC10 word. Also, VAX offers no mechanism comparable to the address word on a DEC10.

Our compiler represents a first attempt at implementing Warren's approach to Prolog compiler construction. Many speed and space improvements are possible (see Appendix 1). Because of the hardware implications mentioned above, however, we should not expect to improve on the performance of Warren's compiler.

2 Syntax, Terminology, and Basic Concepts

An attempt has been made to retain the basic terminology of the Warren report. However, we use a different syntax, and alter the "functor" concept slightly (explained below).

The format of a Prolog clause can be informally described by the following BNF grammar. Nonterminals are in upper-case, terminals in lower-case; ellipsis denotes "zero or more of the preceding item" and vertical bar denotes "or":

```
1)          PROGRAM := (assert MODE-DECLARATION ...  CLAUSE ...)

2)  MODE-DECLARATION := (mode (HEAD-PREDICATE MODE ...))

3)            CLAUSE := (HEAD if BODY) | (HEAD)

4)              HEAD := (HEAD-PREDICATE TERM ...)
```

```
5)              BODY := (GOAL-PREDICATE TERM ...) ...

6)              TERM := ATOM | (TERM ...) | (TERM . TERM)

7)              ATOM := CONSTANT | VARIABLE

8)   HEAD-PREDICATE := CONSTANT

9)   GOAL-PREDICATE := CONSTANT

10)             MODE := + | - | ?
```

Notes:

1. Variables have a '?' as their first or only character. CONSTANTS may be numbers, or strings of characters not starting with '?', or quoted strings whose first quoted character is not '?'.

2. The form "(HEAD)" in production 3 is called a fact or unit clause.

3. The term head will be informally used to denote either HEAD or HEAD-PREDICATE. Similarly, the term goal will denote either GOAL-PREDICATE or the entire term (GOAL-PREDICATE TERM ...). The context will make clear the intended meaning. Note that head and goals cannot be numeric.

4. The form "()" (see production 6) is alternatively spelled "nil".

5. If a term has the form ATOM (production 6), it is referred to as elementary. Otherwise, we speak of a compound term. The head and goals of a clause are sometimes referred to as boolean terms.

6. Currently production 9 is correct. The interpreters at Tech and eventually this compiler will allow some form of variable here.

7. The "if" (see production 3) in a clause is sometimes referred to as the neck, and the end of a clause is often called its foot.

The set of all clauses with the same head predicate is a procedure. We use the head predicate name as the name for the procedure.

The level of a term refers to its depth of nesting. The level of "assert" is one, that of head and goal predicates is three. A skeleton term is a compound term at level three or deeper.

See Figure 1, page 6, for a labeled example of a complete program.

Example:

```
1)      (assert
2)      (mode (append + + -))

3)      ((GO) if (append (a b c) (1 2) ?x))

4)      ((append (?x.?y) ?z (?x.?w)) if (append ?y ?z ?w))
5)      ((append nil ?x ?x))

        )
```

- Lines 4 and 5 define the procedure "append", line 3 defines the procedure "GO".
- Line 3 contains five constants: $a, b, c, 1, 2$ and one variable: $?x$.
- The terms "$(a\ b\ c)$" of line 3 and $(?x.?y)$ of line 4 are examples of skeleton terms. Note that they are both at level 3, whereas their arguments are at level 4.
- "$(append(?x.?y)\ ?z\ (?x.?w))$" is the head of line 4, and "GO" is the head of line 3.
- "$(append\ ?y\ ?z\ ?w)$ is the body of line 4 and constitutes its only goal.

Figure 1: Prolog Terminology

2.1 Functors

Warren's use of the term "functor" is more difficult to adapt to our syntax. We use the term in a more restricted sense. Any compound term that may serve as a goal (or head) is of the form (PREDICATE TERM...)...In this context, we call PREDICATE a functor and each TERM an argument. The arguments are numbered from 0 which specifies the argument's position. The number of arguments is the arity of the corresponding functor. A compound term in any other context (Warren would say) implicitly has the functor "cons". Its arguments are again numbered from 0 but from the first item of the list, rather than the second. Note that as we define "functor", it becomes synonymous with "predicate". See Figure 2, page 8, for an example of the numbering of arguments in a clause.

In the Warren report, the functor is expressed explicitly in all cases. Thus, rather than allowing the form $(a\ b)$, Warren uses $cons(a, cons(b, nil))$. For the Warren compiler there is a semantic difference between an arbitrary list of data (constructed with a series of "cons"-es) and a non-cons functor plus its arguments. The latter never unifies with the former since the functors do not match. Thus, a non-cons functor plus arguments cannot participate in the construction or the breaking down of a list of data.

We find that there is a gain in generality by treating all lists uniformly. In our system, potentially any compound term can be constructed or broken down (when variable goals are implemented, in particular), so we do not make the distinction. In essence, any list containing a fixed number of terms could potentially benefit by this distinction, since it could be stored in a fixed-length array. It would be treated quite differently during compile-time and run-time (in fact, more efficiently in the latter case). However, a design decision was made to store all the compound terms using the LISP technique of linked cons cells. This enables any compound term to unify against any other, and the length of any list becomes flexible.

2.2 Execution Terminology

During the operation of the "Prolog Machine" (to be described later), we conceive of the current action as an attempt by a goal to match with the head of a clause. Suppose the following code is executing, with goal A attempting to match with the clause in line 2.

```
(( ... ) if (A ...) (B ...))    ; 1
((A ...) if (C ...) (D ...))    ; 2
((A ...) if (E ...))            ; 3
```

Example:

```
((X ( (?x.?y). a) ?z (?x.?w)) if (or ( Y ?y ?z ?w) ( Z ?x ?y) ))
   A B C  D   E   F G H  I          J    K  L  M  N    O  P
```

Explanation:
The letters under the clause are naming the items for the discussion. Item E is "a", item A is "$((?x.?y).a)$", etc.
There are four predicates in the clause: X, or, Y, Z. (We know that Y and Z are predicates since we recognize that *or*, a builtin function, takes terms of this form as arguments.) Predicate arguments are numbered from zero. Thus items A, F and G are the arguments of predicate X, numbered 0, 1, and 2. Similarly, goal *or* has arguments J and N. Predicate Y has arguments K, L, and M.
We often speak of the arity of predicates (the number of arguments). The arities of predicates X, *or*, Y, and Z are 3, 2, 3, and 2, respectively.
The arguments of all other terms are numbered from their first term. For example, skeleton term A has two arguments, B and E numbered 0, and 1. Skeleton term G has arguments H and I numbered 0, 1.

Figure 2: Argument Numbering and Arities of Predicates

We call line 1 the parent clause. Goal A is the parent or current goal, and line 2 is the current clause. At times it is useful to consider the grandparent goal (or clause). This is the goal that activated the parent clause. Goal B is the parent goal's continuation. Line 3 is an alternative clause for the current goal. The (latest) choice point refers to the clause that would be activated (tried) on failure of the current clause. In this case, line 3 would be the latest choice point, and we say that shallow backtracking occurs when the current clause fails. If line 3 were tried and failed, some previous clause would be the choice point and would be the next one activated. This type of control transfer is termed deep backtracking. For the clause that is activated on deep backtracking (that is, when the current procedure in its entirety has failed), we introduce the term deep choice point. Thus, if the last clause of a procedure is currently active, (and therefore there are no remaining alternatives–no shallow backtracking point) its latest choice point is the same as its deep choice point. Otherwise, they are different.

2.3 Procedure Determinacy

One of the advantages of compilation is the possibility of recognizing, in advance, when there will be no alternative clauses during the execution of a goal ("determinate" execution). Only when this occurs does the concept of the Prolog procedure closely resemble that of conventional procedural languages such as Pascal: on procedure exit, stack space can be recovered.

Warren defines determinate execution of a goal or procedure as follows [page 24]:

> 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).
>
> For example, consider the following Prolog fragment:

```
(( ... ) if (A ...))                    ; 1
((A ...) if (B ...) (C ...))            ; 2
((B ...))                               ; 3
((B ...))                               ; 4
((C ...))                               ; 5
```

Suppose that goal B (line 2) has successfully matched with the fact on line 3, C has finished matching with its fact on line 5, and control has returned to line 2. We see that C has executed determinately, whereas B has not. Notice in particular

that goal A (line 1) has completed indeterminately since one of the goals invoked during the execution of A has exited indeterminately (namely, B).

Notice how the Prolog cut operator affects the determinacy of procedures. Once the cut is executed, any goal between the parent goal and the cut operator acts as though there were no alternatives (and this includes the parent goal itself). In this sense, as Warren notes, the cut operation "restores conventional determinacy to a procedure" [Warren, page 27].

2.4 Data Representation

We can think of a "term" as an object in the source code. The clause containing the term can be activated many times during the execution of the program, each time with a distinct set of variable values. By substituting for each occurrence of a variable in the term the value to which it has unified, we obtain an instance of that term, and refer to it as a constructed term. The original term is called the source term.

Note that this idea is applicable to an entire clause, and we may speak of a clause instance. One way to think of a clause instance is as the union of the instances of all its terms for a given clause activation.

A basic problem for any implementation is how to represent term instances (constructed terms). The concept of "structure sharing" is the approach used by the compiler. Using structure sharing, a compound constructed term is represented by a pair $< source-term,\ frame >$ where "frame" points to an array of value cells or variable cells that hold the values of the variables contained in the source term, and "source-term" is the address of a direct representation of the source term (called a "skeleton literal"–see 2.5 on Literal Data, below). The position within the array of the value cell for a particular variable is determined by a number, called the variable number, assigned to the variable when the source clause is first scanned. (Figure 3, page 13, gives an example of variable numbering.)

There are four classes of data items, called constructs, that can occur as values of variable cells (cell values). Constructs are structures formed from a pair of long-words (32 bits each), that we may represent as $< type,\ pointer >$. The significance of these fields varies slightly with the construct, but the type field can always be used to determine the type of construct at hand. Generally, it is a simply small integer that encodes the type. The four types (though the constant type is really two types) are as follows:

1. The reference construct is used to indicate that one variable is bound to another. It is of the form $< REF,\ cell\_ptr >$ where cell_ptr contains the address of the value cell for the other variable.

2. To indicate that a variable is still unbound, we use the empty construct, $< UNDEF, UNDEF >$.

3. There are two constant constructs that indicate a variable is bound to an atom or number. They are represented as $< ATOM, atom\_ptr >$ and $< NUMBER, number\_ptr >$.

4. If a variable binds to an instance of a skeleton term, we use a construct of the form $< skel\_ptr, frame\_ptr >$; it represents a constructed term in the structure sharing manner as described above. This construct is termed a molecule. We can say that a molecule also has the $< type, pointer >$ form, since we can use its frame_ptr field to distinguish it. We simply count on the fact that all addresses will be larger than the largest integer used to encode the types of the other four constructs.

See Figure 4, page 14, for a summary of Prolog data types and formats.

2.5 Literal Data

All other data used by the Prolog Machine represent either the arguments of a skeleton term, called inner literals, or the arguments of a goal, called outer literals. We use the term skeleton literal to refer to the vector of inner literals that is associated with a skeleton. Warren speaks also of "functor literals", which we ignore (see Section 2.1 on Functors above). Figure 5, page 15, gives examples.

A literal data item has the same $< type, pointer >$ form as a construct. See Figure 4, page 14, for the various possibilities.

The outer literals are stored consecutively directly in the PLM code area. Skeleton literals may be stored after the code for the clause that contains the source skeleton terms (as Warren does). In our implementation, however, code may be deleted or overlaid before references to the skeleton literals would be discarded. Thus, we use a separate area for all skeleton literal data. Warren stores literal data in a way that takes advantage of the address word concept of the DEC10 architecture, which we can ignore. For the details of skeleton literal storage in our system, see Figure 5. To gain an understanding of how skeleton literals plus constructs representing the variable bindings work together to represent an instance of a term, see Appendix 2.

2.6 Variable Classification

An innovation introduced by Warren is the classification of variables according to how long their values need to be remembered on the stacks (the lifetime of the variable). Void variables are those with a single occurrence in the clause but not

in a skeleton. No space need be allocated to them on the stack, so they have no lifetime. Temporary variables occur only in the head and not in a skeleton. Their cell values on the stack can be discarded when a goal has successfully unified with the head of a clause. Local variables also do not occur in skeletons, but have at least two occurrences in the clause with at least one in the body. Local variables are discarded at determinate procedure exit. Finally, global variables are those with an occurrence in a skeleton. They must be remembered until failure induces backtracking. (See Figure 3, page 13, for examples and a summary. Section 4.3 covers the reason for the lifetimes of these classes of variables.)

2.7 Prolog Environments

The two major aspects to any Prolog implementation are unification and flow of control. The latter includes procedure calls (often recursive), success action, failure action (backtracking), and the cut operation. The Prolog Machine deals with all these control flow issues by using stacks to hold management and variable binding information, and by using a set of special longword data items called registers (a reference to their frequent use and global nature).

As the PLM operates, it attempts to establish a clause instance by constructing an environment on the stacks for the clause. As Warren observes (Warren, page 31), there is a close analogy between representing an instance of a source term and representing an instance of a clause. Structure sharing is used in the former case where the term instance is expressed with the pair $< source\_term, frame >$. Similarly, a clause instance can be represented by a pair $< clause, environment >$. This emphasizes the fact that there is a one-to-one correspondence between an environment and the activation of some clause. An environment is composed of a local stack frame, a (possibly empty) global stack frame, and a (possibly empty) set of trail entries.

The trail entries, held on a third stack (the trail), consist of pointers to value cells that were defined during the creation of the environment. This may include variable cells from earlier environments. The trail record must be kept since these are the cells that must be reset to "undef" on failure of the clause. However, it is unnecessary to include pointers to variable cells that will be discarded automatically on backtracking (for example, the cells of the current environment).

The global frame consists only of the variable cells that occur in skeletons.

The local frame consists of the cells for the local and temporary variables of the clause. It is also used to store the current values of the special registers. This requires six longwords called management area.

See Figure 6, page 16, for the layout of an environment.

Example:

Source (classification and variable numbers shown below source):

$$((A\ ?t\ (?u.?v)\ (?\ .?u)\ ?w\ ?t\ ?y)\ if\ (B\ (?w)\ ?y\ ?z\ ?\ \ ?u)\ (C\ ?x\ ?x))$$

$$((A\ T2\ (G0.G1)\ (G2.G0)\ G3\ T2\ L0)\ if\ (B\ (G3)\ L0\ VD\ VD\ VD\ G0)\ (C\ L1\ L1))$$

where:

$$(VD = void, T = temporary, L = local, G = global)$$

Explanation:

After classification is assigned based on the chart below, variable numbers are assigned left to right, with local and temporary varia- bles forming one sequence, globals forming an independent sequence. Both start with zero. Temporaries are numbered after all locals have been numbered. Void variables receive no number.

The "global anonymous" variables have been correctly classified. See Appendix 6 for an explanation.

Summary of classification:

The following chart is due to Warren [page 53].

Name	Lifetime ends	Criterion
Global	Backtracking.	Occurs in a skeleton.
Local	Procedure completed successfully and determinately, i.e., 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.

Figure 3: Variable Classification and Numbering

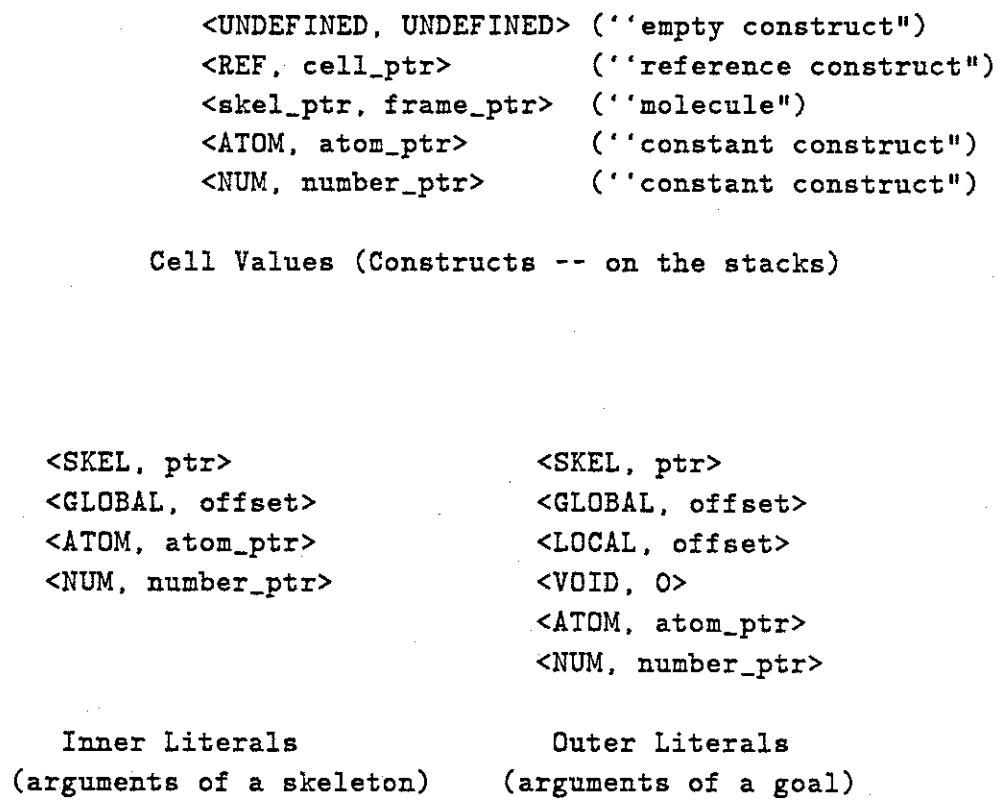


Figure 4: Prolog Machine Data Representation

Note that if the skeleton term in the following example were instead an argument in the head, the corresponding skeleton literal would appear the same, but the pointer to the skeleton literal (L1 in this case) would appear as an argument to a unification instruction, rather than as a field in an outer literal as shown below.

Source clause:

$$((green\ ?x)\ if\ (gold\ \ ?x\ (?x\ (?x\ a)\ .\ b)\ ?y\ 1\ a)\ )$$

In code array arguments of goal "gold" stored as outer literals:

argument number	outer literal	source term
0	<GLOBAL, 0>	$?x$
1	<SKEL, L1>	$(?x\ (?x\ a)\ .\ b)$
2	<VOID, 0>	$?y$
3	<NUMBER, "1">	1 (second field is pointer to 1)
4	<ATOM, "a">	a (second field is pointer to 1)

Source term for skeleton term argument:

$$(?x\ (?x\ a)\ .\ b)$$

Fully dotted equivalent:

$$(?x\ .\ ((?x\ .\ (a\ .\ nil))\ .\ b))$$

Skeleton literal:

Stored as a linked list in consecutive cells of the "literal array". Each inner literal requires two consecutive longwords (total of 8 bytes), so this example requires 64 consecutive bytes.

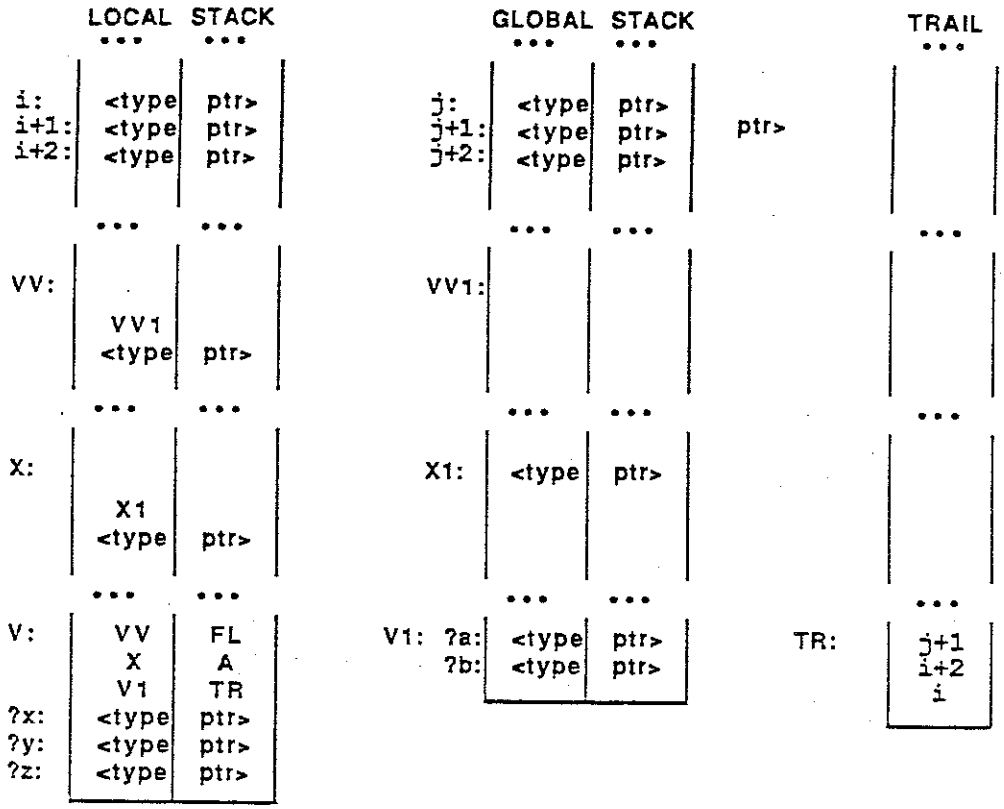
L1: <GLOBAL, 0>
<SKEL, L2>

L2: <SKEL, L3>
<ATOM, "b"> (pointer to atom cell "b")

L3: <GLOBAL, 0>
<SKEL, L4>

L4: <ATOM, "a">
<ATOM, "nil">

Figure 5: Example of Literal Storage



FL: (code-for-alternative-pointer, next-clause-cell-pointer) [cons cell]
(See Figures 7-11, pages 31 - 35, for examples of these "clause cells".)
A: <type, pointer> (some outer literal)

This figure illustrates the layout of a Prolog environment. It cannot be illustrated in full generality, since different situations imply a different appearance. The environment illustrated above shows a current clause with three local variables (*?x, ?y, and ?z*) and two global variables (*?a and ?b*). (They are labeled in the diagram for illustrative purposes. For example, *?y* is actually at address $V + 3 + 1$, where 1 is the variable number for *?y*.)
Also, note that VV can be above or below X, or VV can point to the same environment as X. (See the discussion, Section 3.1)
Note how the $V1$ field of the X and VV environments reflect the current values of $X1$ and $VV1$, respectively. Also note how the sample trail entries must all be from environments prior to VV or $VV1$.

Figure 6: Prolog Environment Layout

3 Managing Flow of Control

3.1 Special Registers

The special registers and their functions are covered in Warren on pages 38 through 40 and 46 through 48. Below, we summarize this information, and supplement it with additional insights that should prove helpful in understanding the operation of the Machine. Also, refer to Appendix 3. See Section 2.2 for the execution terminology used in this section.

In what follows, we often refer to the registers themselves and to the corresponding fields in the management area in which the register values are stored when an environment is created. We will always include the word "field" when referring to the values in the management area. Giving the register name is a reference to the current value in the register (not the field).

- V and $V1$ point to the top of the local and global stacks, respectively.
- The TR register points to the top of the trail.
- X and $X1$ point to the local and global frames of the environment of the parent clause, and thus, of the current (parent) goal.
- VV and $VV1$ point to the environment of the clause that would be activated on failure of the current clause. If there is an alternative clause for the current procedure, then $V = VV$ and $VV1 = V1$, since the current environment is the one that would be reconstructed as the head of the next clause of the procedure attempts to match with this same parent goal. Otherwise, $VV < V$, and some previous environment would become active– the latest one for which there <u>are</u> alternative clauses to try.
- FL, the failure label, indicates which clause to activate on failure of the current clause, <u>if</u> there are alternative clauses to try. Otherwise, the FL is undefined. As we saw in the preceding paragraph, we can recognize this case by comparing V and VV.
- A points into the code area to the vector of outer literals that represents the arguments of the current goal. The instruction for the next goal (or for the foot of the clause) follows these arguments and is the termed the <u>continuation</u> of the current goal. Thus the A register serves a dual purpose: it points to the incoming arguments to be unified against the current clause head, and it directs us to the next instruction to execute on success of this unification.

The management information area is initialized with the current values of these registers when the procedure is invoked (thus, a new environment is being constructed). Therefore, during the unification process, the A, X, and $V1$ fields hold the same value as the corresponding registers do. The $V1$ field is the address of the global frame that corresponds to this local frame, so it serves to connect the two parts of the environment. (In a similar way, the TR field serves to connect the third part of an environment to the rest.)

The FL field would also hold the same value as the corresponding register, but the register value is not actually stored in its field until the entire clause head has successfully matched with the parent. This is appropriate since if failure occurred during unification, the FL register would be altered to point to the clause after the next. We want the FL field to reflect the alternative clause to try when the current one has succeeded so that in the event this environment is reactivated by failure at a deeper level, we can still reach the alternative. Thus, by postponing the assignment of the FL register to its field, we save ourselves this step in the event of failure.

The TR register in general would differ from its field, since it is being altered during the unification process (in fact, TR is the only register possibly altered during unification). The TR field holds the location of the top of the trail before unification gets underway. On failure, the TR register indicates the new top of the trail thus enabling the correct number of altered cells to be reset to "undef".

The VV field is usually different from the VV register. When there are alternative clauses to match with the current goal, $V = VV$ as indicated above. The VV field, on the other hand, always indicates the deep choice point (the environment to activate on failure of the entire current procedure, i.e., when the last alternative in the current procedure has failed.) In short, the field and register are equal if and only if the last clause of the procedure is the current clause.

Note that in the event of deep or shallow failure, the environment that is activated contains the appropriate management information. That is, the register fields are not updated once the new environment has been established. This implies that the same set of management information is required (and the same environment is used) for every clause "attempt" in a particular procedure.

On the other hand, a new environment is established on every success. In fact, the only thing that the environments of two goals from the same clause have in common is the same parent (X and $X1$ fields).

A final observation. In the management area, the VV field can be numerically larger than, smaller than, or equal to the X field. By definition, $VV = X$ (fields) if and only if the deep choice point is the next clause in the parent's procedure. Similarly, if $VV < X$ (fields), then all computation performed since the grandparent goal was invoked is determinate (except that possibly there is an alternative to the current clause; the VV field gives the deep choice point and is not affected by the

multiplicity of the current procedure.) Otherwise ($VV > X$), some goal earlier in the parent clause than the current goal, can potentially match with an alternative clause.

It may be of interest to observe how to pick up the computation from an arbitrary point, based on a snapshot of the stacks (if trying to follow the computation by hand, for example). First, choose an environment pointer to some local frame, and place it in V. Restore the $V1$, X, A, and FL registers from the corresponding fields of the management area. (The FL field is invalid if the environment chosen was suspended during its construction.) Set $VV := V$ and $VV1 := V1$. $X1$ is in the $V1$ field of the environment pointed to by X. If you have chosen a deep choice point (that is, a stack address P such that $P = VV$ field of some environment), then you can trust the FL register and begin execution at the clause it indicates. This corresponds to picking up the execution directly after deep backtracking. It is also possible to start execution at the clause prior to the one indicated by FL (if you can identify it) to repeat the situation currently on the stack. If you have chosen an environment that does not corresponds to any deep choice point, the FL field will be invalid. In this case, begin execution at the goal indicated by $A - 1$ (the CALL instruction for this goal) to repeat the execution shown on the stack.

By now it should be fairly clear how the special registers operate to control backtracking, cut, and procedure success. (See pages 33 and 46 through 48 of Warren.)

3.2 The Cut Operator

The cut in Prolog commits the search to all the choices made since the parent goal was invoked. All clauses that were activated during the execution of the parent goal that had alternatives now act as though there were no alternatives, that is, they become determinate. Local stack space established for all the environments created for these intervening procedure calls can now be recovered. If the current clause fails, the cut demands that the entire procedure fails. Note that the correct latest choice point for the goals after the cut is now the deep choice point for the current clause. In addition to setting an earlier choice point, execution of the cut enables the discarding of all local environments created since the current clause was activated. Note that this is essentially the same case as determinate procedure exit, the other time that local stack space is recovered on success.

3.3 Backtracking

Backtracking is straightforward, given the choice point information readily available in the VV and $VV1$ registers. Shallow backtracking is recognized by the fact that $VV = V$. In this case, FL indicates the clause to activate next. (As was indicated

above, the information in the current management area is already valid.) Otherwise, the current clause is the last one of its procedure, and deep backtracking will take place on its failure. In this case, V and $V1$ are set from the VV and $VV1$ registers (effectively popping the stack). Note that the latter registers had been set back to the deep choice point on entry to this last clause in anticipation of failure (in the TRYLAST macro). The other registers are restored to the appropriate values from the management information in the backtrack environment, and the new FL register indicates the clause to activate next.

For either the deep fail or shallow fail cases, the trail is used to reset the cells to "undef" that were defined sometime during the computation between the activation of the clause that had an alternative and the activation of the clause that failed (actually, not all cells must be trailed; see Chapter 4 on unification). See DEEPFAIL in Appendix 3 for full details.

3.4 Procedure Success

Now let us consider procedure success. Note that the only purpose for the environments is to establish the success or failure of unification of certain goals with certain clause heads. (The binding of variables that occurs in the process is only an important side effect!) At determinate procedure exit, we know, first, that we have succeeded, and second, we know that if there are future failures, these determinate environments could never supply alternative successes. Thus, there is no reason to keep these environments, nor the variable bindings made in the process. (The reason that we cannot similarly discard global stack space on determinate procedure exit is that there may be references to it in molecules in the surviving portion of the the global or local stacks. Of course, the frame pointers of molecules point only to the global stack, so there is no corresponding problem for the local stack. See Section 4.3)

Therefore, procedure success, indicated by reaching the foot of a clause, first involves deciding if the procedure exit is determinate. If so, the local stack space created during its execution can be recovered. In this determinate case, the VV register will already have been set back to the deep choice point. If VV indicates a choice point earlier than the environment for this parent clause, the procedure is succeeding determinately, and we may pop the stack back to the parent environment, and establish a new parent. If the new parent goal's continuation is also the foot of its clause, then additional space may potentially be recovered in the same manner. Thus, procedure success is essentially a search back through the active clauses to find the latest one that has a goal for its continuation. And at each step, space is recovered from the local stack until a clause is found whose parent has an alternative. Once such a clause has been found, we cannot set V to the environment of the parent, but must leave it where it is (at the clause the parent

activated): recall that any environment that we set V to will be replaced with new management information, since this is a success exit. Clauses with an alternative must have their environments preserved, since they represent backtrack points. So suppose we found a clause whose parent had an alternative. The location of the new environment is now established, but the search continues for a new goal to execute.

4 Unification

Unification is basically straightforward to understand, but it involves a careful consideration of many cases, and is tricky to implement. First, some new terminology is needed.

Basic to the unification process is the dereferencing step [Warren, p. 44]. When we dereference a variable, we look up its current binding value. This is done by looking at the construct in the corresponding variable cell. Any reference constructs are followed until a non-reference is found. If this final construct is empty ("undef"), the result of dereferencing is defined to be $< REF, cell\_ptr >$ where cell_ptr holds the address of the cell with the empty construct. Thus, when we say that a variable "dereferences to a reference construct", we mean it is still unbound. In all other cases, the result is simply the final construct found. (The various possibilities for constructs are listed in Figure 4, page 14.)

A value cell on the global stack is considered more senior than one on the local stack. If two cells are on the same stack, the older one is considered more senior. Thus, since our stacks grow toward larger addresses, the cells with smaller addresses are more senior than those with larger addresses.

The following description of unification is summarized from Warren's description, with some added observations of our own. [Warren, page 45].

In order to unify two source terms, we unify the corresponding constructs. First, each must be dereferenced. If the result is two constants, they must, of course, be equal. If one of the two constructs is a reference, then the other construct is assigned to the cell pointed to by the reference. If the result is two molecules, each of the arguments of the corresponding skeleton literals must be unified. (Thus, we see that unification is essentially a recursive process, though we use iterative techniques with a stack in the compiler for the sake of speed.)

Finally, if both constructs are references, the more senior reference is assigned to the cell of the more junior reference. This prevents dangling references when the local stack is popped on determinate procedure exit. (On failure, cells are generally reset to "undef", so the problem does not occur here.)

Whenever a variable cell receives a construct (i.e., is bound) during the unification process, its address must be compared to the VV register (or $VV1$ for global

cells). If the cell would survive a failure (its address is less than VV or $VV1$), the assignment must be remembered on the trail, since on failure, Prolog requires that new unifications be undone. Note that in general, more values are trailed if there are alternatives for the current clause than when the current clause is the last of its procedure. In the first case, $VV = V$ and $VV1 = V1$, and all cells in environments other than the current one would be trailed. On the other hand, more value cells are set back to "undef" on deep failure, since that may include the unifications accomplished during the activation of several clauses, instead of just the current clause.

What the compiler really accomplishes in generating PLM unification instructions is an early recognition of certain common cases. An interpreter depends on a general unification routine, which, of course, must take into account all possibilities and therefore tends to be slow. The compiler, on the other hand, has performed at compile-time one level of case analysis and has encoded the results in the selection of the appropriate macro.

For example, if a $?x$ is encountered for the first time in a left-to-right scan of the head (known as the first occurrence of $?x$), then a UVAR macro is generated (let's assume the variable is found at level three). This macro says, in effect, unify successfully against any incoming term. There is no possibility of failure since $?x$ is as yet unbound. Now a subsequent occurrence (at level three) of this same variable will generate a UREF instruction. This says that $?x$ has bound to something, so it must first be dereferenced. Similar unification macros are generated for the level four arguments to skeletons.

Other cases that can be distinguished at compile time include UATOM (unify an atom against the incoming term), USKEL (unify a skeleton against the incoming term), USKELC (unify a skeleton against an incoming term that is guaranteed to dereference to a reference construct), and USKELD (unify a skeleton against an incoming term that is guaranteed to dereference to a non-reference construct). (See Appendix 4 for a psuedocoded description of the operation of all the unification macros.)

Since arguments deeper than level four are less often used, Warren's design does not allow for the generation of unification instructions for deeper arguments. See Section 4.2 on mode declarations and Section 5.1 on the PLM for a more complete discussion on this point.

4.1 Example of a Unification Instruction

To give a flavor for the kind of reasoning needed during unification, we describe here the macro USKEL. Psuedocoded descriptions of all the unification macros can be found in Appendix 4. Listings of a complete run are included in Appendix 5. Appendix 2 describes how to read a term instance from the stacks. This will

illuminate the representation of variable bindings created by the unification process.

For this description, let us call the skeleton in the head that caused the generation of this USKEL macro the "current skeleton".

The first task of all the unification instructions is to assign to a pair of longword variables the $< type, pointer >$ fields of the incoming argument (an outer literal). For the USKEL instruction and variants, we use $< B, Y >$ to hold this information. Warren, in fact, gives Y and B the status of PLM registers, since they must act globally to communicate their values to the skeleton argument instructions that follow in the macro stream.

At this point, then, $< B, Y >$ can contain any of the possibilities for outer literals (see Figure 4, page 14).

1. If Y indicates some constant type, we fail (a skeleton never unifies with an ATOM or NUMBER).

2. If Y indicates that the incoming argument is a skeleton term, we must unify each of its arguments against the corresponding arguments of the current skeleton. This is the task of the unification macros that follow in the instruction stream. These instructions use the registers B and Y as a molecule representing the constructed term that corresponds to the incoming skeleton. But, currently $< Y, B >=< SKEL, skel\_ptr >$. Thus, B correctly points to the skeleton literal of interest, so now Y is assigned the correct frame: $X1$ (the global frame of the parent clause).

3. If Y indicates a void variable, we are done (no explicit binding is necessary; we simply succeed). Setting Y to zero will cause the IFDONE instruction to skip around the current skeleton's argument instructions.

4. If Y indicates a LOCAL or GLOBAL variable, we dereference it, putting the $< type, pointer >$ result in $< Y, B >$. Now the possibilities for $< Y, B >$ are the various constructs, and the following three steps are used to distinguish among them. (see Figure 4).

   (a) If Y indicates some constant type, we fail (as before).

   (b) If Y indicates a reference construct, we must create a molecule, assign it to the cell pointed to in the reference construct, and trail the assignment if necessary. The correct molecule is $< V1, S >$ where S (an argument to the USKEL instruction) points to the skeleton literal for the current skeleton. In this case we are done, so we set Y to zero to skip the argument instructions. (See 3 above for a similar case.)

   (c) If Y indicates an incoming molecule, we want to unify each of the corresponding skeleton's arguments against those of the current skeleton (as in

case 2). In this case, B and Y already have the $< skel\_ptr, frame\_ptr >$ structure that the argument instructions expect, so we are done.

4.2 Mode Declarations

Another Warren innovation (which has a bearing on unification) is the use of mode declarations. If the user can guarantee that a procedure will always be called using predetermined argument positions for input data and certain others for output data, then the code generated can be considerably more time and space efficient. For example, in calling the append procedure (see Figure 1, page 6), we almost always wish to supply two lists for the first two arguments and expect an answer to be constructed for the third argument (the concatenation of the first two lists). The first two arguments would be the input, and the last would be the output. Note how the skeleton that matches against the first input argument serves to break down the incoming list. Similarly, the skeleton in the third argument position serves to construct a list. Warren terms the former as a "destructor" skeleton, the latter as a "constructor" skeleton.

The PLM has unification instructions for each case. The user, however, has to inform the compiler, with mode declarations, how he intends to use his procedures. If he violates his own declarations, the system prints a warning and fails the unification.

Currently, the only mode specification accepted is for each level three argument of the head predicate. When Warren hints that the idea could be extended [p. 59], he is no doubt referring to the possibility of mode declarations for arguments of a skeleton. But in order to take advantage of such an idea, the level for which unification instructions are generated would similarly have to be deepened. This could certainly be done, with a corresponding increase in complexity to the compiler. The question must be addressed whether the user would ever take advantage of this additional feature, a feature that, from his point of view, provides no functional difference, and may require some effort to use.

See Appendix 4 for the role mode declarations play during unification.

4.3 The Classification of Variables: Why Does it Work?

We attempt here to explain the reasoning behind the classification of variables. The reason for the classification is buried in a detailed look at how unification operates and how term instances are represented.

We use the phrase "binding value" to mean the construct (value) to which a variable has bound. We say that a variable is introduced, or originates in a clause C (or the corresponding environment) if the binding value its value cell receives during the activation of C is not the reference construct.

1. Void Variables

   First, it should be clear that void variables never need to appear on the stack. They cannot transmit their binding values to other variables, nor to the body of the clause, nor to other clauses. Thus, there is no reason to remember their binding values on the stack, nor even to discover what the value is. We may view them as mere placeholders whose only function is to provide the predicates of which they are arguments with the desired arities. If a void variable appears in the head of a clause, it always successfully unifies with the incoming argument. But all unification in the PLM is accomplished through unification instructions generated for the head of a clause. (See Chapter 5.) The only conceivable reason to include a "UVOID" (unify void) instruction would be, again, as a placeholder to assure proper alignment of unification instruction and incoming argument. But this is not necessary in the PLM since each unification instruction carries as an argument its argument position in the head. Each can, therefore, align itself by adding its argument position to the address of the first incoming argument. This address is stored in PLM register A.

   However, a void variable as an argument of a goal must correspond to an "incoming argument", that is, to some outer literal. We need an outer literal $< VOID, 0 >$. If the corresponding argument in the head is not another void variable, there will be some unification instruction present. It must always succeed when matching with $< VOID, 0 >$. But it needs something to match against, and this is really the only purpose for the void outer literal.

2. Temporary Variables

   This case is also quite obvious. By their definition, they occur only in the head of the clause, and occur in no skeleton literal. As with void variables, temporary variables cannot transmit their binding values to the body of the clause, nor, therefore, to subsequent clauses. Thus, once the neck is reached, there is no longer any need to remember the values to which they have bound. Their only purpose is to unify all the corresponding incoming arguments. This unification is reflected in new bindings created in earlier environments that will remain even when the value cells for the temporary variables have been popped off the stack.

3. Local Variables

   The distinction between local and global variables is rather subtle, and perhaps can best be illuminated with examples.

   Clearly, local variables serve to transmit binding information from head to body within a clause, and from clause to clause during procedure invocation.

Their binding values must in general remain on the stack as the goals in the body activate further clauses so that the new clauses may access these values if needed.

For the following example, consult Appendix 6. Suppose a goal G of a clause C contains a local variable $?x$ with value cell located at $?x - address$, and that this variable is introduced in C. Further, let us suppose that $?x$ is as yet undefined.

Consider what happens during unification. If $?x$ matches with $?y$, a local variable in the head of the clause invoked by G, then the cell value for $?y$ becomes $< REF, ?x - address >$. Later, $?y$ may unify with a constant A. In this case, the cell value for $?x$ becomes $< ATOM, A >$, and the cell value for $?y$ is not altered. Or, if $?y$ unifies with a molecule $< frame, skel >$, the cell value for $?x$ takes on this construct, and the cell value for $?y$ again is not altered.

Or, suppose $?y$, still unbound except to $?x$, is an argument in the call to another procedure where it unifies with $?z$, a local variable in the head of the new clause. The cell value for $?z$ becomes $< REF, ?x - address >$, the cell value for $?y$ is still not touched (it still contains $< REF, ?x - address >$), and the cell value for $?x$ remains at the empty construct.

Now if $?z$ unifies with an atom A, neither the cell value for $?y$ nor for $?z$ is altered. Only the cell value of $?x$ becomes $< ATOM, A >$. Or, if $?z$ unifies with a skeleton $< frame, skel >$, only the cell value of $?x$ is altered: it becomes $< frame, skel >$.

In short, when $?x$ becomes unified with a constant or molecule, whether directly (for example, $?x$ occurs in a goal and matches with a skeleton in a head) or indirectly as illustrated above, this binding is represented by a construct placed in the value cell for $?x$, that is, in the originating environment. If, in the meantime, other local variables from later environments become unified to $?x$, then their cell values become $< REF, ?x - address >$.

This gains for us the ability to recover local stack space on determinate procedure exit without losing the binding information established during the activation of these clauses. For example, if a successful return to clause C is a determinate procedure exit, the binding value for $?x$ is safe in the environment for C. Thus, if there are further goals to be invoked that may need this value, we are guaranteed that it will be available. And, if there are no further goals, the value is no longer needed. We simply succeed.

4. Global Variables

Global variables present a situation that does not occur with local variables. The reason that global variable environments must remain on the stack even after determinate procedure exit is as follows: it is possible that not all of the global variables in some of the determinately activated clauses have yet participated in unification, that is, they may still be undefined. And most importantly, if they do become defined at a later time, they have the chance to contribute to the success or failure of the proof.

Doesn't this situation occur with local variables? Not in the same way. In the example above, $?z$ may not yet be defined, but it is bound to $?x$. If $?z$ were later to become bound to a constant, this fact would be immediately reflected in the alteration of the cell value for $?x$, which is in the environment for clause C.

If $?z$ were a global variable, on the other hand, and it became bound to a constant, this fact would not be reflected in the environment for clause C, but only in the environment of the clause containing $?z$. This idea is illustrated in Appendix 6. The example in these figures make use of global anonymous variables, and should reveal why "global voids" cannot be treated as local voids.

There is a related, more mechanical reason that global variable space must be preserved on determinate procedure completion. Recall that the unification process is careful to avoid dangling references when it assigns the reference construct to value cells. This is done by considering the seniority of the cells concerned.

There is another construct that points into the stacks, however, that does not consider the seniority of the cell it references. This is the molecule, whose frame pointer can point to earlier or later environments. Appendix 6 illustrates this with a simple example.

5 The Prolog Machine and Our Implementation

5.1 The Prolog Machine

As we know, the output of the compiler is a sequence of macros plus literal data, the macros and outer literals placed in the code array, and the skeleton literals (inner literals) in the literal array. The code could be considered a form of intermediate language in a system that expanded them into native machine code (as the Warren compiler does). Our system uses the macros themselves as the primitive instructions, and interprets them in order to execute the user program.

The description of the Prolog Machine (PLM) and its instructions is straightforward in Warren (pages 36 through 38 and 49 through 58). We summarize the main points here, indicating briefly where we differ.

The PLM consists of the code area that contains the macros and arguments of the goals, and the stack area, containing the local, global, and trail stacks. (Technically, only the latter is a true stack. The local and global stacks allow random references to the data they contain, but do act like stacks when new environments are pushed or popped.) For Warren, the code area is "read-only", and during the normal course of execution, it is for us too. Since we allow the user to dynamically assert and delete clauses, however, there are times when we manipulate the code areas. The instructions of the PLM are the executable items, the literals are the fixed data. Recall that we store skeleton literals in their own array, to simplify the assert/delete problems, whereas Warren places them at the end of the code for a clause. (He does observe that they could be placed anywhere. See page 33.)

In general, the instructions generated correspond closely to the original source code. Executable instructions are generated for the arguments of the head, but only for terms nested at a level less than five. The deeper arguments are unified in a general unification routine. The motivation for explicit unification instructions for levels three and four is simply that, in practice, most unification is required only at these shallower levels and explicit instructions for this situation are less complex and faster than the general routine has to be.

No executable instructions are used for the arguments of a goal, however. These are represented by outer literals, as described earlier. The goal-predicates themselves correspond to CALL instructions. Prolog built-in goals (for example, "cut" and "fail") have their own macros, all of which are executable. Our system also allows calls to LISP as goals in the body of a clause. The LISP calls can refer to user-defined routines or to other built-in functions such as arithmetic and input/output functions. In addition to the unification and call instructions, the PLM includes initialization instructions that set value cells to "undef", and three flow control instructions, NECK, FOOT, and TRYLAST. The first marks the end of the head (where we place the atom "if" at level two), and the second marks the end of the clause. The TRYLAST is executed as the first instruction for the clause if it is the last clause of the procedure. It serves to reset the choice point registers (VV and $VV1$) to the appropriate deep choice points.

A fourth flow control instruction is added in our system, and in practice something equivalent must be included in any system, though Warren does not mention it. It is a CHECKARITY instruction, whose function is to fail the clause if the arities of the goal and head do not match.

5.2 Run-time Environment

Since we offer the user the ability to assert new rules or delete old ones as the program executes, our run-time environment differs somewhat from that described by Warren. Our compiler was designed to operate in close conjunction with an interpretive LISP. The LISP "half" offers input/output, arithmetic functions, and many built-in list manipulating functions, all accessible as goals in a Prolog rule. It is really a space management system that offers atom and cons cell space and a garbage collector, in addition to the various LISP routines. Currently, the arrays and run-time stacks used by Prolog are allocated and managed only by the compiler.

The atom space, however, is used throughout the compiler. This was important to make the two systems communicate properly. The user should be able to call LISP from Prolog or vice-versa, with each system aware of the current bindings for the atoms. Atoms, once allocated, are never duplicated nor deleted. Since only one copy of an atom is used, all attributes associated with atoms (for example, their names and propery lists) are immediately available from either LISP or Prolog. Our Prolog also makes use of cons cells, allocated by the space management system, and it is here that the LISP garbage collector does play a role.

After the user enters his source, the LISP "reader" processes the s-expressions, creating the internal equivalent (the standard linked cons cell representation), and the atom space. (For our purposes, we consider an atom to be any object that is not a list.) Most atoms are stored as atom cells composed of four pointers: the bind, rule, property list and print-name (pname) pointers. The last gives us access to a dynamic type (dtype) that can be associated with an atom.

5.3 Clause Attachment

Prolog makes use of the rule, pname, and dtype attributes of atoms. The compiler begins with the stored s-expression produced by the reader. If there are mode declarations for an atom, its dtype is altered to reflect this fact. A special cons cell ("mode cell") is assigned to the rule field whose cdr field is set to nil and whose car field contains the offset into an array that stores the declaration arguments.

As is illustrated in Figures 7 and 8 (see pages 31, 32), when a head predicate is recognized by the compiler, the rule field pointer is followed to a list of cons cells (hereafter called "clause cells"), chained together by their cdr fields. A new clause cell is added to the end of the list, its cdr field is set to nil, and its car is set to the code for the corresponding clause. If this is the first clause of a procedure with no mode declaration, the rule field is set to point directly to the new clause cell.

During the processing of a goal, the rule field of its atom is checked to see if either a mode cell or clause cell has been assigned. If it is still 'nil', it is set to a cons cell whose cdr is always nil, and whose car points to TRYLAST followed

by DEEPFAIL. The effect during a run is to fail any goal that has no associated clauses.

The mechanism for recognizing determinate procedure exit is set up during clause attachment. The VV and $VV1$ registers are set back to the deep choice point on entry to the last clause of a procedure. This is the function of the TRYLAST macro, which must, therefore, be the first macro executed when this last clause is activated.

However, since any clause may become the last alternative during a program run (by the action of an assert or delete), we need a method for executing the TRYLAST macro as the first instruction for a last clause, without executing the macro when the clause is not the last. We solve this by providing the TRYLAST macro at the beginning of the code for every clause, and manipulating the car field of the corresponding clause cell. We set it to point to TRYLAST when the clause is truly last, but otherwise point it to the second macro (always CHECKARITY). (See Figure 11, page 35.)

5.4 Specifics of this Implementation

The specifics of our implementation are described in the appendices and figures of this report. This section will serve as a summary and to help direct the reader to the appropriate appendix or figure for more complete information.

Let us take an operational view point of the compiler. The user interface (called "commands") is common to both the LISP and the PROLOG systems. When the user invokes the compiler, it prints out the following message.

```
PROLOG/LISP  IV.1.1  Feb 1985
```

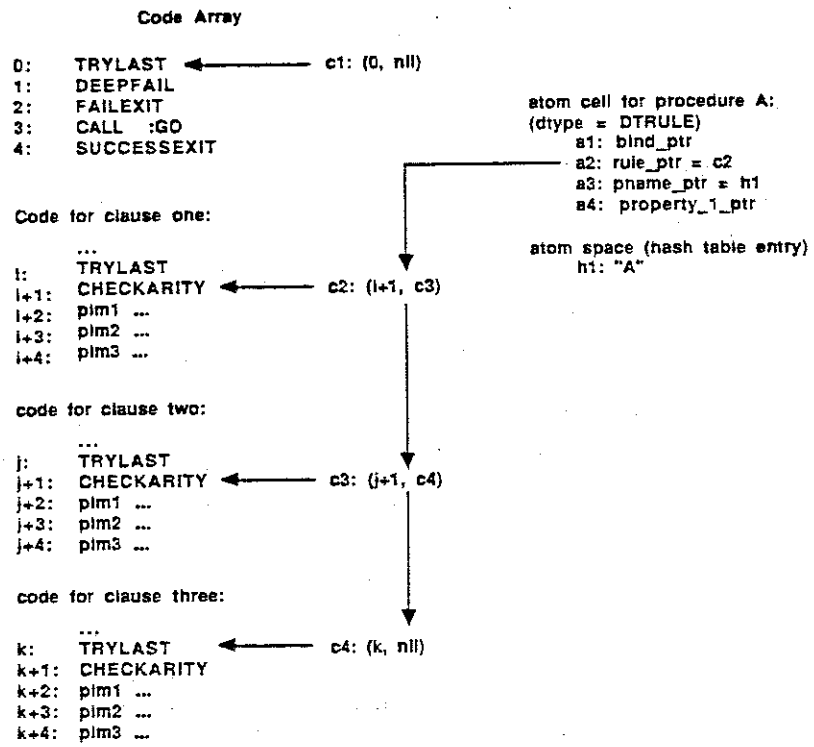
and also prints the prompt ':-'. The top level interface is just a "read, process, output" loop. At this level (top), the user is free to type in a Prolog program , a Lisp program, a Prolog goal or a Lisp function. The top level interface accordingly calls the PROLOG or LISP system, prints whatever message is returned by the called system and waits for user input.When a prolog program is asserted, the following actions take place.

The LISP reader processes the source and hands the compiler the equivalent s-expression in its linked-cons-cell form. The compiler scans through this tree of cons cells, searching for an atom or 'nil'. If the keyword "mode" is found at the correct level, a vector of mode symbols is processed next. The mode array is used to store this information and can be accessed even at run-time because the index into this mode array is made available via the atom cell for the procedure involved. This is illustrated in Figure 8, page 32.

After the mode declarations have been processed, we encounter the clauses of the program. As a clause is scanned, atoms are entered into the item list array,

Notes:

- Addresses are followed by a colon, though we use arrows as much as possible for clarity.
- plm1, plm2, etc. represent arbitrary PLM instructions.
- clause cells are shown as: address:(car-address1,cdr-address2)

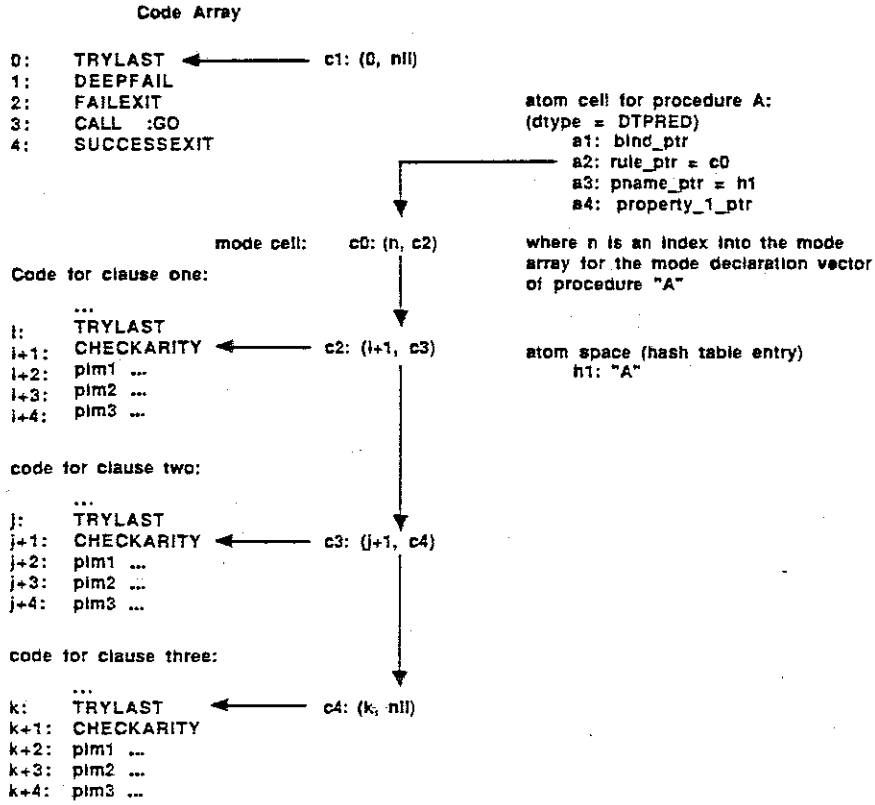


This figure illustrates the code array containing the code for a three-clause procedure named *A*, the associated clause cells, and atom cell. There is no associated mode declaration for *A*. See Figure 8 for an example of this.

Figure 7: Clause Code Access With No Mode Cell

Notes:

- Addresses are followed by a colon, though we use arrows as much as possible for clarity.
- plm1, plm2, etc. represent arbitrary PLM instructions.
- clause cells are shown as: address:(car-address1,cdr-address2)

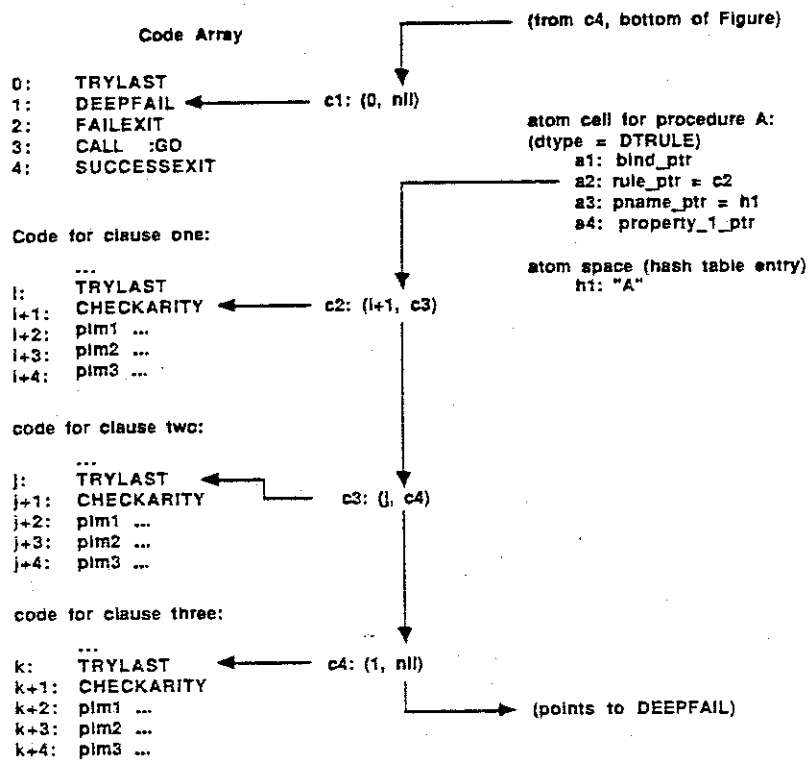


This figure is the same as Figure 7, except here procedure *A* has an associated mode declaration, so its atom cell has a dtype of DTPRED.

Figure 8: Clause Code Access With Mode Cell

Notes:

- Addresses are followed by a colon, though we use arrows as much as possible for clarity.
- plm1, plm2, etc. represent arbitrary PLM instructions.
- clause cells are shown as: address:(car-address1,cdr-address2)

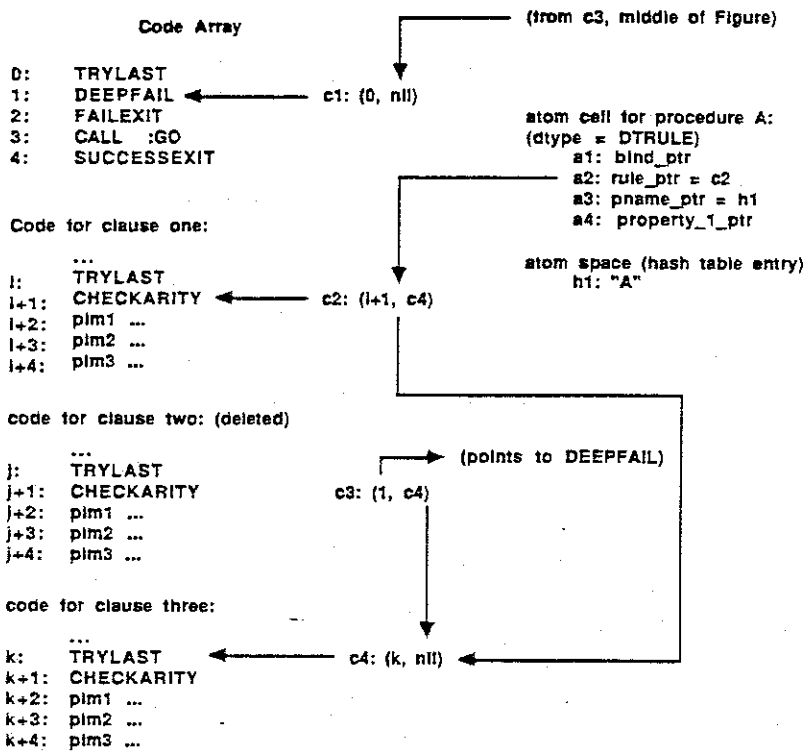


This figure is based on Figure 7, showing the situation after clause three has been deleted.

Figure 9: Clause Code Access: After Deletion

Notes:

- Addresses are followed by a colon, though we use arrows as much as possible for clarity.
- plm1, plm2, etc. represent arbitrary PLM instructions.
- clause cells are shown as: address:(car-address1,cdr-address2)

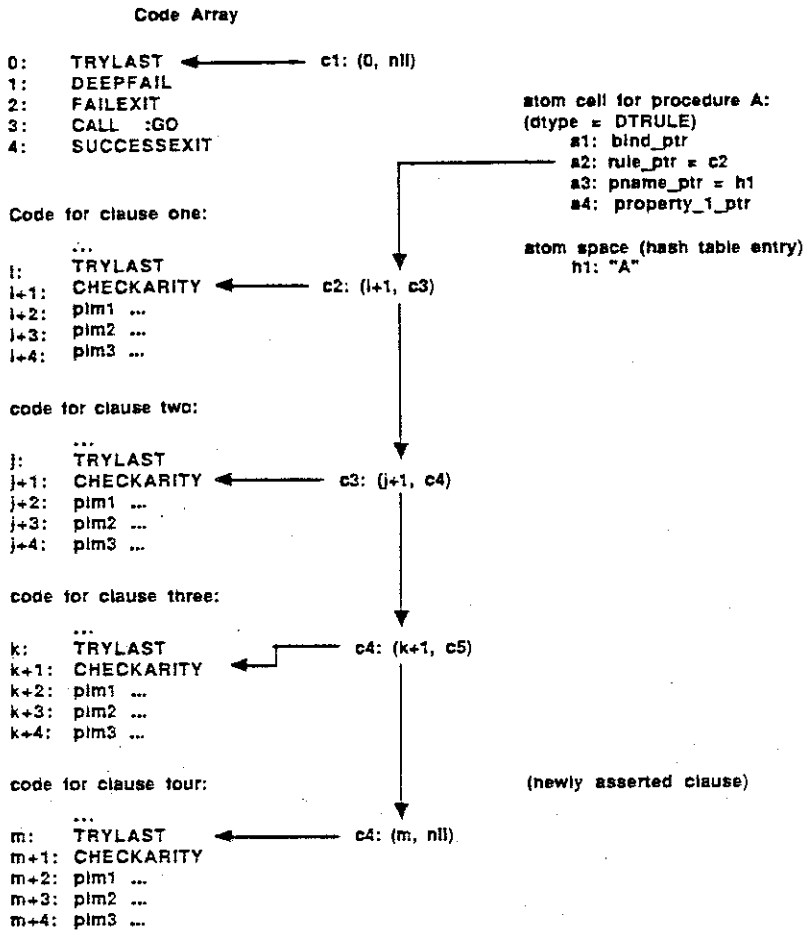


This figure is based on Figure 7, showing the situation after clause two has been deleted.

Figure 10: Clause Code Access: After Deletion

Notes:

- Addresses are followed by a colon, though we use arrows as much as possible for clarity.
- plm1, plm2, etc. represent arbitrary PLM instructions.
- clause cells are shown as: address:(car-address1,cdr-address2)



This figure is based on Figure 7, showing the situation after clause four has been added.

Figure 11: Clause Code Access: After Assertion

which has fields for a pointer to the atom cell, the type of the atom, its argument number, its level, and its frequency of occurrence to this point (if a variable). If the atom is a variable, it is hashed into the symbol table. In this case, the atom pointer field of the item list is used to hold the symbol table index.

There are currently some 36 different types for entries in the item list. These include GOAL, HEAD, VARIABLE, ATOM, NUMBER, NIL, and DOT. Additionally, the type often identifies a built-in goal such as CUT, FAIL, NOT, LISP, ASSIGNMENT, ASSERT, and so on. Finally, the type field gives information on the type of compound term encountered (thus the item list does contain entries for certain non-atoms), such as USKEL, USKELC, USKELD, and USKEL1.

The symbol table holds all information that is inherent to the variable wherever it occurs, that is, information that is independent of the position of the variable in the clause. It has fields for the atom pointer, for the classification of the variable (LOCAL, GLOBAL, etc.), for the variable number, and for the frequency of occurrence in the entire clause. Of course, some of this information is not complete or correct until the entire clause has been scanned, whereas the information for an item list entry is available when the item is encountered in the clause.

See Appendix 7 for a brief description of the routines nextatm, nextatmx, classify, and builtin, which are driven by the routine assertx. These are the routines whose task is to build the entries for the item list and the symbol table for each clause. Appendix 8 has a description of the arrays used in the compiler, including the item list and symbol table. For the definition of argument numbering, see Figure 2, page 8. Finally, see Figure 12, page 41 for a diagram of the calling structure of the compiler routines.

After a clause has been scanned, the variable numbers can be assigned. This is not possible on the first pass since variable numbers for local and global variables form two separate sequences, and the classification of variables is not complete until the entire clause has been seen. See Appendix 7 for a description of assignvar, and Figure 3, page 13, for the variable numbering definition.

The attachment to the rest of the procedure of the code for this clause is accomplished next (though the code has not yet been created, its location is known). Attachment simply involves allocating a new clause cell (Figure 11, page 35) for this clause code, and linking it (usually) to the end of the list of clause cells already established. An argument to the attach routine determines which end of the list to attach the clause cell.

Once the symbol table and item list entries are complete, the routine GENMACRO is called (Appendix 7) which uses this information to generate the PLM code for the clause.

Finally, the symbol table information is cleared to prepare for the processing of the next clause (see the routine clr in Appendix 7). It is not necessary to clear the item list; that information is simply overlaid.

When the user types in a query at the top level, a 'query' clause is constructed from this which contains the query as a goal. See the description of the routine get query for the format and function of this clause (appendix 7). After the query clause is asserted, the PLM machine is invoked ("run"). The operation of the PLM is the subject of Chapters 3 and 4 with examples in Appendix 5.

5.5 Assert Processing

The ability to dynamically assert new clauses or delete previous ones was a major feature added to the basic Warren implementation. The idea of providing access to the code for a procedure via the atom cell and a list of linked cons (clause) cells was developed for facilitating asserts and deletes.

Asserting a new clause is straightforward. The argument to an assert goal (the clause to be asserted), is entered into the expand array during the original processing of the source. During the execution of the program, the goal "assert" or "assert0" causes an expansion of its argument (see Appendix 7, expand routine) that dereferences all the variables to their current bindings. The result is a linked cons cell representation of a clause instance (s-expression) that is passed to the assertx routine. In effect, the compiler is called during run-time, a technique known as incremental compiling.

Attachment of the new code is at the beginning of the list for "assert0", and at the end for "assert". If the the goal was "assert", some adjustment to the car field of the clause cell for the old last clause is necessary: it is no longer last, so this pointer must now direct control to skip the TRYLAST macro. (See Figure 11, page 35, and Section 5.3 above.)

There is one difficulty with this procedure that the current compiler does not solve. When the last clause of a procedure (call it clause C) has been activated, the TRYLAST macro sets the VV and $VV1$ registers to the deep choice point in anticipation of failure. (See Section 3.3) Now, suppose a new clause is asserted and added to the end of this procedure while the clause C is still active. If clause C fails, the user may expect the new clause to be entered. But since VV and $VV1$ have already been reset to the deep fail environment, deep failure will occur, thus overlooking the new clause.

On the other hand, if a clause (clause B) earlier than clause C had been active when a new last clause was asserted, then failure of clause B would cause shallow backtracking. In this case, the new clause will be available for entry if clause C eventually fails since the TRYLAST macro for clause C was skipped when the new clause was added to the list.

This behavior is undesirable since it recognizes a newly asserted clause according to what the user would see as an arbitrary rule: Suppose a clause Cnew is dynamically added to the end of a procedure that has clause C as its current last

Make a second pass through the same portion of the local stack, searching for VV fields equal to VVsave. Change these fields to Cptr (thus making the environment for clause C the deep choice point for these environments).

We are not done: there may have been other invocations of procedure P that own active environments for clause C, each with a different value in their VV fields. Thus a complete search through the local stack would be necessary, with two passes as described above for each such environment found.

Another difficulty involves recognizing which field in the local stack is an A field. There is no way that a search in either direction through the local stack can simply jump from management area to management area. An exhaustive search through each stack entry is necessary. Recognition of an A field would be possible since it points into the code area that has determined (or easily determinable) boundaries. Skeleton literal pointers also do in Warren's implementation. For such an implementation, the TR field (which is one stack entry below the A field) could be searched for instead, since the trail boundaries are also known.

As can be seen, this is an expensive process. It may be possible to gather information during run-time to facilitate the search, but then we shorten the assert processing at the expense of many other routines.

5.6 Delete Processing

The implementation of delete is more difficult, if we ignore the VV field update problem of assert (as we do). We are able to delete clauses without a search of the stack. However, delete involves a matching step not needed with assert. The routines already available for ordinary goal-head unification are useful in part, but our Prolog enables the user to specify the entire clause to be deleted. Thus, we need routines that are able to unify a clause against a clause.

Another issue becomes apparent when both asserts and deletes are allowed. Currently, code for newly asserted clauses is added to the end of the code array. But the code array develops unused areas when clauses are deleted. It would be very desirable for subsequent asserts to use these slots. Clearly, some form of memory management scheme is needed here. Rather than use deleted code areas directly, we are leaning toward a system to compress the code array when new space is needed (a "compaction" scheme).

Note how any system would be made more complex if the skeleton literals in a clause were stored after the code for that clause, as Warren does. This is true since skeleton literals may participate in the representation of clause instances even if the clauses containing the corresponding skeleton terms may not be active.

When a clause D is deleted, we must alter the clause cells for this and adjacent clauses. The address of clause cell D will sit in the FL field for any environments for which clause D is an alternative. Thus, if we are to avoid a search for these

clause. Cnew will be recognized on failure of clause C if and only if Cnew was asserted before clause C was activated. Ideally, however, either a newly asserted clause should not be recognized until the procedure is invoked again, or it should be recognized immediately, no matter which clause of the procedure is currently active.

How could we implement an assert that would overcome this difficulty? There are at least two possibilities. One would be to delay the execution of TRYLAST until after all clauses in a procedure have been executed. This would be easy to implement: simply add an extra clause cell at the end of the list whose car field points to the TRYLAST and DEEPFAIL instructions that reside in the first two slots of the code array. Unfortunately, this technique entirely circumvents the recognition of determinate procedure exit. This is too large a sacrifice to make. Recall that we are able to recover a possibly large amount of local stack space on determinate procedure exit. This savings may well make some program possible to run that could not operate otherwise. Also, we gain a "fast fail" capability, since the deep choice point is guaranteed to provide to us, in one step, the lastest environment with alternative clauses. Otherwise, we would have to search for it in a way similar to the current success processing (see Section 3.4). And finally, recall that the entire motivation for distinguishing between local and global variables was to enable the recovery of local stack space on determinate procedure exit. It is clear that a lot would be thrown away were this simple "solution" chosen!

Thus, it seems that a search through the local stack cannot be avoided. Both the search itself and the definition of what we are searching for are difficult. A partial solution is offered here. This should at least give an idea of the reasoning needed for a full solution that relies solely on information available in the run-time environment as it is currently defined. The following technique appears to cover all possibilities, but should be tested!

Once again, call "clause C the current last clause of a procedure to which a new clause is about to be appended. Call this procedure "P". We must alter the VV field in the management area for any environment created that has a goal of clause C for its parent goal. Currently, it will have a deep choice point equal to the deep choice point of the environment for clause C (recall that clause C was the last of its procedure). We wish to alter it to point to the environment for clause C itself. (This will be the new deep fail environment, since it is about to receive an alternative.)

So, we move up from the current environment (belonging to the clause that invoked the assert), in search of an A field such that A-1 points to a CALL P instruction and whose FL field is "nil" (indicating that this environment corresponds to clause C). Save the VV field of this environment in VVsave, and save the location of this environment in Cptr. Also, alter the FL field to point to the clause cell of the newly asserted clause.

environments we must redirect pointers in the clause cells. The clause cell D is redirected to point to the clause cell after D, if there is one, else the car field for the clause cell D is redirected to point to the DEEPFAIL macro in the second slot of the code array. This will immediately fail any attempt to enter the deleted clause.

If this were the only strategy, then long chains of failure pointers of this type may build up if many clauses are deleted. To prevent this, we redirect the cdr field of the clause cell ahead of clause cell D (call it clause cell C) to point to the clause cell after D, if there is one. This at least will help any new activation of the procedure, or any activation currently underway for which the FL field points to cell C. See Figures 9 and 10 for illustrations, pages 33, 34.

6 Conclusion

There are several main points we tried to convey in this report. Foremost among these is an acknowledgement and an explanation of Warren's important contributions to the implementation of a Prolog compiler. The major features of a Prolog system divide into unification issues and execution flow control issues, and Warren's contributions divide into these categories. Thus, his idea of the mode of an argument (essentially an improvement to unification processing) is significant for improving the space and time requirements of the run-time modules. His observation on the fundamental difference between global and local variables gives us a "fast-fail" ability and a large savings in local stack space.

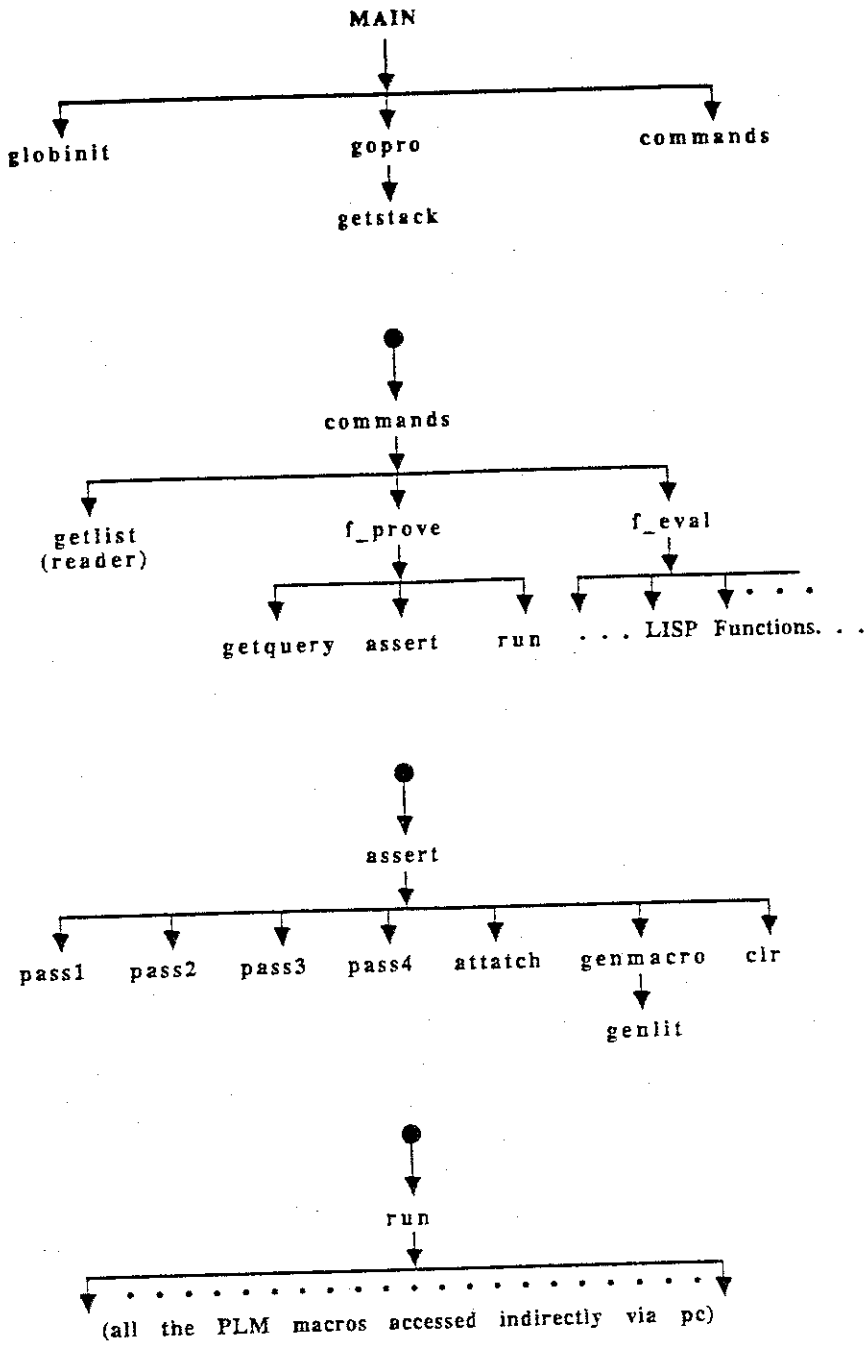
This report also treats in some detail the differences between our implementation and Warren's. The differences are mostly related to the adaptation of Warren's DECSystem 10 implementation to our VAX 11/780 environment. But there are several basic features that all Prologs offer which Warren does not treat. This report addresses these issues.

The contributions that we offer in this paper include a description of the implementation of dynamic assert and delete plus suggestions for their improvement. Also described (in Appendix 3) is a way to implement the not operator. On the theoretical side, an analysis of how Warren's variable classification scheme works is presented (which Warren does not cover in his report).

6.1 Why Compile?

Here we summarize perhaps the most basic message of this report: why compile and not interpret? For those readers who are reading this chapter first, we give brief definitions of most of the terms used.

Compiling a Prolog program involves gathering certain information from the source clauses at compile-time. In general, this information is only complete when



This diagram gives a high-level view of the control structure of the compiler. It includes only the major routines.

Figure 12: Calling Structure of the Compiler

the full clause has been scanned. An interpreter, of course, cannot afford this luxury. Operating without such information puts a run-time computational burden on an interpreter that is avoided by the compiler. Warren [p.50] gives examples of the type of information that is readily available only to a compiler. In summary, this includes the following:

1. The first occurrence of a variable in the head of a clause represents a special case of unification that is much simpler than matching against a subsequent occurrence of the same variable. Consider, for example, $((member\ ?x\ (?x.?y)))$ Here, we term the first $?x$ a "first occurrence", and the $?x$ in the skeleton is a "subsequent occurrence" of the same variable. If one conceives of the matching processes as proceeding from left to right, this leftmost occurrence of the variable must be still be uninstantiated, so unification will always succeed: it merely has to bind the variable to the incoming term.

2. Value cells, the entries on the run-time stacks that record current variable bindings, do not have to be initialized to 'undef' if they are associated with the first occurrence of local variables. In the event of failure, other cells may not need to be initialized to 'undef', so this initialization is postponed to the last possible moment.

3. Void variables are those which are not in a skeleton and have only a single occurrence in the clause. No action needs to be performed for void variables in the head of a clause, and no cell is needed on the stack for such variables in a goal.

4. A full scan of the clause enables the classification of variables according to the length of time it is necessary to remember their values. (For example, as we learned in the introduction, global variables need to be preserved longer than local variables. But the recognition of global variables can only be assured with a full scan of the clause.) This enables popping of the space they occupy on the stack earlier than is possible with an interpreter.

5. The last clause in a procedure can be recognized. This case is important for the detection of determinate procedure exit, which enables early recovery of stack space.

6. (Not yet implemented.) Some of the unification work can be performed at compile time if the clause heads contain constant data. This is particularly useful when there are many clauses in a procedure (for example in a database of unit clauses–think of a telephone directory). Some preliminary "indexing" work by the compiler could reduce such a situation to a table-lookup rather than the normal more costly entry into each clause.

Appendix 1

Suggestions for Future Enhancements

This compiler represents a starting point. The first hurdle was understanding Warren's report and adapting his design to the VAX 11/780 architecture. The addition of the dynamic asserts and deletes was the major improvement over the basic design.

This appendix may serve as a starting point for a project of continued development of the compiler. It contains a brief annotated list of minor and some not-so-minor additions or enhancements that occurred to us during the development of the current system. They vary in implementation difficulty from trivial to probably impossible.

It should be noted that Warren has improved his 1977 design in a new report [Warren, 1983]. He treats some of the following ideas and many additional ones. This document should be consulted before any extensive enhancements are made to our system.

Additions to the language environment:

1. Additional builtin functions should be provided. Some of these are properly a part of any Prolog system. Others fall into the category of convenience features.

   - the goal predicates AND and OR
   - a gensym atom facility
   - immediate unification (==)
   - predicates such as ATOM, VARIABLE, NUMBER, NIL, BOUND
   - (RETRACTA head-predicate) and (RETRACTZ head-predicate), fast-executing versions of RETRACT that would delete from the rule base the first (or last) clause of a procedure, without the overhead of unification and searching inherent in the normal RETRACT.

2. Variable goals would add an interpreter-like versatility to the compiler. If "goal" here means "goal predicate", the implementation is straightforward. However, if it refers to the compound term $(GOAL-PREDICATE\ TERM...)$, complications arise. For one thing, a proper classification of variables cannot be performed at compile-time unless all of them are explicitly available in the source clause. A possible solution to this would be to treat variable goals of this variety similarly to the way dynamic asserts are treated–by essentially

expanding the goal and calling the compiler on the resulting clause when the goal is invoked.

3. Allowing dotted expressions at levels less than four would similarly add to the expressive power of the language. The difficulty here is more severe, however. The distinction between local and global variables becomes blurred. As an example of this, consider the goal $(a?\ x.?y)$ matching against the head $(a\ 1\ 2\ 3)$. $?x$ unifies with 1, and $?y$ with the list (2 3). But how do we express this latter unification? We must have a skeleton literal to represent the "(2 3)" and a molecule becomes the cell value for the cell associated with $?y$. Thus "local" becomes "global", and the two stack idea would probably have to be sacrificed. It is not worth it.

4. A trace package, similar to the one offered by the current interpreters at Tech, is indispensable to program development. This could include printing of the source clause as it was entered, with each variable replaced with its current binding; a break package that would suspend execution at specified points; and selective tracing where only selected clauses would print as they were activated.

5. Additional data types, such as integers and arrays, could be offered.

6. The user could be given some control over the behavior of the compiler and possibly of the code generated (beyond the mode declarations). For example, he should have more control over the printing of error messages, and should be able to set the run-time stack size.

Implementation enhancements:

7 Some currently implemented LISP functions, such as arithmetic and input/output, could be implemented directly to avoid the expensive call to the LISP interpreter.

8 Currently, all clauses with the same head predicate are linked and accessed sequentially, without regard to their arities. Two clauses with the same head predicate but different arities should properly be considered members of distinct procedures. Indexing by head arity would be desirable: it would certainly be faster, and, recognizable at compile time.

9 One aspect of Warren's report not implemented is indexing (the other is garbage collecting). Indexing is a technique where the matching clause is found in essentially constant time, based on the form of the clause head. It

amounts to a partial unification step carried out at compile time. Warren suggests an indexing scheme that bases the selection of the clause on the head predicate plus first argument (provided this argument is constant). At run-time, the selection of the proper clause is reduced to computing a hash function.

10 Garbage collection could be performed on the global stack. Just as space on the local stack can be recovered on determinate procedure exit, some of the value cells on the global stack become recoverable at this time through the processes of garbage collection. Warren notes that since it is an expensive process, garbage collection should only be invoked when global stack space is exhausted. The process is more complex than the standard trace and mark algorithms, since a compaction of value cells and a remapping of addresses that refer to the global stack must take place. As Warren notes, not all inaccessible cells can be discarded since structure sharing requires that the offsets of the value cells remain constant. Thus, if "active" cells surround an inaccessible one, it may not be discarded since the offsets from the beginning of the frame must be maintained for all the cells.

11 Garbage collection could be performed on the code array. This is not an issue in Warren's implementation, since he does not allow dynamic asserts and deletes. The issue here would be whether to perform a compaction, (and when), or whether to undertake a more complex, but faster, memory management scheme such as a best or first fit algorithm for the code of newly asserted clauses. Note how storing skeleton literals at the end of the code for a clause, as Warren suggests, complicates this issue. Just because a clause has been deleted does not imply that there are no molecules on the stacks with references to the skeleton literal for the clause.

12 Can a more compact representation of skeleton literals be found? Unlike the situation with value cells, the type field of the car and cdr of skeleton literal cons cells is never used to hold addresses. But a full 32 bits are reserved for this information.

Another, more challenging possibility: Can the arguments of the skeleton be stored in a more array-like manner? Access could be faster, and storage would certainly be more compact. References to nested lists would still be via pointers to other arrays. The only difficulty appears to be in the representation of the dot. How would the storage of $(a\ b)$ be distinguished from that of $(a.b)$? A more severe difficulty would be the necessity to redesign much of the unification logic, possibly at the expense of speed and simplicity.

13 A "fast-succeed" capability would be very desirable. This has a direct analogy

with recognition of determinate procedure exit that enables, on failure, a direct transfer of control to the procedure that has alternative clauses (a "fast-fail" capability). Just as this case requires recognizing when the last clause of a procedure is activated, a fast-succeed feature would need to know when the last goal of a clause were invoked. In this case, it is not necessary to "call" the goal, and no return address need be stored. Something equivalent to a simple "goto" is needed here since on success control needs to transfer to the most recent currently active clause with a goal (and not a "foot") for its continuation. (Note the analogy with the "latest choice point".) This would be the last clause that did push a return address on the stack. If the last goal of a clause is a recursive call, this technique would be particularly useful, and amounts to optimization of tail recursion.

14 A considerable execution speed improvement should be possible by eliminating the function calls currently used when interpreting the macros in the code array. The C language does not offer a computed goto, so there seems no way to replace the function calls with a faster mechanism, if we stick to C. A more radical change would speed things considerably. If the macros were translated to inline machine code, then not only would the function calls be eliminated, but the efficiency advantages of native code programming could be realized. Note that arguments to the macros would have to be masked in place.

The disadvantage here would be the extensive memory requirement. Some compromise between inline macro expansion and routine calls would be needed. Certainly the general unification routine, for example, would be called rather than replicated for every unification instruction. In fact, this is the approach used by Warren, and some hints on a good balance between macro expansion and routine calls could be gleaned from his Appendix 2 [Warren, 1977, Volume 2].

15 If the code array concept as currently employed were kept, there is still an important space-saving technique that could be included. The code array uses four longwords for each entry (see Appendix 8). But it is only some of the unification macros that require all four fields. Additionally, some of the macros use arguments that do not require a full 32 bits. In short, the code array could be implemented using a variable length entry for each macro. Since each macro is currently responsible for incrementing the pc by one (actually, by 16 bytes), they could easily be modified to increment the pc by an amount determined by the macro type. (Note that this is reminiscent of the way the VAX increments its own pc during the fetch-execute cycle.)

16 The compiler needs fine tuning. Some simple statistics should be kept as it runs typical programs, to determine the number of slots used in the various

pre-allocated arrays. An initial guess had to be employed for their current sizes, but they may need to be altered. (See Appendix 8.) Also, the proportion of the large array used for the dynamic structures that should be allocated to the global, local, and trail stacks could be determined. Currently, the array is split in two, with one half used for the local stack, and the other half used for the global and trail stacks growing toward each other. A more convenient arrangement may be found. It appears, for example, that more space should be given to the local stack, which of course includes the management information. For large programs, however, the reverse may be true, since the global stack space is recovered less often.

17 There may be some minor inefficiencies that can still be eliminated from the macros as defined by Warren. For example:

- The common combination "...(*cut*) (*fail*))" could be combined into a single CUTFAIL macro (which includes FOOT) to eliminate some of the redundant code between the three. (Be careful of the trail cleanup here: CUT and FAIL perform different functions on the trail.)
- If a procedure contains a single clause, it is unnecessary for the CALL macro to set the VV and $VV1$ registers; the TRYLAST macro, which is executed next, simple resets them.
- If a skeleton contains global voids as its first and sometimes second arguments, the IFDONE macro will simply transfer control to the next macro, and is therefore unnecessary.
- If a USKEL instruction sees an incoming reference construct or a void, it sets Y to 0 (see Appendix 4) to trigger IFDONE to transfer around the skeleton argument instructions. This transfer could take place immediately from the USKEL instruction, eliminating the need for the extra conditional. USKEL would simply need a third argument (the same one IFDONE currently uses) to direct this goto to the proper instruction.
- There may be other such situations that are recognizable at compile-time. The question becomes, is the time required to recognize the condition worth the time saved in executing a more efficient macro stream?

18 Currently, whenever an INIT or LOCALINIT is determined to be unnecessary, a NO_OP macro takes its place. Clearly, these should be eliminated. The problem is that the first-occurrence analysis necessary to determine if INIT or LOCALINIT is needed takes place in the genmacro routine (see Appendix 7). It should be moved to the earlier classify routine, unless it is decided to expand the macros into their machine language equivalents (see 14 above).

In the latter case, the NO_OPs could be eliminated during the final expand routine.

Appendix 2

Constructed Term Representation: Reading Results from the Stacks

This appendix should clarify how skeleton literals and cell values (constructs) work together to represent an instance of a term (a constructed term). We use the "reverse" example of Appendix 5. Reproduced below is Listing 5, modified to show only the state of the stacks just prior to printing the answer. We describe how to read the answer using the cell values on the stacks and the skeleton literals stored in the literal array. Understanding this should shed light on the unification process and clause instance representation problem as a whole.

Note that the process illustrated below is in part the function of the routine expand.

```
                    LOCAL STACK                    GLOBAL STACK

stack address                    stack address
          |                                |
          |                                |
          V   type       pointer           |
                                           |
VV,FL     1|         1|    367632|         | environment for user query clause
 X, A     2|         1|         4|         | starts at local stack address 1.
V1,TR     3|      1001|        48|         | User query:
          4|      1004|        s0|         |((:GO) if (reverse (a b) nil ?ans))
VV,FL     5|         1|    367648|         |
 X, A     6|         1|        38|         V    type      pointer
V1,TR     7|      1001|        48|
                                       1001|  ATOM    |          a|
                                       1002|     1001|         s3|
                                       1003|  ATOM    |        nil|
          8|  REF     |         4|
VV,FL     9|         1|    367648|
 X, A    10|         5|        26|
V1,TR    11|      1004|        48|
                                       1004|  ATOM    |          b|
                                       1005|  ATOM    |        nil|
                                       1006|     1001|         s0|
         12|  REF     |         4|
VV,FL    13|         1|    367648|
```

```
 X, A    14|          9|         26|
V1,TR    15|       1007|         48|
         16|       1004|         s0|
(reverse (a b) nil (b a))

skeleton literals:
  s0:    GLOBAL  0      GLOBAL  2
  s1:    GLOBAL  0      GLOBAL  1

  s2:    ATOM    'a'    SKEL    s3
  s3:    ATOM    'b'    ATOM   'nil'
```

Note that local stack address 4 is the value cell for *?ans* (It may appear to be a void variable, but not shown is the print goal that also contains *?ans*, so in fact it is local.) That is, the source term whose term instance we are looking for is *?ans*.

As we see, the cell value for the variable ?ans is the molecule

$$< frame\_ptr, skel\_ptr > = < 1004, s0 >$$

Any molecule represents a nested term, so we provide a set of parentheses. We write the address of the skeleton literal within the parentheses, and next expand *s0*:

```
(s0)
```

According to the inner literal at s0, we may expand this to:

```
(GLOBAL 0 . GLOBAL 2)
```

The current frame (1004) gives the frame address to which we add these offsets. The cell value at 1004 + 0 is <ATOM, b>. Thus we have:

```
( b . GLOBAL 2)
```

The cell value at 1004 + 2 is another molecule: <1001, s0>. Thus we provide another set of parentheses, and a skeleton literal address, and update the current frame to 1001.

```
( b . (s0))
```

As before, this expands to:

(b . (GLOBAL 0 . GLOBAL 2))

Since the current frame is 1001, we look in value cell 1001 + 0 to get:

(b . (a . GLOBAL 2))

Finally, we substitute the value at 1001 + 2:

(b . (a . nil))

or

(b a)

Appendix 3

Annotated Psuedocode for PLM Flow of Control Logic

This appendix illustrates how the registers control the execution of the Prolog Machine, and how they keep track of the stack environments.

In the following pseudocode, the fact that the Prolog clauses have been compiled into PLM macros has been, for the most part, ignored. Considering that this code is operating directly on the source clauses should cause no confusion. However, in order to take into account the underlying macros, translate, for example, "address of a clause" into "address of the first macro for the clause".

Another simplification assumed here is that the FL register is considered to point to a clause (or its first macro). This does, in fact, reflect more closely the Warren approach. In our implementation, FL always points to a cons cell whose car field addresses the first macro of a clause and whose cdr field points to the cons cell for the next clause (or is nil).

```
Notation:

   R(E)  refers to the field in the management information area for PLM
         register R of the environment pointed to by register E.

   :=    assignment

   =     equality (comparison operator)
   !=    inequality (comparison operator)

   L:    label--destination of goto

   ;     statement separator

   { }   compound statement delimiters (used only if needed for clarity)

   *     comment

   Example:  VV := VV(V) assigns the contents of the VV field of the
   environment pointed to by V to the register VV.

The PLM registers:

   V       top of the local stack
```

```
V1      top of the global stack

X       pointer to local frame of environment for parent
X1      pointer to global frame of environment for parent

VV      pointer to local frame of environment for the latest
           choice point
VV1     pointer to global frame of environment for the latest
           choice point

A       pointer to arguments of the current goal (points into
           code area)
FL      pointer to clause to execute on failure of
           current clause (provided there is an alternative)
TR      pointer to top of trail
```

Recall:

The local and global stacks are two addresses (longwords) in width. The PLM management information is stored on the local stack, requiring three locations (= six addresses). It is assumed here that pointer arithmetic accounts for the width of the stack. Thus, if V points to the management information area, $V + 3$ addresses the first value cell for this local frame.

This code assumes that local and global stacks grow toward larger addresses and the trail grows toward smaller ones (as in our implementation).

* * * * * * * * * * * * * * * * *

```
* initialization

      assert (add to rule base):

         ((GO) if (user-query-goal ...) (SUCCESSEXIT))
         FAILEXIT

* The  FAILEXIT  procedure  replaces  the second GO clause and results
* in printing  ''nil" and  exiting the  program if the ((GO) ...)  clause
* fails.  Alternatively, the second clause could be
*           ''((GO) if (failexit))"
* for the same effect.  Of course ''GO" must be some unique procedure
* name.
```

```
   * consider ``(GO)" a user-defined goal:
      pc := address of (GO) goal

      V, VV, X     := bottom of Local Stack
      V1, VV1, X1  := bottom of Global Stack
      TR           := bottom of Trail Stack


TOP:              * loop ...

      case (pc)
         if user-defined goal,     goto CALL
         if LISP goal,             goto LISP
         if ! (not),               goto CALLNOT
         if ONFAIL                 goto ONFAIL
         if ONSUCCESS              goto ONSUCCESS
         if cut                    goto CUT
         if fail                   goto DEEPFAIL
         if foot (no more goals),  goto FOOT
         if SUCCESSEXIT            exit


CALL:

* Note that goal pointed to by pc is now parent goal.  Call it P.

      PROC := goal-predicate of goal P.
      A := address of argument 0 of goal P.

* initialize new environment at V
      VV(V) := VV
      X(V)  := X;      A(V)  := A
      V1(V) := V1;     TR(V) := TR

* update choice points
      VV := V;         VV1 := V1

* remember where we started
      TRhold := TR
```

```
* This will be the current clause.  Call its head Q.
      pc := address of first clause of PROC, the called procedure

      FL := address of second clause of procedure PROC (or nil)


CLAUSE ENTRY:

      if pc points to the LAST clause in procedure PROC
      * Reset previous choice points in anticipation of failure.
        V := VV(V);   VV1 := V1(VV)

      if arity of P != arity of Q, goto FAIL

      unify arguments in Q with arguments in P

* The argument instruction located at pc carries its argument position
* n. The PLM attempts to unify the argument at pc against the parent
* goal argument at A + n, then increments pc.

      if unification fails, goto FAIL

* if reached NECK ...
      if current clause has a body {

      * store FL in case needed on failure of subsequent clause
      * -- it's our pointer to next clause of current procedure
        FL(V) := FL

      * current environment becomes the parent environment
        X := V;  X1 := V1

      * increment stack pointers to point to a new environment
        V := V + 3 + number of local variables in current clause;
        V1 := V1 + number of global variables in current clause;

        pc := address of first goal in body of current clause
        goto TOP

    }
```

```
* else, reached NECKFOOT ...
      else {

      *increment global stack pointer to point to new environment
         V1 := V1 + number of global variables in current clause;

         if V <= VV {    * more clauses -- nondeterminate exit

      * store FL in case needed on failure of subsequent clause--it's
      * our pointer to next clause of current procedure
            FL(V) := FL

      * no value cells since no local variables, i.e., only have manage-

      * ment information to skip
            V := V + 3
         }

      * arity of Q = arity of P
         pc := A + arity of Q

      * pc now points to goal at parent's continuation
         goto TOP

      }


FOOT:

* if choice point is an environment above parent's, we can recover local
* stack space (determinate procedure exit)
      if X > VV, V := X

* restore parent's environment as the current one, restore ''grand-
* parent's" as current parent
      A := A(X)
      X := X(X);  X1 := V1(X)

* point pc to goal at original parent's continuation
      pc := A + arity of head of original current clause
```

```
      goto TOP


FAIL:

* if no alternative clauses (current clause was last clause, so VV has
* already been set back)
      if V != VV, goto DEEPFAIL

   * clean up trail
      if TR != TRhold {
   * then some cell values created during unification of head of current
   * clause that would not be discarded automatically on failure (they
   * are in environments prior to choice point environment = current
   * environment)

         set to UNDEF all value cells
            pointed to by TR+1,...,TRhold
   * Note that our trail grows toward lower addresses.

         TR := TRhold
      }

      next := FL; FL := address of next clause; pc := next

      goto CLAUSE ENTRY


DEEPFAIL:

* pop stack back to latest choice point
      V := VV;  V1 := VV1
   * Note that at this point V = VV and V1 = VV1 as they always do when
   * a new call or clause entry is performed (as we are about to do).

* restore registers to reflect change of environment
      FL := FL(V)
      A := A(V)
      X := X(V);   X1 := V1(X)
      oldTR := TR(V)
```

```
* clean up trail
      if TR != oldTR {     * same comments apply as for FAIL

         set to UNDEF all value cells
            pointed to by TR+1,...,oldTR
   * Note that our trail grows toward lower addresses.

         TR := oldTR;  TRhold := TR
      }

      next := FL; FL := address of next clause; pc := next

      goto CLAUSE ENTRY


CUT:

* take latest choice point to be that of parent (note that the ''parent"
* is now the clause containing the cut)
      VV := VV(X);   VV1 = V1(VV)

* set top of stack to environment below parent's
      V := X + 3 + number of local variables

* tidy up trail
      if X <= VV {
* if latest choice point is younger (later on stack) than parent, there
* is nothing to cut

         P := TR(X)
         if (P != TR) {   * something trailed
* recall that a new value cell is trailed only if it is above the
* current choice point.  Cut is setting the choice point back, so some
* trail entries may become references to unused portions of the stack,
* if failure should occur.

            Check each item from TR to P.  Keep on the trail only those
            value cells that are < VV (or VV1) and discard the rest.
            Keep surviving trail entries contiguous.
         }
      }
```

```
        pc := address of next goal
        goto TOP


CALLNOT:

* during compile, a clause body of the form
*       if  ... (before)
*               (!(goal ...))
*               (after) ...
*
* is transformed to
*       if  ... (CALL before)
*               (CALLNOT goal)
*               (ONSUCCESS dummy_env)
*               (ONFAIL)
*               (CALL after) ...

* redirect failure label to (ONFAIL) goal:
        FL := address of (ONFAIL)

* V holds location of dummy environment
        record V in ONSUCCESS instruction

* create a dummy environment to hold the FL with ONFAIL address
* established above
        VV(V) := VV;      FL(V) := FL
        X(V)  := X;       A(V)  := A
        V1(V) := V1;      TR(V) := TR

* Note that goal pointed to by pc is now parent goal.  Call it P.

        PROC := goal-predicate of goal P.
        A := address of argument 0 of goal P.

* update choice points
        VV := V;          VV1 := V1

* increment top of local stack for true goal environment
```

```
        V := V + 3

* remember where we started
        TRhold := TR

* This will be the current clause.  Call its head Q.
        pc := address of first clause of PROC, the called procedure

        FL := address of second clause of procedure PROC (or nil)

* initialize new environment for true goal at V
        VV(V) := VV
        X(V)  := X;      A(V)  := A
        V1(V) := V1;     TR(V) := TR

* update choice points
        VV := V;         VV1 := V1

        goto CLAUSE ENTRY



ONFAIL:

* reached if ``(goal)" fails in ``(!(goal))" goal, so redirect failure
* forward to next goal

* Reset previous choice points
        V := VV(V);    VV1 := V1(VV)

        pc := address of next goal
        goto TOP


ONSUCCESS:

* reached if ``(goal)" succeeds in ``(!(goal))" goal, so redirect to
* failure

* reset choice point from the one stored in dummy environment
        VV := VV(dummy);  VV1 = V1(VV)
```

```
        goto DEEPFAIL

LISP:

* pass to LISP system the goal with each variable fully dereferenced

        P := expand(goal)

        if evaluate(P) = nil, goto DEEPFAIL

        pc := address of next goal
        goto TOP
```

Appendix 4

An Annotated Psuedocode for the Unification Macros of the PLM

For notation used in this appendix, refer to Appendix 3.

Additional notation:

1) We use the notation type(P) and pointer(P) to indicate the <type, pointer> fields of the value cell pointed to by P. (In the C source code, we use the terms ''class" and ''value" instead of ''type" and ''pointer".) The notation

<type, pointer>(P) := <T,U>

means type(P) := T; pointer(P) := U. Similar conventions will cause no confusion.

2) We use the term ''current" to refer to the term in the head that generated the macro.

3) We show molecules as <frame_ptr, source_ptr>. This is the order actually used on the stack. This order is consistent with the <type, pointer> order used for other constructs.

4) We use <constant, atom_ptr> to mean <ATOM, atom_ptr> or <NUMBER, number_ptr>.

5) The notation

<T,U> := dereference(frame, offset)

is an inline routine that performs the dereferencing operation (see Section 4.0) starting at the value cell pointed to by frame + offset. The result <type, pointer> is assigned to <T,U>.

6) The notation ''trail(U)" is an inline routine that checks whether the value cell pointed to by U is above (more senior than) the environment indicated by VV (or VV1, if U points to global stack). If so, U is pushed onto the trail.

7) We use the term ''succeed" to mean ''return", if the unification macro is viewed as a procedure; else it means ''goto next PLM instruction".

```
UVAR    Unify incoming term with the first occurrence of a variable
           that is a level 3 argument of the head.

        arg1: argument number for current variable
        arg2: the frame of value cell for current variable (LOCAL or
              GLOBAL)
        arg3: variable number for current variable (offset into frame of
              its value cell)

Example:
      ?z in head of ((A ?x (?x.?y) ?z) if (B ?x ?y ?z))
      would generate UVAR(2,LOCAL,0)

* set <T,U> to <type, pointer> of incoming argument (outer literal)
      <T,U> := <type, pointer>(A + arg1)

* possibilities for <T,U> are the outer literals:
*     <SKEL, skel_ptr>
*     <GLOBAL/LOCAL, offset>
*     <VOID, 0>
*     <constant, atom_ptr>

* dereference if necessary, using parent's frame (X or X1).
      if T = LOCAL, <T,U> := dereference(X,U)
      else if T = GLOBAL, <T,U> := dereference(X1,U)

* not documented in Warren:
      if T = VOID, <T,U> := <UNDEF, UNDEF>

* Incoming argument a molecule.  Note that U = skel_ptr already
* and X1 is the appropriate frame.
      if T = SKEL, T := X1

* possibilities for <T,U> at this point
*     <frame_ptr, skel_ptr>
*     <REF, cell_ptr>
*     <constant, atom_ptr>
*     <UNDEF, UNDEF>

* create the variable binding (trailing not necessary since in current
* environment, which will be discarded on failure)
```

```
      if arg2 = LOCAL, <type, pointer>(V + arg3 + 3) := <T,U>

* else, arg2 = GLOBAL
      else

   * do not assign a local reference to a global cell
        if T = REF and U points to local stack [

      * instead, reverse direction of the reference
           type(U) := REF;  pointer(U) := V1 + arg3
           <type, pointer>(V1 + arg3) := <UNDEF, UNDEF>
        ]

        else <type, pointer>(V1 + arg3) := <T,U>
```

```
UVAR1  Unify incoming term with the first occurrence of a variable
          that is a level 4 argument of a skeleton term.

       arg1: argument number for current variable (can only be 0 or 1)
       arg2: the frame of value cell for current variable (LOCAL or
             GLOBAL)
       arg3: variable number for current variable (offset into frame of
             its value cell)

Example:
      ?y in head of ((A ?x (?y.?x) ?z) if (B ?x ?y ?z))
      would generate UVAR1(0,GLOBAL,1)

* B points to skeleton literal of term unifying with (dereferenced
*    parent goal), Y points to its frame

* set <T,U> to <type,pointer> of incoming argument (inner literal)
      <T,U> := <type, pointer>(B + arg1)

* possibilities for <T,U> are the inner literals:
*    <SKEL, skel_ptr>
*    <GLOBAL, offset>
*    <constant, atom_ptr>

* dereference if necessary, using frame Y and offset U.
      if T = GLOBAL, <T,U> := dereference(Y,U)


* argument of skeleton is skeleton [for example, ((?x.?y).?z)]
   * set correct frame; B already points to skeleton literal
      else if T = SKEL, T := Y

* possibilities for <T,U> at this point (result of dereferencing)
*    <frame_ptr, skel_ptr>
*    <REF, cell_ptr>
*    <constant, atom_ptr>

* create the variable binding (trailing not necessary since in current
* environment, which will be discarded on failure)
* Note that arg2 can be local only if current variable is argument
* to a mode '+' skeleton (USKELD).
```

```
      if arg2 = LOCAL, <type, pointer>(V + arg3 + 3) := <T,U>

* This time, if T = REF, we know that U points into global stack
* (compare to similar situation for UVAR macro).

* else, arg2 = GLOBAL
      else
         <type, pointer>(V1 + arg3) := <T,U>
```

```
UATOM  Unify incoming term with an atom at level three.
(UNUM is identical, except uses num_ptr and type NUM.)

        arg1: argument number for current atom
        arg2: atom pointer for current atom

Example:
       'nil' in head of ((A ?x (?y.?x) nil) if (B ?x ?y ?z))
       would generate UATOM(2,atom_ptr) where PNAME(atom_ptr) is 'nil'

* set <T,U> to <type, pointer> of incoming argument (outer literal)
       <T,U> := <type, pointer>(A + arg1)

* possibilities for <T,U> are the outer literals:
*     <SKEL, skel_ptr>
*     <GLOBAL/LOCAL, offset>
*     <VOID, 0>
*     <constant, atom_ptr>

* dereference if necessary, using parent's frame (X or X1).
       if T = LOCAL, <T,U> := dereference(X,U)
       else if T = GLOBAL, <T,U> := dereference(X1,U)

* possibilities for <T,U> at this point
*     <frame_ptr, skel_ptr>
*     <REF, cell_ptr>
*     <constant, atom_ptr>

* if incoming term an atom (or bound to an atom), must be same as
* current atom
       if T = ATOM, [if arg2 != U, fail, else succeed]

* incoming term an ununified variable; create binding
       else if T = REF, <type, pointer>(U) := <ATOM, arg2>

* no action required for VOID incoming term (simply succeed)
* in all other cases, fail
      else if T != VOID, fail
```

```
UATOM1 Unify incoming term with an atom that is a level 4 argument
          of a skeleton term.
(UNUM1 is identical, except uses num_ptr and type NUM)

        arg1: argument number for current atom (can only be 0 or 1)
        arg2: atom pointer for current atom

Example:
       'nil' in head of ((A ?x (nil.?x) ?y) if (B ?x ?y ?z))
       would generate UATOM1(0,atom_ptr) where PNAME(atom_ptr) is 'nil'
       Note that ((A ?x (?x) ?y)) would be compiled as
          ((A ?x (?x.nil) ?y)) and would generate UATOM1(1,atom_ptr)

* B points to skeleton literal of term unifying with (dereferenced
*    parent goal), Y points to its frame

* set <T,U> to <type,pointer> of incoming argument (inner literal)
       <T,U> := <type, pointer>(B + arg1)

* possibilities for <T,U> are the inner literals:
*    <SKEL, skel_ptr>
*    <GLOBAL, offset>
*    <constant, atom_ptr>

* dereference if necessary, using frame Y and offset U.
       if T = GLOBAL, <T,U> := dereference(Y,U)

* possibilities for <T,U> at this point
*    <frame_ptr, skel_ptr>
*    <SKEL, skel_ptr>
*    <REF, cell_ptr>
*    <constant, atom_ptr>

* if incoming term an atom (or bound to an atom), must be same as
* current atom
       if T = ATOM, [if arg2 != U, fail, else succeed]

* incoming term an ununified variable; create binding;
* trail if necessary
       else if T = REF, <type, pointer>(U) := <ATOM, arg2>; trail(U)
```

```
* for all other cases, fail
      else, fail
```

```
UREF    Unify incoming term with a variable that is not a first
           occurrence which is a level 3 argument of the head.

        arg1: argument number for current variable
        arg2: the frame of value cell for current variable (LOCAL or
              GLOBAL)
        arg3: variable number for current variable (offset into frame of
              its value cell)

Example:
      the second ?z in head of ((A ?x (?z.?x) ?z) if (B ?x ?y ?z))
      would generate UREF(2,GLOBAL,1)

* set <T,U> to <type, pointer> of incoming argument (outer literal)
      <T,U> := <type, pointer>(A + arg1)

* possibilities for <T,U> are the outer literals:
*     <SKEL, skel_ptr>
*     <GLOBAL/LOCAL, offset>
*     <VOID, 0>
*     <constant, atom_ptr>

* if T = VOID, we are done (always succeeds)
      if T = VOID, succeed

* first, fully dereference incoming argument <T,U>
      if T = LOCAL, <T,U> := dereference(X,U)
      else if T = GLOBAL, <T,U> := dereference(X1,U)

* Incoming argument a molecule.  Note that U = skel_ptr already
* and X1 is the appropriate frame.
      if T = SKEL, T := X1

* set up <type, pointer> for current variable
      T1 := arg2;  U1 := arg3
      if T1 = LOCAL, <T1,U1> := dereference(V,U1)
      else  <T1,U1> := dereference(V1,U1)       * T1 = GLOBAL

* possibilities for <T,U>  and <T1,U1> at this point
*     <frame_ptr, skel_ptr>
*     <REF, cell_ptr>
```

```
*     <constant, atom_ptr>

* now that both terms are dereferenced, use the common ''ref" routine
      goto ref
```

```
UREF1  Unify incoming term with a variable that is not a first
          occurrence which is a level 3 argument of the head.

       arg1: argument number for current variable
       arg2: the frame of value cell for current variable (LOCAL or
             GLOBAL)
       arg3: variable number for current variable (offset into frame of
             its value cell)

Example:
      the second ?x in head of ((A ?x (?z.?x) ?z) if (B ?x ?y ?z))
      would generate UREF1(1,GLOBAL,0)

* B points to skeleton literal of term unifying with (dereferenced
*    parent goal), Y points to its frame

* set <T,U> to <type,pointer> of incoming argument (inner literal)
     <T,U> := <type, pointer>(B + arg1)

* possibilities for <T,U> are the inner literals:
*    <SKEL, skel_ptr>
*    <GLOBAL, offset>
*    <constant, atom_ptr>

* first, fully dereference incoming argument <T,U>
* Note that T = LOCAL or VOID is not possible
     if T = GLOBAL, <T,U> := dereference(Y,U)

* argument of skeleton is skeleton [for example, ((?x.?y).?z)]
   * set correct frame; B already points to skeleton literal
     else if T = SKEL, T := Y

* set up <type, pointer> for current variable
     T1 := arg2;  U1 := arg3
     if T1 = LOCAL, <T1,U1> := dereference(V,U1)
     else  <T1,U1> := dereference(V1,U1)       * T1 = GLOBAL

* possibilities for <T,U>  and <T1,U1> at this point
*    <frame_ptr, skel_ptr>
*    <REF, cell_ptr>
*    <constant, atom_ptr>
```

```
* now that both terms are dereferenced, use the common ''ref" routine
      goto ref
```

```
REF    The common code between UREF and UREF1 (not in Warren).
(We ignore the type NUMBER here, since extending the code to include
that case is trivial.)

       Unify <T,U> and <T1,U1>

* if both are ATOMs ...
      if T and T1 = ATOM, [if U != U1, fail, else succeed]

      else if T = REF  [            * then there are 2 possibilities

   * 1) T1 is a molecule or atom; create binding, and trail if necessary
   * Note how we decide if <T1,U1> is a molecule
        if (T1 > REF or T1 = ATOM),
           <type, pointer>(U) := <T1,U1>;  trail(U)

   * 2) T1 is also a reference
   * Here, we assign the more senior cell to the more junior

        if U and U1 point to same stack
           if U < U1, <type, pointer>(U1) := <REF,U>;  trail(U1)

        else if U points to global and U1 to local stack
           <type, pointer>(U1) := <REF,U>;  trail(U1)

        else                              * U1 global, U local
           <type, pointer>(U) := <REF,U1>;  trail(U)
     ]

* else if T1 = REF, then T must be molecule or atom; create binding
      else if T1 = REF
         <type, pointer>(U1) := <T,U>;  trail(U1)

* else if both are molecules, must call general unification routine
      else if T and T1 > REF, [if unify(T,U,T1,U1), succeed, else fail]

* else, one is a molecule and other is atom -- fail
      else fail
```

```
USKEL  Unify incoming term with a skeleton that is a level 3 argument
          of the head (mode '?', or no declaration).

       arg1: argument number for current skeleton term
       arg2: pointer to skeleton literal for current skeleton

Example:
     (?x.?y) in head of ((A ?x (?x.?y) ?z) if (B ?x ?y ?z))
     would generate USKEL(1,skel_ptr) where skel_ptr points to
     skeleton literal generated for (?x.?y)

Note:  In order skip the argument instructions of the skeleton
       (for example, if the incoming argument is a reference construct)
       we set Y := 0, which will cause IFDONE to skip around them.

* set <Y,B> to <type, pointer> of incoming argument (outer literal)
     <Y,B> := <type, pointer>(A + arg1)

* possibilities for <Y,B> are the outer literals:
*    <SKEL, skel_ptr>
*    <GLOBAL/LOCAL, offset>
*    <VOID, 0>
*    <constant, atom_ptr>

* fail immediately if incoming term an atom
     if Y = ATOM, fail

* This case is not in Warren:
* Incoming argument a molecule.  Note that B = skel_ptr already
* and X1 is the appropriate frame.
     if Y = SKEL, Y := X1; succeed

* not documented in Warren:
* Y := 0 will skip argument instructions
     if Y = VOID, Y := 0; succeed

* dereference if necessary, using parent's frame (X or X1).
     else if Y = LOCAL, <Y,B> := dereference(X,B)
     else if Y = GLOBAL, <Y,B> := dereference(X1,B)

* possibilities for <Y,B> at this point
```

```
*     <frame_ptr, skel_ptr>
*     <REF, cell_ptr>
*     <constant, atom_ptr>

* fail immediately if incoming term dereferenced to an atom
       if Y = ATOM, fail

* if incoming term is ununified variable, bind it to a molecule,
* and we are done, so set Y := 0; trail if necessary
       if Y = REF, Y := 0;  <type, pointer>(B) := <V1,arg2>; trail(B)

* if Y > REF (incoming argument a molecule), Y already has address
* of frame and B has address of skeleton literal, so do nothing
```

```
USKEL1 Unify incoming term with a skeleton that is a level 4 argument
        of a skeleton term (mode '?' or '+').

       arg1: argument number for current skeleton term
       arg2: pointer to skeleton literal for current skeleton

Example:
       (?x.?y) in head of ((A ?x ((?x.?y).?z) ?z) if (B ?x ?y ?z))
       would generate USKEL(0,skel_ptr) where skel_ptr points to
       skeleton literal generated for (?x.?y)

 * B points to skeleton literal of term unifying with (dereferenced
 *    parent goal), Y points to its frame

 * set <T,U> to <type,pointer> of incoming argument (inner literal)
       <T,U> := <type, pointer>(B + arg1)

 * possibilities for <T,U> are the inner literals:
 *    <SKEL, skel_ptr>
 *    <GLOBAL, offset>
 *    <constant, atom_ptr>

 * dereference if necessary, using frame Y and offset U.
 * T = VOID or LOCAL not possible
       if T = GLOBAL, <T,U> := dereference(Y,U)

 * if incoming argument is another skeleton, will have to call
 * general unification routine to unify <Y,U> and <V1,arg2>
       if T = SKEL, [if unify(Y,U,V1,arg2) succeed, else fail]

 * possibilities for <T,U> at this point (result of dereferencing)
 *    <frame_ptr, skel_ptr>
 *    <REF, cell_ptr>
 *    <constant, atom_ptr>

 * as before, fail if atom
       if T = ATOM, fail

 * if T = REF, create molecule (V1,arg2), and trail if necessary
       if T = REF, <type, pointer>(U) := <V1,arg2>; trail(U)
```

```
* if incoming argument dereferenced to a molecule, unify ...
      if T > REF, [if unify(T,U,V1,arg2) succeed, else fail]
```

```
USKELD Unify incoming term with a skeleton that is a level 3 argument
        of the head (mode '+').

       arg1: argument number for current skeleton term

Example:
       (?x.?y) in head of ((A ?x (?x.?y) ?z) if (B ?x ?y ?z))
       with mode declaration (mode (A ? + ?)) would generate USKELD(1).

 * set <Y,B> to <type, pointer> of incoming argument (outer literal)
       <Y,B> := <type, pointer>(A + arg1)

 * possibilities for <Y,B> are the outer literals:
 *     <SKEL, skel_ptr>
 *     <GLOBAL/LOCAL, offset>
 *     <VOID, 0>
 *     <constant, atom_ptr>

  * fail immediately if incoming term an atom
        if Y = ATOM, fail

  * This case is not described in Warren:
  * Incoming argument a molecule.  Note that B = skel_ptr already
  * and X1 is the appropriate frame.
        if Y = SKEL, Y := X1; succeed

  * not documented in Warren:
        if Y = VOID, mode error; fail

  * dereference if necessary, using parent's frame (X or X1).
        else if Y = LOCAL, <Y,B> := dereference(X,B)
        else if Y = GLOBAL, <Y,B> := dereference(X1,B)

  * possibilities for <Y,B> at this point
  *     <frame_ptr, skel_ptr>
  *     <REF, cell_ptr>
  *     <constant, atom_ptr>

  * fail immediately if incoming term dereferenced to an atom
        if Y = ATOM, fail
```

```
* if incoming term is ununified variable, mode declaration error
      if Y = VOID, mode error; fail

* if Y > REF (incoming argument a molecule), Y already has address
* of frame and B has address of skeleton literal, so do nothing
```

```
USKELC Unify incoming term with a skeleton that is a level 3 argument
          of the head (mode '-').

        arg1: argument number for current skeleton term
        arg2: pointer to skeleton literal for current skeleton

Example:
      (?x.?y) in head of ((A ?x (?x.?y) ?z) if (B ?x ?y ?z))
      with mode declaration (mode (A ? - ?)) would generate
      USKELC(1,skel_ptr) where skel_ptr points to skeleton literal
      generated for (?x.?y).

* set <Y,B> to <type, pointer> of incoming argument (outer literal)
      <Y,B> := <type, pointer>(A + arg1)

* possibilities for <Y,B> are the outer literals:
*     <SKEL, skel_ptr>
*     <GLOBAL/LOCAL, offset>
*     <VOID, 0>
*     <constant, atom_ptr>

* fail immediately if incoming term an atom
      if Y = ATOM, fail

* This case is not in Warren:
* Incoming argument a molecule.  Note that B = skel_ptr already
* and X1 is the appropriate frame.
      if Y = SKEL, Y := X1; succeed

* not documented in Warren:
* if Y = VOID, do nothing
      if Y = VOID, succeed

* dereference if necessary, using parent's frame (X or X1).
      else if Y = LOCAL, <Y,B> := dereference(X,B)
      else if Y = GLOBAL, <Y,B> := dereference(X1,B)

* possibilities for <Y,B> at this point
*     <frame_ptr, skel_ptr>
*     <REF, cell_ptr>
*     <constant, atom_ptr>
```

```
* if incoming term is ununified variable, bind it to a molecule,
* trail if necessary
      if Y = REF, <type, pointer>(B) := <V1,arg2>; trail(B)

* in all other cases, mode error
      else mode error; fail
```

```
UNIFY  Unify current argument with incoming argument in the case
          they are both molecules.

       arguments: (F1, S1, F2, S2)
       where <F1,S1> represents the <frame, skel_ptr> molecule for
       one of the arguments (it does not matter which), and <F2,S2>
       represents the molecule for the other.

       This is set up as a function (here and in the actual code),
       returning TRUE or FALSE (success or failure).
```

Note: The following psuedocode is meant to explain the function of the general unification routine. It does not reflect the exact processing used in the code, since the clearest psuedocode, it developed, was not the best coded implementation. The logic described in the next paragraph, however, is followed in the actual code.

We conceive of this routine as matching two binary trees (each argument is represented by a binary tree). The top do-while loop searches down the left children of a node, pushing the right children onto a stack as it goes, until failure or until it reaches a leaf on one or both trees. When it does, we unify the leaves and pop the stack, replacing the current pair of nodes with the top of the stack. These nodes' left children have already been processed, so we unify the nodes' right children (cdr's). If they are both molecules themselves, we loop to the top again to begin the search down their left children. Otherwise, we pop the stack again, and continue.

```
loop forever [

do [

1. push cdr's and frames: push(F1,S1+1,F2,S2+1)

2. if car of S1 = GLOBAL, <F1,S1> := dereference(F1,pointer(S1))
   if car of S1 = GLOBAL, <F1,S1> := dereference(F2,pointer(S2))

3. if car of S1 is ATOM
   a) if car of S2 is ATOM or REF, break out of loop
   b) if car of S2 is MOLECULE, return (FALSE)

4. else if car of S2 is ATOM
   a) if car of S1 is REF (can not be ATOM), break out of loop
   b) if car of S1 is MOLECULE, return (FALSE)
```

```
* ''car of S1 is not a leaf" means type(S1) = SKEL or F1 > REF
] while (neither the car of S1 nor the car of S2 is a leaf,
         S1 := car(S1); S2 := car(S2)),  loop


* possibilities at this point:
*
*     MOLECULE for only ONE argument
*     REF      for one or both
*     ATOM     for one or both

do [

1. if both are atoms, return (FALSE) unless they are the same

2. if car of S1 is an atom or a molecule (F2 = REF)
   assign it to the cell S1

3. if car of S2 is an atom or a molecule (F1 = REF)
   assign it to the cell S2

4. if both are references
   assign the cell of the more senior to the cell of the more junior

5. if stack is empty, return (TRUE)

6. pop(F1,S1,F2,S2)

* process cdr's:

1. if cdr of S1 = GLOBAL, <F1,S1> := dereference(F1,pointer(S1))
   if cdr of S1 = GLOBAL, <F1,S1> := dereference(F2,pointer(S2))

2. if cdr of S1 is ATOM
   a) if cdr of S2 is ATOM or REF, continue with loop
   b) if cdr of S2 is MOLECULE, return (FALSE)

3. else if cdr of S2 is ATOM
   a) if cdr of S1 is REF (can not be ATOM), continue with loop
   b) if cdr of S1 is MOLECULE, return (FALSE)
```

```
* ''cdr of S1 is not a leaf" means type(S1) = SKEL or F1 > REF
] while (either the cdr of S1 or the cdr of S2 is a leaf), loop

S1 := cdr(S1); S2 := cdr(S2)

] loop
```

Appendix 5

Example Developmental Tracings for the Reverse Procedure

This appendix shows several tracings for a run of the reverse procedure defined below. All the tracings represent the same run. The listings have been modified for the sake of clarity in places. Mainly, this meant replacing a cryptic number with more useful information or adding explanatory comments.
Listing 1: Source program:

```
;program:
                (assert
                ((reverse nil ?x ?x))
                ((reverse (?x.?y) ?z ?ans) if (rev ?y (?x.?z) ?ans))
                )

;user query:    ((reverse (a b) nil ?ans))

;user query as seen by PLM:

;               ((:GO) if (reverse (a b) nil ?ans)
;                         (print (reverse (a b) nil ?ans) )
;                         (fail))
;
;Note that all answers will be found since (fail) is the last goal of
;the user query.
```

Listing 2: Contents of the item list, symbol table, and expand arrays:

```
CountL = 0
CountG = 0
                        Name Type       Var_no      Occur

       183:               ?x  Temporary  0           many

contents of item_list follows

    Index     Atom  Type        Type        Arg_no        Level        Occur

0:          reverse  HEAD        8           0             3            3
1:              nil  NIL         5           0             3
2:    183        ?x  VAR         2           1             3            1
3:    183        ?x  VAR         2           2             3            2
4:                   NECKFT      23          0             2

CountL = 1
CountG = 3
                        Name Type       Var_no      Occur

       183:               ?x  Global     0           many
       184:               ?y  Global     1           many
       185:               ?z  Global     2           many
       385:             ?ans  Local      0           many

contents of item_list follows

    Index     Atom  Type        Type        Arg_no        Level        Occur

 0:         reverse  HEAD        8           0             3            3
 1:                  USKEL       30          0             3
 2:    183        ?x  VAR         2           0             4            1
 3:                   DOT         28          2             4
 4:    184        ?y  VAR         2           1             4            1
 5:    185        ?z  VAR         2           1             3            1
 6:    385      ?ans  VAR         2           2             3            1
 7:                   NECK        22          0             2
 8:         reverse  GOAL        6           2             3            3
```

```
 9:    184        ?y   VAR        2        0        3        2
10:                     SKEL      29        1        3
11:    183        ?x   VAR        2        0        4        2
12:                     DOT       28        0        4
13:    185        ?z   VAR        2        1        4        2
14:    385      ?ans   VAR        2        2        3        2
15:                     FOOT      26        0        3

enter query...

CountL = 1
CountG = 0
                          Name Type       Var_no      Occur

       385:               ?ans  Local        0        many

contents of item_list follows

     Index      Atom  Type        Type        Arg_no      Level      Occur

 0:                :GO  HEAD         8         0          3          0
 1:                     NECK        22         0          2
 2:           reverse   GOAL         6         2          3          3
 3:                     SKEL        29         0          3
 4:                 a   ATOM         3         0          4
 5:                     DOT         28         1          4
 6:                     NESTED      34         1          4
 7:                 b   ATOM         3         0          5
 8:               nil   NIL          5         1          3
 9:    385        ?ans  VAR          2         2          3          1
10:             print   LISP         7         3          3          0
11:              fail   FAIL        11         3          3
12:                     FOOT        26         0          3

contents of expand array follows

                Index      Atom  Type        Var_no  Occur

                   0:             print  ATOM
                   1:                    (
                   2:           reverse  ATOM
```

```
 3:                    (
 4:                  a  ATOM
 5:                  b  ATOM
 6:                     )
 7:                     NIL
 8:   385       ?ans    LOCAL          0        1
 9:                     )
10:                     )
```

Listing 3: Contents of the code array:

```
  Index     Macro                Arg1              Arg2              Arg3

    0:      FAIL
    1:      TRYLAST
    2:      DEEPFAIL
    3:      FAILEXIT
    4:      CALL                 :GO
    8:      SUCCESSEXIT

code for ((reverse nil ?x ?x)) :

    9:      TRYLAST              0                 190648
   12:      UATOM                0                    nil
   15:      UVAR                 1                 LOCAL             0
   19:      UREF                 2                 LOCAL             0
   23:      NECKFOOTO            3

code for ((reverse (?x.?y)  ?z ?ans )  if
       (reverse  ?y  (?x.?z)  ?ans)) :

   25:      TRYLAST              0                 190768
   28:      USKEL                0                     72            Op
   32:      INIT                 0                      1
   35:      IFDONE              45
   37:      UVAR1                0                 GLOBAL            0
   41:      UVAR1                1                 GLOBAL            1
   45:      UVAR                 1                 GLOBAL            2
   49:      UVAR                 2                  LOCAL            0
   53:      NECK                 1                      3
   56:      DEPART           reverse3                   0            3
   60:      GLOBAL               1
   62:      SKEL                68
   64:      LOCAL                0
   66:      FOOT                 3

skeleton literal (?x.?z) :

   68:      GLOBAL               0                 GLOBAL                2
```

```
skeleton literal (?x.?y) :

   72:      GLOBAL                  0            GLOBAL                  1

Code for  ((:GO) if  (reverse (a b) nil ?ans)
     (print  (reverse  (a b) nil ?ans) )
*****        (more_ans))


   76:      LOCALINIT               0            0
   79:      NECKO                   1
   81:      CALL              reverse3           0                       3
   85:      SKEL                   97
   87:      ATOM                  nil
   89:      LOCAL                   0
   91:      LISP                print            0
   94:      MORE ANS
   95:      FOOT                    0

Skeleton Literal (a b) :

   97:      ATOM                    a            SKEL                  101
  101:      ATOM                    b            ATOM                  nil
```

NOTE (*****).

The last goal in the clause for :GO will be either (more_ans) or (fail) depending on whether the qflag is set or not. The qflag tells the compiler to automatically generate all answers to a query. If it is not set (that is the default), then the goal (more_ans) will, at run time, query the user whether he desires more answers (if any).

Listing 4: Trace of PLM registers:

```
bottom of LOCAL, GLOBAL, TR stacks are 2178052 2186052 2194048
   and of code array is 137828

AFTER execution of the macro to the left, the new contents
of the PLM register is shown.  Note that not every executed
macro is listed; see Listing 6 for this.

after:     A     V     V1    X     X1    VV    VV1   TR    TRho pc   FL

init             1     1001  1     1001
===> call: :GO
call       4                                 1     1001        48   33   367632
localinit                                                            36
neck0            5           1                                       37
===> call: reverse
call       38                                5     1001        48   6    367648
fail                                                   48            13   365584
trylast                                      1     1001              14
init                                                                 17
ifdone                                                               18
uvar1                                                                19
uvar1                                                                20
uvar                                                   48            21
uvar                                                   48            22
neck             9     1004  5     1001                              25
===> call: reverse
call       26                                9     1004        48   6    367648
fail                                                   48            13   365584
trylast                                      1     1001              14
uskel                                                  48            16
init                                                                 17
ifdone                                                               18
uvar1                                                                19
uvar1                                                                20
uvar                                                   48            21
uvar                                                   48            22
neck             13    1007  9     1004                              25
===> call: reverse
```

```
call       26                          13    1007        48    6    367648
uatom                                        48          48    8
uvar                                         48          48    9
uref                                         44          44    10
neckfoot0        16                                            29
foot       26    16          5     1001                       29
foot       38    16          1     1001                       41
first answer: (reverse (a b) nil (b a))
deepfail         13    1007                                    367648
deepfail   26    13    1007  9     1004          48    48      13    365584
trylast                                1     1001              14
deepfail         1     1001                                    367632
deepfail   4     1     1001  1     1001          48    48      2     365584
nil
```

Listing 5: Trace of the local and global stack contents:

Note: Management information areas are identified by the labels for the fields at the far left. The information in the FL and TR fields may not be useful, but is correct for this run. The stack contents is printed in the order in which it was defined during the run. For skeleton literal references (for example, ''s0"), see Listing 3.

```
                LOCAL STACK                    GLOBAL STACK

===> call: :GO
stack address                    stack address
          |                            |
          |                            |
          V   type       pointer       |
                                       |
VV,FL     1|         1|    367632|     |
 X, A     2|         1|         4|     |
V1,TR     3|      1001|        48|     |
          4|  UNDEF   |  UNDEF   |     |
===> call: reverse                     |
                                       |
VV,FL     5|         1|    367648|     |
 X, A     6|         1|        38|   V     type       pointer
V1,TR     7|      1001|        48|
                                    1001|  UNDEF    |  UNDEF    |
                                    1002|  UNDEF    |  UNDEF    |
                                    1001|  ATOM     |         a |
                                    1002|      1001 |        s3 |
                                    1003|  ATOM     |       nil |
          8|  REF     |         4|
===> call: reverse

VV,FL     9|         1|    367648|
 X, A    10|         5|        26|
V1,TR    11|      1004|        48|
                                    1004|  UNDEF    |  UNDEF    |
                                    1005|  UNDEF    |  UNDEF    |
                                    1004|  ATOM     |         b |
                                    1005|  ATOM     |       nil |
```

```
                                    1006|       1001|          s0|
        12|  REF      |          4|
===> call: reverse

VV,FL   13|          1|     367648|
 X, A   14|          9|         26|
V1,TR   15|       1007|         48|
        16|       1004|         s0|
         4|       1004|         s0|
first answer: (reverse (a b) nil (b a))
         4|  UNDEF    |  UNDEF    |
nil
```

Listing 6: Trace of PLM macro routine entry:

```
begin execution ...
pc is 137876 offset is 3, call
===> call: :GO
pc is 138356 offset is 33, checkarity
pc is 138372 offset is 34, no_op
pc is 138388 offset is 35, localinit
pc is 138404 offset is 36, neck0
pc is 138420 offset is 37, call
===> call: reverse
pc is 137924 offset is 6, checkarity
pc is 137940 offset is 7, uatom
entered fail, V & VV are 2178084 2178084
pc is 138036 offset is 13, trylast
pc is 138052 offset is 14, checkarity
pc is 138068 offset is 15, uskel
pc is 138084 offset is 16, init
pc is 138100 offset is 17, ifdone
pc is 138116 offset is 18, uvar1
pc is 138132 offset is 19, uvar1
pc is 138148 offset is 20, uvar
pc is 138164 offset is 21, uvar
pc is 138180 offset is 22, no_op
pc is 138196 offset is 23, no_op
pc is 138212 offset is 24, neck
pc is 138228 offset is 25, call
===> call: reverse
pc is 137924 offset is 6, checkarity
pc is 137940 offset is 7, uatom
entered fail, V & VV are 2178116 2178116
pc is 138036 offset is 13, trylast
pc is 138052 offset is 14, checkarity
pc is 138068 offset is 15, uskel
pc is 138084 offset is 16, init
pc is 138100 offset is 17, ifdone
pc is 138116 offset is 18, uvar1
pc is 138132 offset is 19, uvar1
pc is 138148 offset is 20, uvar
pc is 138164 offset is 21, uvar
```

```
pc is 138180 offset is 22, no_op
pc is 138196 offset is 23, no_op
pc is 138212 offset is 24, neck
pc is 138228 offset is 25, call
===> call: reverse
pc is 137924 offset is 6, checkarity
pc is 137940 offset is 7, uatom
pc is 137956 offset is 8, uvar
pc is 137972 offset is 9, uref
common routine
pc is 137988 offset is 10, no_op
pc is 138004 offset is 11, no_op
pc is 138020 offset is 12, neckfoot0
pc is 138292 offset is 29, foot
pc is 138292 offset is 29, foot
pc is 138484 offset is 41, lisp
first answer:  (reverse (a b) nil (b a))
pc is 138500 offset is 42, deepfail
entered deepfail
pc is 138036 offset is 13, trylast
pc is 138052 offset is 14, checkarity
pc is 138068 offset is 15, uskel
entered fail, V & VV are 2178148 2178052
entered deepfail
pc is 137860 offset is 2, failexit
nil
```

Appendix 6

Illustrations of the global versus local problem. See text, Section 4.3.

This appendix illustrates differences between the global and local variables, using a series of figures. Figure 13 shows how unifications are reflected in a local variable's originating environment. The next two figures are particularly revealing, in that they compare two almost identical programs. Figure 15 uses global anonymous variables, while Figure 14 uses local voids in the same context. Note in particular how the cell values for the global variables of Figure 15 play exactly the same role as the local variables $?x$ and $?y$ play in Figure 14. This in itself is compelling reason for providing global stack space for global, even if "void", variables.

Figure 16 illustrates how a forward reference can develop in a molecule. After reviewing this figure, it may be believed that anonymous global variables in clause a and clause b still did not require space on the global stack. The reasoning might be that if the terms (?) of clause "a" and (?) of clause "b" were replace by ? in both clauses, essentially the same proof (execution) would go through but without the need for the global stack. This is true, except for one point: there is a difference between local variable $?x$ binding to the term (?) and to the term ?. The latter case gives no information about $?x$ and its cell value would remain "undef". The former case, however, gives a structure to $?x$ that it did not have before, and its cell value becomes a molecule as in the above example. Consider the following:

```
TRY succeeds:

((TRY) if (a ?x) (b ?x))
((a ?))
((b A))
```

```
TRY fails:

((TRY) if (a ?x) (b ?x))
((a (?) ))
((b A))
```

```
clause C:    (( ... ) if (G ?x)  (H ?x))

clause D:    ((G ?y) if (J ?y))

clause E:    ((J ?z) if (K ?z))

clause K:    ((K A))
```

appearance of local stack just before invocation of K:

environment of clause C	?x-address	UNDEF	UNDEF	
environment of clause D	?y	REF	?x-address	
environment of clause E	?z	REF	?x-address	

appearance of local stack just before invocation of H:

environment of clause C	?x-address	ATOM	''A"	
environment of clause D	?y	REF	?x-address	
environment of clause E	?z	REF	?x-address	

This figure illustrates an example described in the text, Section 4.3.

Figure 13: Local Variable Unification

```
Source Code:

      (assert
      ((u) if (a ?x) (b ?y) (c ?x ?y) (d ?x) (e ?y))

      ((a ?))
      ((b ?))
      ((c ?w ?w))
      ((d A))
      ((e B))
      )

Query:
      ((u))

Before the call to the first goal of clause u:
  ?x   7|  UNDEF   |  UNDEF   |  cell values of
  ?y   8|  UNDEF   |  UNDEF   |  environment for u

(The call to a and to b do not create any new cell values.)

After call c:
  ?w  12|   REF    |    7     |  in environment for c
  ?y   8|   REF    |    7     |  in environment for u

After call d:
  ?x   7|   ATOM   |    A     |  in environment for c

After call to e:
  nil
```

This and the following figure illustrate the difference between global and local variables.

Figure 14: Local Versus Global Variable Unification

```
source code:

      (assert
      ((u) if (a ?x) (b ?y) (c ?x ?y) (d ?x) (e ?y))

      ((a (7) ))
      ((b (7) ))
      ((c ?w ?w))
      ((d (A) ))
      ((e (B) ))
      )

Skeleton  S1:  GLOBAL   0    ATOM     nil     [for (?) in clause a]
Literals: S2:  GLOBAL   0    ATOM     nil     [for (?) in clause b]
          S3:  ATOM     A    ATOM     nil     [for (A) in clause d]
          S4:  ATOM     B    ATOM     nil     [for (B) in clause e]

              LOCAL STACK                    GLOBAL STACK

Before call to first goal of clause u:
  ?x   7|  UNDEF   |  UNDEF   |  cell values of
  ?y   8|  UNDEF   |  UNDEF   |  environment for u


After the call to a:

       in environment for u:
  ?x   7|   1001   |    S1    |
                                        in environment for a:
                                  ?  1001|  UNDEF   |   UNDEF  |


After the call to b:

       in environment for u:
  ?y   8|   1002   |    S1    |
                                        in environment for b:
                                  ?  1002|  UNDEF   |   UNDEF  |

After the call to c:

       in environment for c:
  ?w  12|   1001   |    S1    |
                                        in environment for b:
                                  ?  1002|   REF    |   1001   |
After the call to d:

                                        in environment for a:
                                  ?  1001|   ATOM   |    A     |
After the call to e:
  nil
```

Figure 15: Local Versus Global Variable Unification

```
Source Code:
      (assert
      ((:GO) if (u (?x) ?y) (print (u (?x) ?y)) (fail) )
      ((u (?x) ?y) if (a ?x) (b ?y) (c ?x ?y) (d ?x))
      ((a (p ?z) ))
      ((b (p ?z) ))
      ((c ?w ?w))
      ((d (p q) ))
      )
Query:
      ((:GO))

Skeleton     s10:  ATOM      p       SKEL     s11
Literals:    s11:  GLOBAL    0       ATOM     nil
             s20:  ATOM      p       SKEL     s21
             s21:  GLOBAL    0       ATOM     nil

               LOCAL STACK                GLOBAL STACK
====> call: :GO
       Vi field = 1001
                                    1001|  UNDEF   |  UNDEF   |
        4|  UNDEF   |  UNDEF   |

====> call: u
       Vi field = 1002
                                    1002|  UNDEF   |  UNDEF   |
                                    1002|  REF     |      1001|
        8|  REF     |         4|

====> call: a
       Vi field = 1003

            This molecule ----->    1001|      1003|      s10 |
contains a forward reference.       1003|  UNDEF   |  UNDEF   |

====> call: b
       Vi field = 1004
        4|      1004|      s20 |
                                    1004|  UNDEF   |  UNDEF   |
====> call: c
       Vi field = 1005
       12|      1003|      s10 |
                                    1004|  REF     |      1003|
====> call: d
       Vi field = 1005
                                    1003|  ATOM    |         q|
(u ((p q)) (p q))
nil
```

This figure illustrates the possibility that a forward reference may occur in a molecule. Thus a dangling reference could be created if the global stack were popped on determinate procedure exit.

Figure 16: Forward Reference in Molecule

program and query. Getlist returns an s-expression stored as linked cons cells.

4) The program and query are asserted (see assertx).

Routines: getquery, getlist, assertx, getstack, run

GETQUERY

1) The user query is constructed here. It has the form

```
(assert ((:GO) if userqueries (print userqueries) (fail)))
```

where 'userqueries' has the form

```
(g1 ...) (g2 ...) ... (gn ...)   ; n >= 1
```

The user enters 'userqueries' enclosed in parentheses.

Note that all answers the Prolog program is able to find will be printed (since the goal 'fail' is placed in the clause). This could easily be changed to give control over this feature to the user.

Routines: none

GETSTACK

The only purpose is to allocate the run-time stack space. It is passed a parameter giving the size of the array to allocate. Thus, the size could easily be established at the top level by the user before a run.

Routines: none

ASSERTX

1) Processes the mode declarations of a program. Thus, the vector of modes for each clause is entered into the mode array. Also, the dtype of the procedure atoms is changed to reflect the fact that there is an associated mode declaration. (See setmodeptr.)

2) The main processing loop for clauses resides here. The routine nextatm returns the next atom in the source and its level. It also decides when the end of the clause or program has been reached.

```
process mode declarations
```

Appendix 7

Compiler Routines

This appendix describes the main functions of the most important routines in the compiler. Since the compiler is so rapidly evolving, some of the highest level routines may have changed by the time this is read. Certainly main, for example, will no longer be used to enable compiler development traces. Also, as a user interface is built, some high level control may be altered. The aspects of the compiler that implement Warren's ideas, however, should remain stable for some time, and it is here that this appendix should prove most helpful.

Note that the function of a routine is implied, in part, by the subroutines it calls. Each routine is described by giving a list of tasks actually performed in that routine, plus a list of the routines it calls. See Figure 14, page 101, for a diagram representing the calling structure of the compiler.

MAIN

1)The routine main queries the user to establish the kind of run desired. Currently the compiler can be run with any of thirteen developmental traces enabled. Appendix 5 shows examples of the output from some of these. Main is the entry point for the compiler.

Routines: globinit (initializes the LISP system), gopro

GOPRO

1) The code array is initialized with the five macros that begin all PLM programs: TRYLAST, DEEPFAIL, FAILEXIT, CALL, and SUCCESSEXIT. The arguments to the CALL are established: this is the call to the routine ":GO", a unique head-predicate name that establishes a clause for the user-query goal (see GETQUERY). The rule field of all goal-predicate atoms is initialized to point to this TRYLAST. Thus, in the event the user fails to provide any clauses for a procedure, this technique will ensure that its invocation will fail.

2) The cdr field on the clause cell for :GO is set to a clause cell for the FAILEXIT macro (in the third slot of the code array– see Figure 7, page 31). This will cause the correct action to take place on failure of the user query (or when all answers have been found).

3) The user is prompted to enter the program, and then the query. A routine in the storage management system provided by LISP, getlist, is used to read in his

```
do [     *loop once for each clause
   do [   *loop once for each atom

      if (atom is a built-in goal)
           call BUILTIN;  level := NEXTATMX

      else call CLASSIFY; level := NEXTATM

   ] while not done with clause

   ASSIGNVAR
   ATTACH
   GENMACRO
   CLR

] while not done with program
```

Routines: nextatm, nextatmx, setmodeptr, getmodeptr, builtin, classify, assignvar, attach, clr.

NEXTATM

1) Finds the next atom (number, variable, etc.) in the source, scanning from left to right.
2) Records argument number for all atoms and lists.
3) Finds and records arities of predicates.
4) Recognizes the end of a clause and the end of the program.
Routines: none

NEXTATMX

This routine is called when processing the arguments in a built-in function (and expand array is being filled). It is similar to nextatm, but does not need its full generality.

1) Finds the next atom (number, variable, etc.) in the source, scanning 2) Enters the type '(' and ')' into expand array. 3) Recognizes the end of a clause and the end of the program.

Routines: none

SETMODEPTR

This routine is called for new mode declarations.

1) Attach a mode cons cell ("mode cell") to the rule field of procedure's atom. The car field is set to the index of the mode array that stores the vector of modes declared for this procedure. The cdr field is temporarily set to nil (since during mode declaration processing, no clauses have been seen). The dtype for this procedure atom is set to DTPRED. (See Figure 8, page 32, for a diagram of clause attachment with an atom of dtype DTPRED.)

Routines: none

GETMODEPTR

This routine is called to find the mode array index for a particular procedure.

1) Return the mode array index from the mode cell associated with this procedure. If this procedure atom does not have a dtype of DTPRED, return zero. (Note that the zeroth slot of the mode array is reserved for this: it contains the vector of symbols "? ? ... ?" that corresponds to no mode declaration.)

Routines: none

BUILTIN

This routine is called when classifying the atoms and variables of a builtin function (these functions are expanded during run-time–see the routine expand). It is similar to classify but does not need its full generality.

1) Enter tokens into expand array that represent the atom found. 2) Classify variables into void (if new) or local. Thus temporary variables become local, but no variable become global based on its appearance in a built-in function.

Routines: none

CLASSIFY

This large routine is called when classifying the atoms and variables of a clause (except for the arguments of a built-in function; see builtin for this).

1) Recognize anonymous variables (written as '?'). If they are in skeletons, give

them a unique variable name, so they may be hashed into the symbol table. (This is a "global void", a variable that is treated as any other, except that in the head, there is no PLM instruction generated for it.)

2) Recognize other types of void variables, and temporary, local and global variables. Enter them into symbol table, keep track of the number of occurrences (one or many), and enter them into the item list.

This step is performed as follows: Whenever a variable is seen on a level deeper than three, it is classified immediately as global (unless mode '+' is declared for a level four variable). Otherwise, the first time a variable is seen, it is classified as void. If it is seen again and we are still in the head, it is temporary, but if we are in the body, it is local. Thus, the variable categories fall into the order or hierarchy "void, temporary, local, global". Once a variable has achieved a given classification, it can only move "higher" (to the right) in this ordering. In this way, it is possible to correctly classify the variables on a single pass through the clause.

4) Recognize different (non-variable) atom contexts, such as head-predicate, goal-predicate, or other atom.

5) Recognize different types of skeleton terms (in the head: uskel, uskelc, uskeld, and uskel1; in the body: skel–level three or nested–level four or more).

Routines: cl_var, cl_atom, cl_num, cl_goal, cl_nest, cl_void, collectively described in 1 through 5 above, and hashvar, described below.

ASSIGNVAR

This routine assigns numbers to variables in the symbol table. These are the numbers used as offsets into stack frames to find the corresponding value cell. Note (see assertx) that assignvar is called after an entire clause is processed, since it is only then that we know the correct classification for all the variables. This is needed since the assignment of variable numbers is in two independent sequences, one for locals and temporaries, and another for globals (see Figure 3, page 13). The variable number assignment can be performed quickly, since the symbol table indices for all the variables of a clause are available in the arrays var_listE and var_listI (see Appendix 8).

Routines: none

ATTACH

This routine creates the linked cons cell (clause cell) structure that provides a sequential access to each of the clauses of a procedure during the execution of a program. It is called (see assertx) after the clause has been translated into expand

array and item list entries. We know the location of the code for the clause at this point, even though it has yet to be placed there by genmacro. See Figures 7-11, (pages 31-35) for examples. The process is summarized in Section 5.3.

Routines: none

HASHVAR

This routine simply finds a free slot for a variable in the symbol table, or finds the index of the variable if it has been previously entered. The calling routine does not care which. The routine also flags in the entry itself a bit to indicate if this was a new variable or not. This is used for void variable classification.

Routines: hashindex, which performs the hashing function

CLR

This routine simply sets to zero (clears) the entries in the symbol table used during the processing of a clause. The entries are found easily since the indices are stored in arrays as the variable are encountered. Clearing of the symbol table is necessary since all variables in Prolog are local in scope to the clause in which they appear.

Routines: none

GENMACRO

This is another large routine that creates the code for a clause from the information gathered in the item list. The first occurrence analysis (necessary to decide what initialization to "undef" must occur) takes place here. It is suggested (see Appendix 1) that it be moved to the classification routines. Since it occurs in genmacro, no_op instructions can not be (easily) avoided in the PLM macro instruction stream.

1) Create code for the clause, place it in the code array.

2) Define the arguments for each macro (from information stored in the item list, expand array, and symbol table).

3) Perform the first occurrence analysis in order to establish the arguments for INIT and LOCALINIT.

Routines: genlit

GENLIT

This routine transforms arguments to a skeleton into the corresponding skeleton literal form. The arguments are read from the item list and are stored as cons cells in sequential locations in the skel_lit array. The format of cons cell used here differs from the simple $< address, address >$ form that the LISP storage management provides. See Figure 4, page 14.

Routines: none

RUN

The main purpose of this routine is to provide the central control for the calling of the C routines that define the PLM macros. Their execution, of course, constitutes the execution of the Prolog program entered by the user. A pointer called pc is used to access the C routine pointer from the code array. A simple loop transfers control to this routine, indirectly through the pc. The routines (PLM macros) themselves increment the pc or redefine it (for example, in a fail routine).

1) Initializes the PLM registers.

2) Runs the Prolog program.

Routines: none (except the indirectly accessed PLM routines, described in other appendices)

EXPAND

This routine is properly part of the run-time PLM routines, but is not a PLM macro. It is called by the macro LISP. Its function is to expand (dereference) all the variables in a term and create a linked structure of cons cells that corresponds to the resulting s-expression. This is passed to LISP for evaluation. See Appendix 2 for an example of recreating a constructed term representation as a written s-expression. This is precisely the function of EXPAND, except the final form, of course, is stored internally.

This routine attempts to avoid duplication of effort. If it has expanded a particular variable, a pointer to the result is saved (we make temporary use of the symbol table for this) in case this variable appears again in the same expression.

Appendix 8

Arrays used by the Compiler

The compiler makes use of several fixed length arrays to accomplish its processing. This appendix describes their fields and summarizes their function.

Note first the sizes of the C data types referred to below:

```
integer:          4 bytes
short:            2 bytes
unsigned short:   2 bytes
pointer:          4 bytes
```

Also, note that in C, a "union" is an object that at different times may hold different members. Its size is the size of the largest member it has been declared to contain.

The Symbol Table declared as: st [MAXARGS*10]

Fields:	Size:	Possible values or example entry:
atom cell pointer	pointer	pointer to ?x
type	short	VOID, TEMPORARY, LOCAL, or GLOBAL
occurrence flag	high-order bit of next field	ONE or MANY
variable number	unsigned short	non-negative integer

Notes:

The symbol table serves as a hash table for all the variables of a single clause. Its purpose is to hold all information for a variable that is independent of its position in the source. Before processing the next clause each field that was used is cleared, to ready it for the next clause. Direct access to the locations of the variables within the symbol table is provided by the var_listE and var_listI arrays.

The Item List	declared as: item_list[MAXARGS*6]	
Fields:	Size:	Possible values or example entry:
Union of: atom cell pointer symbol table index	pointer	1) pointer to non-variable atom, e.g. ''append" 2) non-negative integer index into st. 3) unused
type	unsigned short	36 types, e.g. ATOM, VAR, DOT, SKEL, CUT, LISP, etc.
argument number	unsigned short	non-negative integer
level number	unsigned short	non_negative integer
occurrence flag	unsigned short	1) ONE or MANY, if var 2) arity, if predicate 3) expand array index

Notes:

The item list holds almost a direct representation of the source clause in a linear array. The level number field keeps track of the nesting of the term. The item list and symbol table together contain all the information necessary for the macro generator to create the code for the clause. The information for the next clause will start at the beginning of the item list, overlaying the current contents.

All position dependent information is kept here. Every atom adds an entry, but some additional information is inserted. For example, every skeleton has an associated entry. Mode declarations are read to determine the type of skeleton entry to use.

One type of source information is not kept in the item list. Any term that will later be expanded has its arguments recorded in the expand array. A single entry is placed in the item list, with the occurrence flag field set to the expand array index.

The Expand Array	declared as: exp[EXPSIZE]	
Fields:	Size:	Possible values or

		example entry:
Union of: atom cell pointer symbol table index	pointer	1) pointer to non-variable atom, e.g. ''append" 2) non-negative integer index into st. 3) unused
type	unsigned short	10 types, e.g., ATOM, VOID, LOCAL, GLOBAL, DOT, LISP, RIGHT PAREN, LEFT PAREN
level	unsigned short	non-negative integer
variable number	unsigned short	non-negative integer
occurrence flag	unsigned short	ONE or MANY

Notes:

The expand array is used to hold any term that may later be expanded (i.e., all variables fully dereferenced and the s-expression cons cell list recreated) and passed to LISP. This occurs, then, for all LISP goals, and for the user query (which actually contains a call to the LISP "print" function). The information does not duplicate the item list, but rather replaces it for these terms. This creates no difficulties because the arguments of these terms never participate in unification and are not properly a part of the Prolog system. They do not, for example, contribute to the local/global classification (though the presence of a variable can promote a void to a local variable).

The Variable Arrays	declared as:	var_listE[MAXARGS] var_listI[MAXARGS]

Notes:

These unsigned short arrays simply record the symbol table array indices for all variables as they are entered into the expand array (var_listE) and the item list (var_listI). This speeds up access to the variables when 1) clearing the symbol table, and 2) assigning variable numbers, 3) expanding the second occurrence of a variable (see expand in Appendix 7).

The Code Array	declared as: cl_code[CODESIZE]	
Fields:	Size:	Possible values or example entry:
PLM macro	pointer	pointer to C routine that defines it
argument 1	integer	These depend on the
argument 2	integer	macro. Argument 3 holds
argument 3	integer	arity if a CALL, etc.

Notes:

This array corresponds to Warren's executable code area. It does not, however, hold skeleton literal data. The code for a clause can be deleted (by removing the cons cell that points to it), or added. Currently, new clauses are added at the end. Eventually, some form of memory management scheme will be needed here.

The Literal Array	declared as: sk_lit[CODESIZE/10]	
Fields:	Size:	Possible values or example entry:
type	unsigned short	See Fig 4, inner literals. SKEL, ATOM, NUM, GLOBAL
value pointer	pointer	points to values or variable offset.

Notes:

Here we store skeleton literal data. By not storing skeleton literals data in the code array (as Warren does) we are able to perform compaction on the code array without destroying literal data still in use.

The Mode Declaration Array declared as: mode[MAXPROC][MAXARGS]

Notes:

This two-dimensional integer array holds rows of mode declaration symbols. The rule field of head predicate atom cells points to a mode cons cell if there is a mode declaration associated with them. The car of this cell contains the index to this array which enables access to the correct row. We retain run-time access to the mode declarations in case dynamic asserts require them. Note that as a side benefit, this enables the user to change the modes for newly asserted clauses if desired.

The Stacks

Fields:	Size:	Possible values or example entry:
class	pointer	1) See Fig 4, constructs 2) Management Information
value	pointer	1) See Fig 4, constructs 2) Management Information

Notes:

The stacks are held in a single array whose initial space is allocated as the compiler executes. Thus, the user could choose the size of this array. Currently, the first element of the stack array is the bottom of the local stack, the middle element is the bottom of the global stack, and the last element is the bottom of the trail, which grows toward the global stack. Other arrangements are possible.

Appendix 9

Error Messages

This appendix is a simple list of all the error messages of use to the user. Many developmental messages have been eliminated from the compiler. These were of the sort "compiler error: genlit finds and unknown type", and so on.

The call to the error routine is shown. The first argument gives the severity level of the error. There are three:

- WARNING This is an informational message for the user.
- SEVERE/DISASTER These two types of errors will halt the compilation of the program or the running of it (as the case may be)

The second argument is a pointer to the offending clause (for printing the clause along with a message) or is 'nil'.

The third argument gives the message that is printed.

The error messages generated by the PROLOG part of the compiler are listed below with a brief explanation of the message wherever necessary.

nextatm.c

```
errorp(disaster,nil,"out of space: nesting level too great");
```

meaning : trying to create a very deeply nested list structure.

action : modify the program

pass1.c, pass2.c, pass3.c, pass4.c

```
errorp(disaster,current_cl,"internal conflict of variable :
  do not use the form '?:xxx' for variable name");

errorp(disaster,current_cl,"pass1.cl_atom: unknown atom type");
```

meaning : internal error. Reader returned unknown type.

action : report to the Prolog Compiler project administrator.

```
errorp(severe,current_cl,"too many anonymous variables");
```

meaning : The current clause has many variables that are used only once in the clause (serving no purpose).

action : get rid of such variables.

```
errorp(disaster,nil,"assert expected at the beginning ");

errorp(disaster, nil, "head-predicate needed to be asserted");

errorp(severe,nil,"cannot redefine builtin function");
```

meaning : The current clause tries to redefine a built-in function that is not re-definable, such as 'or' 'and' '!' .

action : Change the predicate name to avoid conflict with built-in functions.

```
        errorp(warning,nil,"redefining LISP function with user rule
\n");
```

meaning : The current clause tries to redefine a built-in function such as append.
The system will allow you do that. But, beware. Any calls to this rule appearing physically before this rule would have been compiled as calls to LISP
and not as the ordinary calls to prolog rule !!!

action : Change the predicate name to avoid conflict with built-

in functions.

```
        errorp(disaster, nil, "variable goals not implemented \n");

        errorp(disaster,nil,"syntax error: expecting '(goal ...)'");
```

meaning : Missing '(' after if .

action : Check the current clause for syntax errors.

```
        errorp(severe, nil, "missing '(' before a goal");
```

meaning : Missing '(' before a goal.

action : Check the current clause fro syntax errors.

```
        errorp(disaster ,nil,"variable goals not yet implemented\n");

        errorp(severe, nil, "after 'and' , 'or' and '!'
        '(' is a must " );
```

meaning : The 'or' , 'and' and '!' goals must be enclosed in parentheses.

action : obvious.

```
        errorp(severe, nil, "at least 2 args for or, and");
```

```
errorp(severe, nil, "not more than one argument for '!'");

errorp(severe,nil,"cannot assign to non-variable");

errorp(disaster,nil,"syntax error in assignment goal");

errorp(disaster, nil, "dot found at wrong level");
```

meaning : In the current clause, there is a dot that does not belong to a list !!! (In prolog '.' is the built-in list construction operator.)

action : obvious.

```
errorp(disaster,nil,"only atom can be a predicate name");
```

meaning : In the place of predicate, there can be only an atom either a constant or a variable. A compound term such as a list is not allowed.

action : obvious.

```
errorp(disaster, nil, "only '(' expected at level one");
```

meaning : Following '(assert ' at the top level, '(' is expected .

action : Obviouly, the clause following the assert does not start with a '(' that needs to be fixed.

```
errorp(disaster, nil, "only 'mode' or LP at this level");
```

meaning : Expecting a mode declaration or the start of a clause. i.e, a '(' .

action : rectify the syntax error

```
errorp(severe, nil,"'nil' cannot be procedure name");
```

meaning : Obviously, a rule can not start with 'nil'.

```
action   :    use some other name for a predicate.

          errorp(severe, nil,"level incorrect for mode declaration");
          errorp(warning, nil,"too many mode declarations");

meaning  :    Please refer to the chapter 2 of this thesis that details
              the correct syntax for  mode declarations using BNF grammar.

          errorp(disaster, nil,"too many args in mode declarations");

meaning  :    Internal limit of the compiler for the mode array is exceeded.

action   :    reduce the mode declarations or increase the internal limit
              and recompile the compiler !!



run.c
-----
          errorp(disaster,nil,"out of space: local stack full");
          errorp(disaster,nil,"out of space: global stack full");

meaning  :    obvious

action   :    reload the compiler  with the  -p option to increase
              the stack size.

          errorp(disaster,nil, " ");

          errorp(disaster,nil,"from uref, unknown argument type");
          errorp(disaster,nil,"from uref1, unknown argument type");

meaning  :    internal error. Bug ?

action   :    report to the prolog project administrator.

          errorp(disaster,nil,"from fail, nil FL");
          errorp(disaster,nil,"from dpfail, nil FL");
          errorp(disaster,nil,"from ctfail, nil FL");
```

```
errorp(disaster,nil,"assign: unknown type returned from :=");
errorp(disaster,nil,"from uref or uref1, unknown type");
```

meaning : internal error. Bug ?

action : report to the prolog project administrator.

unify.c

```
errorp(disaster,nil,"from unify, unknown type");
```

meaning : internal error. Bug ?

action : report to the prolog project administrator.

Appendix 10

Index to Warren's Volume 1

The following index may be useful for anyone reading Warren's report for the first time. It often directs you to the exact line containing the reference. We found it particularly useful in understanding a new term or concept that seemed unclear where first explained.

```
        Index to:

        IMPLEMENTING PROLOG: compiling predicate logic programs
          Volume 1
        by David H. D. Warren
        DAI RESEARCH REPORT NO. 39

This index covers pages 3 through 64 of volume 1.

Notation used:  pn/gn:ln = page number pn, paragraph number gn, line
                          number ln.  If gn or pn are negative, count
                          from bottom up.

    Example:    50/3:-2  page 50, 2nd line from bottom of 3rd paragraph
                45/mid   page 45, about middle of page

                pn/gn1;gn2;...;gni = (multiple reference) page number
                          pn, paragraph numbers gn1, gn2,...., and gni

Abbreviations:  s.a.   see also
                def.   definition
                ff     and following  (50/3ff = page 50, 3rd & succeed-

                       ing para.) (1/2:3ff = ...3rd & succeeding lines)

                ind. ref. indirect reference  (not explicitly mentioned)
                diag.  diagram
                fig.   figure
```

Bibliography

Roach J. and Fowler, G., The HC Manual: Virginia Tech Prolog/Lisp Interpreter, Department of Computer Science, Virginia Tech, October 1981.

Warren, David H.D., Implementing Prolog: compiling predicate logic programs, Volume 1, DAI Research Report No. 39, May, 1977.

Warren, David H.D., Implementing Prolog: compiling predicate logic programs, Volume 2, DAI Research Report No. 40, May, 1977.

Warren, David H.D., An Abstract Prolog Instruction Set, Technical Note 309, SRI Project 4776, Standford Research Institute International, Menlo Park, CA, October 1983.

Additional Reading

Tick, E. and Warren, D.H.D., "Towards a Pipelined Prolog Processor", 1984 International Symposium on Logic Programming, Sponsored by the IEEE Computer Society, pp 29-40.