A Tutorial on the Warren Abstract Machine
for Computational Logic

John Gabriel, Tim Lindholm, E. L. Lusk, R. A. Overbeek
Mathematics and Computer Science Division

June 1985

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
CONTENTS

1. Introduction .................................................................................................... 1
2. Procedures ...................................................................................................... 2
3. The Abstract Machine .................................................................................. 3
   3.1. The Argument Registers ......................................................................... 4
   3.2. The Local Stack ...................................................................................... 4
   3.3. The Global Stack .................................................................................... 6
   3.4. Some Facts to Remember ..................................................................... 6
4. The Basic Structure of Procedures in Warren Assembler Language .......... 7
   4.1. A Simple Lowest-level Procedure ........................................................... 7
   4.2. Introduction to Backtracking ................................................................ 8
       Exercises .................................................................................................. 10
   4.3. A Procedure that Invokes a Single Subroutine ..................................... 10
   4.4. A More Complex Example .................................................................. 12
5. Temporary and Permanent Variables .......................................................... 17
6. Allocate/Deallocate and Trimming ................................................................ 17
   Exercise ....................................................................................................... 18
7. Indexing ......................................................................................................... 18
   7.1. An Example .......................................................................................... 18
   7.2. The General Indexing Mechanism ....................................................... 21
   7.3. A Note on Dynamic Modification of Code ............................................. 23
8. Another Example .......................................................................................... 23
   Exercise ....................................................................................................... 26
9. Binding One Valuecell to Another ............................................................... 26
10. Unsafe Variables ......................................................................................... 27
11. Miscellaneous Instructions Not Yet Covered ............................................ 27
12. Additional Operations to Support Theorem Proving ................................. 28
   12.1. A Theorem-Proving Example in Warren Assembler Language ............ 28
   12.2. Half-Match Unification ....................................................................... 39
13. A Comparison with Warren's Original Description .................................... 39
14. Performance .................................................................................................. 41
15. Summary ..................................................................................................... 42

Acknowledgement .......................................................................................... 42
References ........................................................................................................ 23

Answers to Exercises ....................................................................................... 49
1. Introduction

Computational logic can roughly be defined as that branch of artificial intelligence that is based on logic. It includes logic programming, theorem-proving, rewrite-rule systems, and production-rule systems. Such fields are going through a truly interesting transition. It now is becoming apparent that performance improvements for computational logic systems could be as high as 3 to 5 orders of magnitude during the next 5 to 7 years. These improvements will come from three distinct sources:

1. The processors used in the commonly available computing systems will improve by a factor of 5 to 10. That is, commonly available, cheap processors will increase substantially in speed.

2. Multiprocessors featuring up to 1024 nodes will become widely available. Each node may contain 4 to 16 tightly coupled processors (on a shared memory).

3. Substantial speedups will occur due to improvements in the implementation of the basic algorithms.

The last point will be the focus of this document.

It is our belief that the Warren Abstract Prolog Machine[1] represents a major breakthrough in the design of computational logic systems. It certainly will be the basis of numerous implementations of logic programming. However, its significance goes well beyond logic programming. It seems likely to us that high-performance implementations of classical theorem-proving algorithms, such as hyperresolution, paramodulation, demodulation (rewrite-rule systems), and subsumption, based on an extended version of the Warren machine will all appear within the next three years.

Since no easily accessible, tutorial description of the Warren machine exists (at least as far as we know), we will start this document with a basic introduction to the motivation of the machine and the instructions that define it. We will then
discuss the fairly limited extensions required to extend the machine for more
general use outside of implementations of logic programming.

2. Procedures

The Warren machine is very similar to an abstract machine for the execu-
tion of recursive, block-structured languages such as Pascal or C. The primary
differences involve unification in the procedure call mechanism and the
automatic consideration of alternatives (backtracking).

The basic programming unit in the Warren machine, as in most program-
ing languages, is the procedure or subroutine. We will attempt to gradually fill
in the details of exactly how to code procedures in Warren assembler language.
Initially, the reader should think of a procedure as roughly analogous to a pro-
cedure in a more common language, such as Pascal or a familiar assembler
language.

Although the applications of the Warren machine go well beyond implement-
ing Prolog, it was certainly inspired by a desire to create a high-performance
implementation of that language. Hence, it will be of some use to reflect on the
role of procedures in Prolog. Let us consider the following example:

% path(X,Y,Z) sets Z to a list of the nodes in a directed path
% from X to Y. Thus, path(a,e,X) might instantiate
% X to [a, c, d, e] in the case in which a is
% connected to c, c to d, and d to e.

path(X,Y,Z) :- pathrecurs(X,Y,Z,[X,Y]).

pathrecurs(START,END,ANSWER,EXCLUDELIST) attempts to compute
% ANSWER as a path from START to END, subject
% to the restriction that the path cannot include
% nodes from EXCLUDELIST (except that no check
% is made to keep START or END from being in
% EXCLUDELIST—the object is just to avoid
% getting caught by cycles in the graph).

pathrecurs(START,END,[START,END],EXCLUDELIST) :-
  connected(START,END).
pathrecurs(START,END,[START|TAIL],EXCLUDELIST) :-
  connected(START,NEXT),
  not_elof(NEXT,EXCLUDELIST),
  pathrecurs(NEXT,END,TAIL,[NEXT|EXCLUDELIST]).

% elof(X,Y) is true iff X occurs in the list Y.

elof(X,[X|Y]).
elof(X,[H|T]) :- elof(X,T).

% not_elof(X,L) is true iff X does not occur in the list L.

not_elof(X,[H|T]) :- elof(X,L), !, fail.
not_elof(X,L).

This Prolog program for computing paths between nodes in a graph contains
four procedures. Presumably, there is also a collection of unit clauses (or
facts) of the form

connected(a,b).
connected(a,c).
connected(b,d).

which define a specific directed graph. This collection of facts will be considered
a fifth procedure, since we will define a Prolog procedure as the collection of
clauses in which the head literal contains a given predicate symbol and has a
specific number of arguments.

We will use the term "alternative" to refer to multiple approaches within a
single procedure. An alternative corresponds to a clause in the Prolog represen-
tation of such a procedure. In addition, procedure names used within program
segments will be of the form "procedure/arity" (e.g., path/3).

This set of procedures offers enough diversity to illustrate most of the cases
required to understand the concept of procedure within the Warren machine.
The following cases must all be considered in detail:

1. The procedure connected/2 illustrates the situation in which alterna-
tives exist (due to the fact that the procedure contains more than one
clause), but none of the alternatives requires the invocation of subrou-
tines (or subprocedures). That is, connected/2 is a lowest-level sub-
routine. We have listed it first, since we view "lowest-level" subroutines
as fundamentally simpler than those procedures that invoke other sub-
routines.

2. The procedure path/3 illustrates the situation in which there is only
one approach to computing the desired result—there are no alterna-
tives (although subroutines may, of course, introduce alternatives). It
also illustrates the case in which a procedure requires the invocation of
a single subroutine (in this case pathrecurs).

3. The procedure pathrecurs/4 features alternatives (since there are two
clauses, one can think of the procedure as having two approaches that
can be used to compute the desired results). In addition, the second
clause of pathrecurs requires the successive invocation of three sub-
routines.

We are going to analyze exactly how each of these different types of procedures
is coded in Warren assembler language. Initially, we will defer the question of
how to handle the case in which a procedure cannot compute a desired solution
(or another desired solution, in the case in which multiple solutions will be
returned). The process of recovering from failure (to go on and explore other
alternatives), or simply going on to consider another solution, is called back-
tracking. This is a major consideration, and we will return to it once the basic
overview of each type of procedure has been discussed.

3. The Abstract Machine

There are three fundamental data areas manipulated by a procedure—the
argument registers, the local stack, and the global stack. In this section, we try
to clarify the essential role played by each of these data areas and illustrate
their use by coding several short procedures. Throughout this discussion, we
have tried to avoid confusing two uses of the term "variable". The common use
of the term, to reflect an area of memory manipulated by a program, is avoided. Rather, the term is used to refer only to logical formulas that can be instantiated. When it is necessary to speak of a data item conforming to the common notion of variable, we shall call the item a machine register, or simply register. This can cause some confusion, as well, since most programmers have a fairly precise notion of what constitutes a machine register. However, our use of the term is actually quite close to the common usage. For example, when we speak below of the field continst, which is used to contain the address of where control should return upon successful completion of a subroutine, it is quite realistic to envision an actual machine register containing the value.

3.1. The Argument Registers

A procedure is passed a number of arguments. In most languages, some arguments are input arguments and some are output arguments, with the goal of the procedure being to compute the values of the output arguments from those of the input arguments. A similar, but not identical, outlook exists for the Warren machine.

The arguments are referenced by a set of registers—A0, A1, A2, ...An. The arguments themselves are logical formulas. There are four types of logical formulas.

1. **variables.** In the Warren machine, we identify the variable with a structure in memory called a valuecell which will be used to record instantiations of the variable.

2. **constants.** Constants can be symbols, integers, floating-point numbers, or the nil list. A symbol is just a string of characters representing a predicate symbol, function symbol, or uninterpreted constant.

3. **lists.** We will represent a list as [l1,l2,l3,...,lm], where each lm is a logical formula. A list has a head, consisting of the first element of the list, and a tail, consisting of the remaining elements of the list. The empty list is represented by the symbol "nil". If we use the notation [H|T] to represent the list with head H and tail T, then the list [l1,l2,l3,...,lm] is equivalent to [l1][l2][l3][...][lm|nil][...].

4. **structures.** A structure is a logical formula of the form func(f1,f2,f3,...,fk), where func is a symbol and f1,f2,f3,...,fk are all logical formulas.

A procedure invocation (i.e., invoking the procedure with a fixed set of values for the arguments) amounts to a request to compute a set of answers. For example, the invocation of the procedure path/3 with the three arguments a, b, and X might be viewed as a request to compute the set of possible values of X that represent paths between the two points a and b. In general, an answer is represented as an instantiation of the set of arguments. Thus, the normal distinction between input and output arguments is blurred. An "input" argument may be instantiated in some cases (where the answer would only apply to the instantiation of the original input argument).

3.2. The Local Stack

The only type of data that we have mentioned is the collection of arguments referenced by the registers A0, A1, A2, ...An. A procedure will require other data areas in order to compute the desired results. The local stack is one of several other data areas that can be manipulated by instructions in a procedure. Upon entry to a procedure, a machine register points to a position in the local stack. We will refer to this register as LPOS. A procedure may claim memory from the
local stack for two distinct uses:

1. If the procedure includes alternative approaches for computing solutions, a mechanism for keeping track of which alternative should be tried next is required. The set of data items required to record this information is collectively called a choice point. The exact contents and use of choice points will become clear only after we have considered the question of how to handle backtracking (and, as we stated above, we are going to defer that discussion until after we have established the overall framework). For now, it is enough to grasp that the role of a choice point is to record a position in the set of alternatives, and to know that this position is recorded on the local stack. Note that this position can be deallocated at the point in which the procedure begins to pursue the last alternative (and, hence, is not required for procedures in which there is only one approach).

2. The second use of the local stack involves "scratch areas". Upon entrance to a procedure, two critical registers which we will call continst (continuation instruction) and current (current environment), contain values established by the invoking procedure. The invoking procedure does not save these registers before the invocation, since the invoked procedure may not need to alter them (in which case the save/restore would be wasted). However, if the invoked procedure needs to alter these registers, it must save the original values and see that they are restored before control is returned to the invoking procedure. The scratch area in the local stack is used for this purpose. In addition, the values in the argument registers A0, A1, ..., An are not preserved through subroutine invocations. Hence, if a reference to a data item (i.e., a logical formula) must be preserved through a call to a subroutine, the reference must be saved in the scratch area. This is achieved by establishing a set of valuecells in the local stack. These value cells can then be bound to the desired formulas to preserve the ability to reference the formulas. Later, the binding established in these value cells in the local stack can be transferred to the argument registers (if they are to be used as arguments to an invoked subroutine) or to value cells in the global stack (if they are to become part of an answer returned to the calling procedure). We shall refer to the scratch area used to preserve the continst, the current, and the local value cells as an environment. Basically, an environment provides the scratch area required to save and restore values in those cases in which subroutines might destroy references that must be preserved. An environment is allocated at the start of utilizing a single alternative approach towards computing a desired answer, since the amount of scratch area required will depend on the sequence of subroutine invocations required by that approach. It is deallocated at the end of exploring that single approach (and another environment may have to be allocated/deallocated, if another approach has requirements for scratch memory). Now, we can also clarify the contents of the continst and current registers. The continst records the address to which control should return in the invoking procedure, and current is used to record the address of the "current environment" (which is the environment established by the invoking procedure, until the invoked procedure decides to establish its own environment, if necessary).
3.3. The Global Stack

The global stack is a second central data area that can be manipulated by instructions in a procedure. It is used to construct logical formulas that are used as answers for the calling procedure. That is, logical variables in the input arguments will be bound to these constructed formulas, which then become the answer returned to the caller. Upon entrance to a procedure, a register called GPOS will contain the address of the next available memory in the global stack. The memory in the global stack will be deallocated only when the calling procedure no longer needs to reference the constructed answers. Exactly how this is achieved must, again, be deferred until we cover the question of how to handle backtracking.

3.4. Some Facts to Remember

We have noted that formulas can occur in both the local and global stacks. At this point we include a short list of statements that are true, but for which a detailed explanation must be deferred. In general, they are consequences of the more temporary nature of memory in the local stack compared to the global stack.

1. No valuecell in the global stack can be bound to a formula in the local stack.
2. Valuecells in the local stack must reference either formulas in the global stack or formulas that occur earlier in the local stack.
3. The only formulas that occur in the local stack are variables (represented by valuecells).
4. Structures and lists exist only in the global stack. Since no valuecell in the global stack can be bound to a valuecell in the local stack, no subformulas of a list or structure can be in the local stack.

4. The Basic Structure of Procedures in Warren Assembler Language

At this point, we have introduced the central data areas—the argument registers, the local stack, and the global stack—and are ready to consider the basic structure of the routines coded in Warren assembler language. Throughout this discussion we are going to ignore a significant optimization introduced by Warren called indexing. The basic idea of indexing is to isolate rapidly a subset of alternatives in a routine, by looking at the first argument passed to the routine (via A0). That is, indexing isolates a subset of possible alternatives. Those alternatives that are "ruled out" by indexing could not be usable, and this can be detected rapidly by examining the first argument. For example, suppose that the procedure connected/2 is invoked with A0 referencing the logical formula a and A1 referencing the formula z. Then, the set of alternatives from the procedure connected/2 that must be considered are those in which the first argument of the head literal (in the case of connected/2, the only literal) is either a variable or the constant a. Indexing will be considered in detail in a later section. For now, we will assume that all alternatives are possibly applicable and must be considered.

Throughout this section, we will present examples in the following format (although not always in the following order):
4.1. A Simple Lowest-level Procedure

Let us start with the procedure `connected/2`, since it is a lowest-level procedure. It could be simpler (if there were no alternatives to consider), but its structure is fairly straightforward. We assume, for simplicity, that the following three clauses make up the entire procedure:

```
connected(a,b).
connected(a,c).
connected(b,d).
```

Here then is the Warren assembler code for the procedure:

```
connected/2
try_me_else  C2,2  % create choice point
   get_constant a,A0  % match 1st arg against "a"
   get_constant b,A1  % match 2nd arg against "b"
   proceed  % successful return
retry_me_else C3  % update position
   get_constant a,A0  % match arg against "a"
   get_constant c,A1  % match arg against "c"
   proceed  % successful return
trust_me_else_fail C3  % last alternative
   get_constant b,A0  % match arg against "b"
   get_constant d,A1  % match arg against "d"
   proceed  % successful return
```

Each of the three sections of code corresponds to one of the Prolog clauses in the procedure. The first instruction, the `try_me_else`, creates a choice point in the local stack (with C2 as the next alternative to consider and two argument registers to be saved). The second alternative begins with a `retry_me_else`, which updates the "next alternative" to C3. The last alternative begins with `trust_me_else_fail`, which deallocates the choice point in the local stack (since there will be no more alternatives to consider after the third).

The `get_constant` instructions are used to match an incoming argument against the specified constant. The incoming argument must be either a variable or a constant; otherwise, backtracking will occur (and the next alternative will be considered). If the incoming argument is a variable (i.e., if the argument register references a value cell), the value cell will be bound to the specified constant.

Finally, the `proceed` instructions cause a successful return to the caller.

The reader should consider the example carefully, keeping in mind that we have not yet clarified the whole topic of how alternatives become considered via
backtracking. For now, it is enough to see that if the first clause "fails" (because one of the get_constant instructions fails), then control will pass to the next alternative, beginning at C2.

**try_me_else label,n**
This instruction is used only in procedures which include multiple alternatives. It precedes the code for the first alternative. It causes a choice point to be created in the local stack and sets the next alternative (stored in the choice point) to the address given as its operand. The choice point will save values for the first n argument registers. Execution continues with the instruction immediately following the try_me_else.

**retry_me_else label**
In a procedure that contains several alternatives, this instruction must precede the code for all but the first and last alternatives (i.e., it precedes the middle alternatives). It causes the "next alternative" in the choice point to be set at the address given as its operand. Execution continues with the instruction immediately following the retry_me_else.

**trust_me_else fail**
In a procedure that contains several alternatives, this instruction should precede the last alternative. It causes the current choice point to be deallocated from the local stack (since there are no more alternatives to consider). Execution continues with the instruction immediately following the trust_me_else.

**get_constant constant,Ai**
The get_constant instruction takes two operands. The first designates a constant, and the second an argument register. The instruction attempts to "match" the constant with the incoming argument. If the argument register references an unbound variable, the valuecell will be bound to the designated constant. If the argument register references the same constant, no action occurs. In either of these cases, the get_constant succeeds and the next instruction will be the one immediately following the get_constant. If the match does not succeed, backtracking will occur.

**proceed**
The proceed instruction is used to return from an alternative that did not require invoking a subroutine (i.e., an alternative represented by a unit clause). The next instruction executed will be designated by the contents of the continst register.

### 4.2. Introduction to Backtracking

Backtracking is a topic that we will revisit several times. Initially, we will attempt to give just enough of an overview for the reader to be able to form an accurate (though incomplete) conceptual grasp of the flow of control in the Warren machine. Backtracking occurs whenever an instruction (such as get_constant) fails, or when another answer to a query is desired. To understand the implementation, it is useful to understand what information is stored in a choice point. The following registers together determine the contents of a choice point:

1. The continst is a register that gives the address of where control should go upon successful completion of a procedure.
2. The **current** register contains the address of the "current environment" in the local stack. The details of what this means will be covered later.

3. The **lastchpt** register contains the address of the last choice point in the local stack (that is, the previous choice point).

4. The **LPOS** register contains the current position in the local stack.

5. The **GPOS** register contains the current position in the global stack.

6. The **TPOS** register contains the current position in the trail, which is a fairly specialized data area used to record instantiations of variables (so that they can be returned to an uninstantiated state at the appropriate time).

7. The argument registers record the arguments passed to procedures.

8. The **nextclause** register contains the address of the next alternative to be considered.

When a choice point is created, the current values of these registers are stored in the choice point.

To understand exactly how the contents of a choice point can be used, it is first important to form an idea of what is meant by the **state** of the Warren machine. Essentially, the state of the machine is a position in some computation. This is determined by the first seven items listed above that are stored in a choice point, the **nextinst** register (which gives the address of the next instruction to be executed), and the contents of the local and global stacks. By contents of the stacks, we mean the data items stored in the stacks ahead of the current positions (given by **LPOS** and **GPOS**). The point that is significant about a choice point is that if the first seven items in the list are reset from their values in the choice point, and if the next instruction address **nextinst** is reset from the **nextclause** register in the choice point, then the machine is almost completely reset to consider the next alternative. All that remains to be tidied up is the contents of the stacks ahead of the current positions that may have been altered. As it turns out, all that might have been altered are variable valuecells (through the process of instantiation). Hence, if the instantiations are carefully recorded so that they can be "undone" (by setting the altered valuecells back to their uninstantiated status), then a complete reset will have been accomplished.

Valuecells that must be uninstantiated in a reset are recorded in the **trail**. As valuecells are altered, entries are added to the trail, which is itself a stack. A current position, called **TPOS**, is maintained in the trail. To reset the valuecells altered since a choice point was created, the machine simply pops entries off the trail and resets the corresponding valuecells, until the end of the trail reaches the value of **TPOS** stored in the choice point.

Hence, once a choice point has been added to the local stack, all computations that follow the creation of the choice point can easily be reset to consider the next alternative represented by the choice point (discarding all contents of the local and global stacks past the points given by the stored **LPOS** and **GPOS** values) by simply reloading the registers from the choice point and resetting valuecells based on information from the trail. This gives an extremely efficient and powerful mechanism for rapidly recovering from a computation which cannot be completed—the machine just "resets" back to consider the next alternative. When all of the alternatives in a choice point have been considered, the choice point is deallocated, and the previous choice point becomes the mechanism for resetting to previously unexplored alternatives. In what follows, the phrase "backtracking occurs" means that this reset operation takes place.
Exercises

1. Give the Warren assembler code for the Prolog procedure consisting of the single clause

\[ \text{link}(2, 17, 2400). \]

2. Now give the Warren assembler code for the Prolog procedure consisting of the following two clauses:

\[ \begin{align*}
\text{link}(2, 17, 2400). \\
\text{link}(2, 18, 1200).
\end{align*} \]

Which of the instructions causes a choice point to be created? Which field in the choice point contains the address of the next alternative, and what will it contain when the choice point is created?

3. Finally, give the code for the Prolog procedure consisting of the following four clauses:

\[ \begin{align*}
\text{link}(2, 17, 2400). \\
\text{link}(2, 18, 1200). \\
\text{link}(1, 17, 1200). \\
\text{link}(2, 16, 300).
\end{align*} \]

4.3. A Procedure that Invokes a Single Subroutine

The procedure \textit{path/3} given previously was defined by the following single Prolog clause:

\[ \text{path}(X, Y, Z) : - \text{path_recur}(X, Y, Z, [X, Y]). \]

We will first give the definitions of the new instructions required to write the procedure in Warren assembler language, then we will present the actual code.

\textbf{put_list} \ Al

Execution of this instruction causes a skeletal \textit{list} (that is, a list in which neither the head nor the tail has yet been defined) to be created in the global stack. The argument register \textit{Al} is set to reference the skeletal list. Furthermore, a register called \textit{nextarg} is set to reference the spot in the list into which a reference to the head of the list being constructed should be stored. The register \textit{nextarg} is used only when processing lists and structures and points to the next argument (in a list, the head or tail) to be processed. Finally, the machine is put into \textit{write mode} in preparation for filling in these arguments on the global stack.

\textbf{unify_pil}

This instruction is identical to \textit{unify_constant}, except that the constant in this case is the empty list.

\textbf{unify_x_value} \ Al

If the machine is in \textit{write mode} the reference in \textit{Al} is copied into the global stack at the location specified by \textit{nextarg}, and \textit{nextarg} is incremented (to the next argument in the structure or list being built). If the machine is in \textit{read
mode, then the argument designated by nextarg is unified with the logical formula referenced by Ai. If the unification fails, backtracking occurs. Otherwise, the "dereferenced result" is put into Ai. That is, if Ai references anything except a valuecell, Ai will not be altered. On the other hand, if Ai references a valuecell that is bound to another logical formula, Ai will be altered to reference the formula to which the valuecell was instantiated. If the valuecell is bound to another instantiated valuecell, the chain of valuecells is examined, and Ai is altered to reference either an unbound valuecell or a formula that is not a variable (i.e., Ai is set to the image found by continuing down the chain of bound valuecells). This process of locating the ultimate image of a reference is called dereferencing. If the unification is successful, nextarg is incremented.

unify_x_local_value Ai
This instruction is identical to unify_x_value, except in the case in which the value in Ai dereferences to a valuecell in the local stack. In this case, a new valuecell is allocated in the global stack, the valuecell in the local stack is bound to the new valuecell, and Ai is set to reference the new valuecell in the global stack.

execute procedure
This instruction resets nextinst to reference the first instruction in the designated procedure. Thus, when control is returned from the executed procedure, it will not return to the procedure in which the execute occurred, but rather to the caller of the procedure that issued the execute. It is used only when the last subroutine in a procedure is invoked.

The code for path/3 in Warren assembler language is as follows:

```
pah/3     put_list   A4          % first build [Y]
          unify_x_local_value A1   % [Y]
          unify_nil               % nil
          put_list   A3          % build 4th argument
          unify_x_local_value A0   % [X]
          unify_x_value A4        % [Y]
          execute     pathrecurs/4 % go to pathrecurs
```

Consideration of the Prolog representation of this procedure shows that its basic meaning is "path/3 must construct a fourth argument and invoke pathrecurs/4". This is exactly what is accomplished by the code given above. A4 is used as a scratch variable in which to construct the list [Y].

There are several fine points to be noted in the use of the above instructions. The first involves the concept of mode. The Warren machine is said to be operating in either read mode or write mode. Some instructions set the mode, and others take actions that depend on the setting of the mode. Thus the mode is one way in which information is passed from one instruction to another. The put_list instruction sets the mode to write mode. The actions taken then by the unify instructions depend on the fact that the machine will be in write mode. Later, we shall cover uses of the unify instructions that are quite different from their use here (in building a list for the fourth argument). In general, unification in write mode causes formulas to be written on the global stack, whereas unification in read mode reads formulas that already exist.

The other subtlety in the above code involves the difference between the unify_x_value and the unify_x_local_value instructions. Both instructions (when executing in write mode) are used to fill in arguments in a list or
structure being constructed on the global stack. The \texttt{unify}_x\_local\_value\ instruction includes a check to make sure that the referenced formula is not a valuecell in the local stack (remember that no list of structure on the global stack may contain a reference to a valuecell on the local stack). This instruction must be used when the argument register might reference a valuecell in the local stack (which is the case for any incoming argument). On the other hand, in the code for \texttt{path/3}, \texttt{A4} must point to a list constructed in the global stack, because we just put it there. Hence, the \texttt{unify}_x\_value\ instruction is employed to avoid a superfluous check.

4.4. A More Complex Example

Now let us consider a somewhat more complex example, the \texttt{pathrecurs} procedure described by the following Prolog clauses:

\begin{verbatim}
pathrecurs(START,END,[START,END],EXCLUDELIST) :-
  connected(START,END).

pathrecurs(START,END,[START|TAIL],EXCLUDELIST) :-
  connected(START,NEXT),
  not_elof(NEXT,EXCLUDELIST),
  pathrecurs(NEXT,END,TAIL,[NEXT|EXCLUDELIST]).
\end{verbatim}

The Warren assembler language version of this procedure is as follows:

\begin{verbatim}
pathrecurs/4
  try_me_else C2,4 % create choice point
  get_list A2 % [ 
  unify_x_local_value A0 % START [
  unify_x_variable A4 % ]
  get_list A4 % 
  unify_x_local_value A1 % END ]
  unify_nil % ]
  execute connected/2 % invoke connected C2

trust_me_else_fail allocate 4 % last alternative
  get_y_variable Y0,A1 % environment with
  get_list A2 % 4 valuecells
  unify_y_x_local_value A0 % START]
  unify_y_variable Y1 % TAIL]
  get_y_variable Y2,A3 % save EXCLUDELIST in Y2
  put_y_variable Y3,A1 % point A1 at Y3 (NEXT)
  call connected/2,4 % invoke connected
  put_y_value Y3,A0 % point A0 at NEXT
  put_y_value Y2,A1 % point A1 at EXCLUDELIST
  call not_elof/2,4 % invoke not_elof
  put_unsafe_value Y3,A0 % set A0 to NEXT
  put_y_value Y0,A1 % set A1 to END
  put_y_value Y1,A2 % set A2 to TAIL
  put_list A3 % ]
  unify_x_local_value A0 % NEXT]
  unify_y_local_value Y2 % EXCLUDELIST]
  deallocate % remove environment
\end{verbatim}
Here we have an example that is substantially more complex than our preceding two procedures. Since there are alternatives (i.e., two clauses in the procedure), a choice point will have to be constructed. In addition, we will need scratch valuecells in the local stack (i.e., in an environment) to keep track of temporary results.

To begin with, the `try me else` instruction creates the choice point necessary to handle the alternatives. The second argument indicates the number of argument registers to be saved and then restored between alternatives. This number is always the number of arguments to the procedure.

First let us analyze the code generated for the first alternative. Essentially, one could think of this procedure as first building a list to return as the third argument of the caller (i.e., bind the third argument of the calling procedure to the list `[START,END]`, and call `connected` to determine whether the two points are actually connected. This interpretation makes sense in the case in which the first two arguments are input arguments, and the third argument is an output argument. However, it is legitimate to call the routine with

\[
\text{pathrecurs}(X,Y,\text{list},Z)
\]

in which case a completely different interpretation is required. It is very instructive to see exactly how the actual code handles both of these cases. First, the `get_list` instruction checks to see whether the third argument is an uninstantiated variable or a list (if it is neither, backtracking will automatically occur). In the usual case, the third argument will be a variable, in which case the `get_list` instruction will put the machine in write mode. This will cause the

\[
\begin{align*}
\text{get_list} & \quad A2 \quad \% [ \\
\text{unify} & \quad A0 \quad \% \text{START} \\
\text{unify} & \quad A4 \quad \% \text{[} \\
\text{get_list} & \quad A4 \quad \% \text{END} \\
\text{unify} & \quad A1 \quad \% \text{]} \\
\text{unify} & \quad \text{nil} \quad \%
\end{align*}
\]

instruction sequence to construct the list `[START,END]` in the global stack (pointing A2 at the constructed list). On the other hand, if the third argument is a list, the machine proceeds in `read mode`, in which case `START` and `END` are unified with the head and tail of the list. Note that A4 is being used here as a "scratch" register.

Finally, assuming that the incoming arguments are successfully processed by the `get_list` and `unify` instructions, control is then transferred to the `connected` procedure by the `execute` instruction.

It is perhaps worth noting that another way to code this first alternative would be as follows:

\[
\begin{align*}
\text{put_list} & \quad A4 \quad \% \text{first, build [END]} \\
\text{unify} & \quad A1 \quad \% \\
\text{unify} & \quad \text{nil} \quad \%
\text{get_list} & \quad A2 \quad \% [ \\
\text{unify} & \quad A0 \quad \% \text{START]} \\
\text{unify} & \quad A4 \quad \%
\end{align*}
\]

execute \hspace{1cm} \text{pathrecurs}/4 \% invoke \text{pathrecurs}
This alternative is preferable in the case in which the third argument is an output variable (resulting in one less valuecell allocated from the global stack), but is inferior for the case in which the third argument is input (leading to an unnecessary construction of the sublist).

**get_list Ai**

Execution of a *get_list* instruction causes the contents of Ai to be dereferenced. If the resulting reference is to an unbound valuecell, a skeletal list will be created in the global stack, *nextarg* will be set to reference the location of the head in the new list (i.e., the address of where the reference to the head must be inserted into the skeletal list), and the mode will be set to *write mode*. If the dereferenced value points to a list, *nextarg* will be set to reference the location of the head in the list, and the mode will be set to *read mode*. If the dereferenced value is to neither an unbound valuecell nor to a list, then backtracking occurs.

**unify_x_variable Ai**

If the machine is in *read mode*, the argument referenced by *nextarg* is inserted into Ai, and *nextarg* is incremented. If the machine is in *write mode*, a new uninstantiated valuecell is created in the global stack. A reference to the new valuecell is stored in Ai and in the argument designated by *nextarg*, and *nextarg* is incremented.

Now let us consider the code generated to represent the second alternative (or clause) in *pathrecurs*. There are a number of new complexities introduced in this part of the procedure. The most interesting is the fact that, in this alternative, it is necessary to allocate an environment containing four valuecells to retain data between subroutine calls, since the argument registers may be modified by a call to a subroutine. These valuecells will be used to hold END, TAIL, NEXT, and EXCLUDELIST (notice that START does not have to be saved through a subroutine invocation). In the code, the labels Y0, Y1, Y2, and Y3 are used to refer to the four valuecells allocated in the environment. This is a naming convention established by Warren.

In the usual case, in which the first, second, and fourth arguments are input arguments (and the third is to be set to the answer), the instructions can be easily interpreted. The following instructions allocate the environment, save the value of END in Y0, construct a skeletal form of the answer in the global stack, and save EXCLUDELIST in Y2. The skeletal form of the answer has the head bound to the value of START, and the tail is bound to an uninstantiated valuecell (for the TAIL). The TAIL will get instantiated by the invoked subroutines.

```
allocate 4 % environment with % 4 valuecells
get_y_variable Y0,A1 % save END in Y0
get_list A2 %
unify_x_local_value A0 % START]
unify_y_variable Y1 % TAIL]
get_y_variable Y2,A3 % save EXCLUDELIST in Y2
```

After processing the input parameters and getting the skeletal form of the answer prepared in the global stack, the following instructions perform the call to *connected/2*. 

```
Note that A0 still references START before the call, and the put_y_variable simply creates an empty (uninstantiated) valuecell in the global stack to represent value chosen for NEXT.

The invocation of not_elof is achieved by the following instructions:

```
put_y_variable Y3,A0
% point A0 at NEXT
put_y_variable Y2,A1
% point A1 at EXCLUDELIST
call not_elof/2,4
% invoke not_elof
```

The code to make the final subroutine invocation is somewhat more complex:

```
put_unsafe_value Y3,A0
% set A0 to NEXT
put_y_value Y0,A1
% set A1 to END
put_y_value Y1,A2
% set A2 to TAIL
put_list A3
% [EXCLUDELIST]
unify_y_local value A0
% NEXT]
unify_y_local value Y2
% EXCLUDELIST]
deallocate
% remove environment
execute pathrecurs/4
% invoke pathrecurs
```

The main complexity involves the fact that the deallocate releases the environment (with the four local valuecells) before the routine is invoked. Thus, the four arguments passed to pathrecurs' must all reference values in the global stack or in the local stack ahead of the deallocated environment. The only danger comes when a variable valuecell in the environment was initialized with an instruction like a put_y_variable instruction. In that case, it is possible for the valuecell to dereference to a valuecell in the local environment. In this case, a put_unsafe_value must be used to make sure that the argument register is set to reference a value that does not occur in the environment (it will "globalize" a valuecell, if necessary; this involves creating a new valuecell in the global stack). This whole topic is somewhat complex and deserves an entire section (which we will supply a bit later).

allocate n

Execution of this instruction causes a new environment to be allocated on the local stack. The current value of both continu and currenv, the continuation address for successful completion and the address of the current environment, are stored in the new environment. The new environment will include n valuecells, which we will refer to as Y0, Y1,...Yn-1. Finally, currenv will be reset to reference the new environment.

In general, an alternative will allocate an environment if it needs one, and it will need one if it is going to call (not execute) another procedure. The environment can be thought of as belonging to a particular alternative, and when we want to refer to this environment when discussing the sequence of code in some alternative, we will call it the local environment.

unify_y_variable Y1

Execution of this instruction can have either of two effects, depending on the mode. If it is executed in read mode, it accesses the next argument designated
by \texttt{nextarg} and binds Yi to the argument. If it is executed in \textit{write mode}, a new uninstantiated valuecell is allocated in the global stack; Yi is bound to the new valuecell, a reference to the newly-created valuecell is placed at the location designated by \texttt{nextarg}, and \texttt{nextarg} is incremented.

\textbf{get\_y\_variable Yi,Aj}

Execution of this instruction binds Yi to the logical formula referenced by Aj.

\textbf{put\_y\_variable Yi,Aj}

Execution of this instruction sets Aj to reference Yi.

\textbf{call procedure.n}

Execution of this instruction causes an invocation of the designated procedure, after "trimming" the current environment to contain only n valuecells. Since not all of the valuecells allocated for an environment always need to be maintained throughout the entire computation for an alternative, valuecells can be trimmed from the end of an environment. This ability depends on the fact that the valuecells are allocated at the end of an environment in the local stack, and the ones that can be released the earliest are kept at the end. Thus, a programmer can carefully organize his use of the variables Y0, Y1,.., to allow a gradual reduction in the size of the environment. In our example, we did not actually trim any valuecells before the complete deallocation of the environment.

The actual transfer of control to the designated procedure is accomplished by first setting \texttt{continst} to reference the instruction immediately following the \texttt{call} (i.e., setting the return address), and then altering \texttt{nextinst} to reference the first instruction in the designated procedure.

\textbf{put\_y\_value Yi,Aj}

Execution of this instruction sets Aj to reference the logical formula to which Yi is bound. If Yi is uninstantiated, but was initialized by a \texttt{put\_y\_variable} instruction, then Aj will be set to reference Yi itself.

\textbf{put\_unsafe\_value Yi,Aj}

First, the value stored in Yi is dereferenced. If the dereferenced value is to a valuecell in the current environment, that valuecell is bound to a new uninstantiated valuecell in the global stack, and Aj is set to reference the new valuecell. Otherwise, the dereferenced value is inserted into Aj (i.e., Aj is set to reference the logical formula given by the dereferenced value of Yi).

\textbf{unify\_y\_value Yi}

If this instruction is executed in \textit{read mode}, the next argument designated by \texttt{nextarg} is unified with the logical formula referenced by Yi (which can, of course, cause backtracking to occur). If it is executed in \textit{write mode}, the value stored in Yi is stored in the argument designated by \texttt{nextarg}. In either case, \texttt{nextarg} is incremented. It is assumed that Yi is bound to a logical formula other than a valuecell in the local stack.

\textbf{unify\_y\_local\_value Yi}

This instruction is identical to \textbf{unify\_y\_value}, except in the case in which the value in Yi dereferences to a valuecell in the local stack. In this case, a new valuecell is allocated in the global stack, and the valuecell in the local stack is bound to the new valuecell.
deallocate

Execution of this instruction deallocates the current environment in the local stack. More precisely, context and current are reset from the values in the environment being released.

5. Temporary and Permanent Variables

After one has coded a number of procedures that were originally represented as Prolog clauses, it becomes clear that it would be useful to have some rules specifying exactly which Prolog variables will require valuecells in the local environment. Warren has supplied the basic method of determining exactly how many local valuecells will be required, and how they should be ordered (i.e., which valuecell should correspond to each of the Prolog variables that requires such a valuecell). In the following discussion, it is assumed that we are talking about a set of variables that occur in a single Prolog clause.

A variable will be called temporary if it fulfills each of the following three conditions:

1. The first occurrence of the variable is in the head literal, in a structure, or in the last goal literal.
2. The variable does not occur in two distinct goal literals.
3. If the variable occurs in the head literal, it does not occur in any goal literal other than the first.

Temporary variables do not require a valuecell in the local environment.

Any variable that occurs in the Prolog clause and is not a temporary variable is called a permanent variable. Each permanent variable will require a valuecell in the local environment. For each permanent variable, we can determine the position in the environment of the corresponding valuecell by considering which goal is the last one in which the variable occurs. The valuecells should be ordered so that those variables that occur in later goal literals have valuecells that occur earlier in the environment. This allows us to trim off valuecells as they no longer remain useful (see next section).

6. Allocate/Deallocate and Trimming

The exact effects of the allocate and deallocate instructions require some amplification. When an allocate n instruction is executed, an environment is allocated on the local stack. This environment will contain two fields used to save values of the registers context and current, followed by n valuecells (which are used for the permanent variables). The previous value of current is stored in the environment, and then current is reset to reference the newly allocated environment.

Consider the case in which a routine r1 allocates an environment, calls routine r2, deallocates the environment, and then executes routine r3. Let us refer to the current environment at entrance to r1 as s1, and the newly-allocated environment as s2. We are now going to focus on what it means to deallocate s2.

There are two cases to consider:

1. If the invocation of r2 created a choice point, we will say that s2 is "protected by a choice point".
2. If no choice points exist in the local stack following s2, then the environment is "not protected by a choice point".

The execution of the deallocate instruction simply resets current and context from s2. If s2 is protected by a choice point, then the memory occupied in the stack is really not reusable (and is not altered in any way by the execution of
the deallocate). For example, should the protecting choice point ever be used to "reset" the machine during backtracking, e2 becomes "active" again (in the sense that the deallocate instruction may again be executed, resetting the current and continue from the values stored in e2). On the other hand, if an unprotected environment is deallocated, then that memory in the stack becomes usable again.

The memory is reclaimed by the execution of a call proc,n instruction. This instruction works as follows:

1. If the current environment is not protected, set the "next available memory" pointer in the stack just past the nth value cell in the environment.
2. Transfer control to proc.

Exercise

4. Consider the following Prolog procedure that can be used to form a list of 2-tuples from corresponding elements of two input lists (or to decompose a list of 2-tuples into two distinct lists):

project([],[],[]).
project([[X,Y]|T],[X|T1],[Y|T2]) :- project(T,T1,T2).

Write the Warren assembler language to implement the procedure.

7. Indexing

We have not yet covered all of the instructions defined for the Warren machine. The most significant class remaining are the instructions to support indexing, a topic that was covered briefly in an earlier section. Now we will explore the topic in detail. We begin by considering the example Warren gave to illustrate the approach; then we will reflect on the general structure of indexes, and finally we will discuss how to handle the problem of dynamic addition and deletion of code.

7.1. An Example

Consider the following Prolog definition of the call procedure, followed by the corresponding Warren assembler language version of the routine:

call(X or Y) :- call(X).
call(X or Y) :- call(Y).
call(trace) :- trace.
call(notrace) :- notrace.
call(nl) :- nl.
call(X) :- builtin(X).
call(X) :- ext(X).
call(call(X)) :- call(X).
call(repeat).
call(repeat) :- call(repeat).
call(true).
call/1  try_me_else          C8a,1
         switch_on_term    C1a,L1,null,L2
L1      switch_on_constant 5,[
                              
                     [trace,C3]  
                     [notrace,C4]  
                     [nl,C5]  
                   ]
L2      switch_on_structure 1,[
                                
                     [or/2,L3]  
                   ]
L3      try                  C1,1
         trust                C2
C1a     try_me_else          C2a,1  % call(  
         get_structure        or/2,A0  % or(  
         unify_x_variable     A0    X,Y) ::  
         execute              call/1  % call(X).
C2a     retry_me_else        C3a,1  % call(  
         get_structure        or/2,A0  % or(  
         unify_void           1    X,  
         unify_x_variable     A0    Y) ::  
         execute              call/1  % call(Y).
C3a     retry_me_else        C4a   % call(  
         get_constant         trace,A0  % trace) ::  
         execute              trace/0  % trace.
C4a     retry_me_else        C5a   % call(  
         get_constant         notrace,A0  % notrace) ::  
         execute              notrace/0  % notrace.
C5a     trust_me_else_fail   C8a,1  % call(  
         get_constant         nl,A0    nl) ::  
         execute              nl/0    nl.
C6a     retry_me_else        C7a   % call(X) ::  
         execute              builtin/1  % builtin(X).
C7a     retry_me_else        L4    % call(X) ::  
         execute              ext/1  % ext(X).
L4      trust_me_else_fail   C8a,L5,null,L7
L5      switch_on_constant  2,[
                                
                     [repeat,L6]  
                     [true,C11]  
                   ]
Before studying the details of the indexing structure, let us consider the new instructions introduced by this example.

switch_on_term \( L_v, L_c, L_l, L_s \)

Execution of this instruction causes a branch which is based on the type of the logical formula referenced by \( A_0 \). If \( A_0 \) references a variable, a branch to \( L_v \) occurs. If \( A_0 \) references a constant, a branch to \( L_c \) occurs. If \( A_0 \) references a list, a branch to \( L_l \) occurs. If \( A_0 \) references a structure, a branch to \( L_s \) occurs. If any operand is "null", backtracking will occur if the corresponding type is detected.

switch_on_constant \( n, \text{table} \)

This instruction generates at compile time a hash table for access to clauses that have designated constants occurring as the first argument. The first argument gives the number of entries allocated in the table. It need not correspond to number of entries in the table. The second specifies a list of 2-tuples. Each tuple gives a constant and the address associated with the constant. Execution of this instruction causes an examination of the constant referenced by \( A_0 \). If the constant occurs in an entry in the hash table, a branch to the corresponding address occurs. Otherwise, backtracking occurs.

switch_on_structure \( n, \text{table} \)

This instruction generates a hash table for access to clauses that have designated functions occurring in the first argument. The first argument gives the number of entries allocated in the table. The second specifies a list of 2-tuples. Each tuple gives a function symbol and the address associated with the symbol. Execution of this instruction causes an examination of the structure referenced
by A0. If the function symbol occurs in an entry in the hash table, a branch to the corresponding address occurs. Otherwise, backtracking occurs.

**try L,n**

This instruction causes a choice point to be created in the local stack. The alternative address stored in the choice point will be the address of the next instruction after the *try*, and *n* argument registers will be stored in the choice point. Finally, *nextinst* is set to L (causing a branch to L to occur).

**trust L**

This instruction causes the current choice point to be discarded (resetting *lastchpt* to the address of the previous choice point before deallocating the current choice point). Then *nextinst* is set to L (causing a branch to L to occur).

**get_structure func,Ai**

Execution of this instruction causes the value in Ai to be dereferenced. If the result is a variable (and, hence, unbound), then a new skeletal structure is constructed in the global stack, and the variable is bound to the new structure. This will cause *nextarg* to be set to where the first argument should be inserted into the skeletal structure, and the machine will continue in *write mode*. If the dereferenced value does not reference a variable, then it is checked to see if it is a structure with *func* as the function symbol. If so, *nextarg* is set to reference the first argument of the structure, and the machine will continue in *read mode*. If the dereferenced value is not a variable nor a structure with a function symbol equal to *func*, then backtracking will occur.

**unify void n**

If the machine is in *read mode*, *nextarg* is incremented past *n* arguments. If it is in *write mode*, *n* new valuecells are created in the global stack, and the next *n* arguments referenced by *nextarg* are set to reference the created valuecells.

### 7.2. The General Indexing Mechanism

The assembler language code for the preceding example illustrates the basic indexing scheme utilized by Warren. Essentially, it is based on the following straightforward scheme.

Suppose that the clauses in a given procedure are $C_1$, $C_2$, ..., $C_m$. These are broken into groups $G_1$, $G_2$, ..., $G_n$. Each group is either a single clause with a variable occurring as the first argument of the head literal, or a set of clauses in which none of the clauses contains a variable as the first argument of the head literal. These groups result in the following generated code:

```assembly
(procedure) try_me_else L2,k
<code for G_1>
L2 retry_me_else L3 <code for G_2>
...
Lm trust_me_else_fail <code for G_m>
```
Nothing really needs to be said about those groups that are single clauses with a variable as the first argument of the head literal. The other groups can be indexed via the following code:

\[
\begin{align*}
\text{Li} & \quad \text{switch\_on\_term} \quad \text{Liv}, \text{Lic}, \text{Lil}, \text{Lis} \\
\text{Lic} & \quad \text{switch\_on\_constant} \quad n, \text{table} \\
& \quad \text{<code for handling multiple clauses for the same constant>} \\
\text{Lis} & \quad \text{switch\_on\_structure} \quad n, \text{table} \\
& \quad \text{<code for handling multiple clauses for the same function symbol>} \\
\text{Lil} & \quad \text{<code for handling multiple clauses that have a list as the first argument>} \\
\text{Liv} & \quad \text{try\_me\_else} \quad \text{Li2}, k \\
\text{Lic1} & \quad \text{<code for the first clause in the group>} \\
\text{Li2} & \quad \text{retry\_me\_else} \quad \text{Li3} \\
\text{Lic2} & \quad \text{<code for the second clause in the group>} \\
& \quad \ldots \\
\text{Lij} & \quad \text{trust\_me\_else\_fail} \\
\text{Licj} & \quad \text{<code for the last clause in the group>}
\end{align*}
\]

Clearly, special cases (such as a single clause in a group) might result in simplified versions of the above structure. The sections of code for handling multiple clauses with a given constant symbol, multiple clauses with a given function symbol, and multiple clauses for a list are as follows:

\[
\begin{align*}
\text{Lim1} & \quad \text{try} \quad \text{<label for first clause>, <number of arguments>} \\
& \quad \text{retry} \quad \text{<label for second clause>} \\
& \quad \text{retry} \quad \text{<label for third clause>} \\
& \quad \ldots \\
& \quad \text{trust} \quad \text{<label for the last clause in the set>}
\end{align*}
\]

This is the first time that we have seen the \textit{retry} instruction. Its definition is basically what one would expect, given the definitions of the \textit{try} and \textit{trust} instructions.

\textbf{retry L}

This instruction causes the next alternative in the current choice point to be set to the instruction immediately following the \textit{retry}, and \textit{nextinst} is set to L (causing a branch to L to occur).
7.3. A Note on Dynamic Modification of Code

Support of the Prolog operations `assert` and `retract` require the dynamic addition, deletion, and modification of code. For example, `assertz` requires either adding a clause to the last group, or altering the last `retry_me_else_fail` to a `retry_me_else_add` and adding a new group.

The question of how to retract clauses is somewhat more difficult. We offer no solution in the case of Prolog implementors; it seems likely that, in the case of Prolog, `assert` and `retract` should be restricted to interpreted (rather than compiled) procedures. However, for other applications in which clauses can be identified by an attached identifier, a limited form of retract can be handled. This is achieved by maintaining a hash table associating a clause identifier with a pointer to the code generated by the clause. The retract is then achieved by altering the code for the clause to a call to the intrinsic `fail`. We realize that the above comments are extremely cursory, and feel that the reader should form his own judgment on the matter.

8. Another Example

In order to offer another illustration of the concepts that we have covered, as well as to introduce a few more instructions, we include another simple example. The assembler code that we present will correspond to the following Prolog code:

```prolog
:- test([1,2,3,4,5,6,7,8,9,10]).

test(LIST) :- display('input list: '),
            display(LIST),
            nl,
            nrev(LIST,REV),
            display('reversed list: '),
            display(REV),
            nl.

nrev([],[]).
nrev([XILO],L) :- nrev(LO,L1), conc(L1,[X],L).

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

Note that the code includes a query, followed by a number of procedures. The query sets up a list of 10 elements (the integers 1 to 10) and invokes the procedure `test` to display the original list, create a new list of the elements in reverse order, and display the resulting list. The Warren assembler language version of this code is as follows:

```assembler
query/0
  put_list A1 % SUB1=[
  unify_constant 10 % 10]
  unify_nil % [10]
put_list A2 % SUB2=[
  unify_constant 9 % 9]
  unify_x_value A1 % SUB1]
put_list A1 % SUB3=[
  unify_constant 8 % 8]
  unify_x_value A2 % SUB2]
```
- 24 -

put_list  A2  % SUB4= [7]
unify_constant  7  % 7]
unify_x_value  A1  % SUB5= [3]
put_list  A1  % SUB5= [3]
unify_constant  6  % 6]
unify_x_value  A2  % SUB4]
put_list  A2  % SUB4]
unify_constant  5  % 5]
unify_x_value  A1  % SUB5]
put_list  A1  % SUB5]
unify_constant  4  % 4]
unify_x_value  A2  % SUB6]
put_list  A2  % SUB6]
unify_constant  3  % 3]
unify_x_value  A1  % SUB7]
put_list  A1  % SUB7]
unify_constant  2  % 2]
unify_x_value  A2  % SUB8]
put_list  A2  % SUB8]
unify_constant  1  % 1]
unify_x_value  A1  % SUB9]
nl/0  % 
execute  test/1

test/1
allocate  2  % test( get_y_variable  Y1,A0  % LIST) :-
put_constant  'input list: ',A0 call_foreign  display/1,2
put_y_value  Y1,A0  % display(LIST), call_foreign  display/1,2
call_foreign  nl/0,2
put_y_value  Y1,A0  % nrev(LIST, put_y_variable  Y0,A1  % REV) call  nrev/2,1
put_constant  'reversed list: ',A0 call_foreign  display/1,1
put_unsafe_value  Y0,A0  % display(REV), call_foreign  display/1,1
deallocate  execute_foreign  nl/0  % nl.
nrev/2  % swich_on_term  Lvl,Lc,Ll,null
Lv  % try_me_else  C2,2
Note the trimming of valuecells that occurs in the procedure for \textit{nrev/2}. Several new instructions are introduced by this example.

\textbf{unify\_constant \ const}

If this instruction is executed in \textit{read mode}, it accesses the argument designated by \textit{nextarg}. It dereferences that argument and checks the dereferenced value. If the dereferenced value is an uninstantiated valuecell, the valuecell is bound to \textit{const}. Otherwise, if the dereferenced value is not the constant \textit{const}, then backtracking occurs. If the machine is in "write mode", a reference to \textit{const} is put into the argument designated by \textit{nextarg}. In any event, if backtracking does not occur, \textit{nextarg} is incremented to the next argument.
put\_constant \ const, Ai
Execution of this instruction puts a reference to the constant \ const \ into Ai.

call\_foreign \ proc, n
This instruction is similar to the call instruction, with the exception that proc is here a "foreign subroutine". That is, proc is coded in a language other than Warren assembler language (usually). This instruction is an extension to Warren's defined instruction set.

execute\_foreign \ proc
This instruction is similar to the execute instruction, with the exception that proc is here a "foreign subroutine". This instruction is an extension to Warren's defined instruction set.

get\_nil \ Ai
Execution of this instruction causes the value in Ai to be dereferenced. If the dereferenced value is to an uninstantiated valuecell, the valuecell will be bound to nil, the empty list. Otherwise, if the dereferenced value is not the empty list, backtracking will occur.

get\_X\_value \ Ai, Aj
Execution of this instruction causes a unification of the two formulas referenced by Ai and Aj. If the unification succeeds, Ai will be altered to reference the fully dereferenced value of the unification. If the formulas cannot be unified, backtracking will occur.

Exercise
5. In a previous section, Warren assembler code for the following 3-clause procedure was given:

\begin{verbatim}
connected(a, b).
connected(a, c).
connected(b, d).
\end{verbatim}

How would you modify the code to utilize indexing on the first argument?

6. Give the Warren assembler that would be used to encode the following Prolog routine:

\begin{verbatim}
connected(a, b).
c:connected(a, c).
:connected(b, d).
connected(X, e) :- incoming(X).
connected(f, X) :- outgoing(X).
connected(g, h).
connected(g, i).
\end{verbatim}

9. Binding One Valuecell to Another
Whenever a point is reached where two logical variables must be unified, one of the two corresponding valuecells is bound to the other. The choice of which valuecell to bind is significant. One should envision the entire global stack as preceding the entire local stack (i.e., the "addresses" of valuecells in the
global stack are lower than those of valuecells in the local stack). Furthermore, the stacks grow into larger addresses (i.e., the address of a newly-allocated valuecell in the global stack will be larger than the address of any previously allocated valuecell in the global stack). It is imperative that, when one valuecell is bound to another, the valuecell with the larger address is instantiated.

The reason for binding the valuecell with the larger address involves the use of the trail, the data structure used to record bindings. As long as a valuecell was allocated after the creation of the last choice point, it is not necessary to record a binding of that valuecell in the trail. If backtracking were to occur, the valuecell would be deallocated; hence, there is no point in retaining the information required to reset it back to an uninstantiated condition. By binding valuecells with larger addresses, the size of the trail can be minimized. It is also the case that valuecells in the local stack are trimmed when the contents can be discarded. There is no problem in trimming a valuecell that is bound to another valuecell that precedes it. However, if the valuecell occurring earlier in the stack were bound to a valuecell being trimmed, a "dangling reference" would result.

10. Unsafe Variables

At this point, we have covered a majority of the instructions in the Warren assembler language. However, it seems appropriate to cover the concept of "unsafe" variable in a bit more detail.

The comments in the preceding section imply that the only valuecells that could be bound to a valuecell \( V \) in the local environment are valuecells that follow \( V \) in the environment. The execution of a `putUnsafeValue` instruction ensures that the referenced formula is not a valuecell in the local environment (which might get trimmed, leaving a dangling reference). Note, however, that the `putUnsafeValue`, in the case in which a new valuecell is created in the global stack, binds the valuecell in the environment to the newly-created valuecell. This is necessary, since the valuecell in the local environment may not be one of the trimmed valuecells (and, hence, may still be legitimately accessed by an invoked subroutine).

This whole topic is actually quite complex. It helps simplify it to remember the following points, which we listed earlier:

1. No valuecell in the global stack can be bound to a formula in the local stack.
2. Valuecells in the local stack must reference either formulas in the global stack or formulas that occur earlier in the local stack.
3. The only formulas that occur in the local stack are valuecells.
4. Structures and lists exist only in the global stack. Since no valuecell in the global stack can be bound to a valuecell in the local stack, no subformulas of a list or structure can be in the local stack.

11. Miscellaneous Instructions Not Yet Covered

Our previous examples utilized most of the instructions in the complete Warren instruction set. However, there remain a few worth mentioning.

`getVariable Ai,Aj`
This instruction copies the contents of \( Aj \) into \( Ai \).
**get_y_value Yi,Aj**

Execution of this instruction causes the formulas referenced by Yi and Aj to be unified. Backtracking will occur if the formulas cannot be unified.

**put_x_variable Ai,Aj**

Execution of this instruction causes a new valuecell to be allocated in the global stack. Then a reference to the new valuecell is inserted into both Ai and Aj.

**put_x_value Ai,Aj**

Execution of this instruction copies the value of Ai into Aj (i.e., it is identical to the `get_x_variable` instruction).

**put-nil Ai**

Execution of this instruction sets Ai to reference "nil", the empty list.

**put-structure func,Ai**

Execution of this instruction causes a skeletal structure to be allocated in the global stack, with the function symbol for the new structure being `func`. Ai is set to reference the new structure, and the machine is set to write mode.

12. **Additional Operations to Support Theorem Proving**

We have covered the opcodes included in the original Warren machine. With these operations, high performance implementations of Prolog have been created. However, none of the instructions discussed up to this point perform an "occurs check" when unifying two formulas. To transfer the advantages in performance achieved with the Warren machine in the context of Prolog implementation to the construction of theorem proving systems, we have included a set of new instructions that perform an occurs check. The following table lists the two sets of instructions supported on our extended Warren machine. The operations in the left column do not include the occurs check; those in the right column do:

<table>
<thead>
<tr>
<th>unify_x_value</th>
<th>o_unify_x_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unify_y_value</td>
<td>o_unify_y_value</td>
</tr>
<tr>
<td>unify_x_local_value</td>
<td>o_unify_x_local_value</td>
</tr>
<tr>
<td>unify_y_local_value</td>
<td>o_unify_y_local_value</td>
</tr>
<tr>
<td>get_x_value</td>
<td>o_get_x_value</td>
</tr>
<tr>
<td>get_y_value</td>
<td>o_get_y_value</td>
</tr>
</tbody>
</table>

The operations that support an occurs check do so only conditionally. That is, we have added a new status indicator, called the `occurs-check-mode`, which can be set to on or off. It is set only during the execution of `call`, and `execute` operations. The indicator can be set to off by specifying `no_occurs` as an optional additional operand on these operations. If the operand is not specified, or if it is specified as any other value, execution of one of the above operations will set the `occurs-check-mode` to on. Those operations that support occurs checks do so only when the `occurs-check-mode` is on.

12.1. **A Theorem-Proving Example in Warren Assembler Language**

To illustrate the utility of these operations, we will consider a standard example from the literature of theorem proving. The problem is to show that any group in which \( x \cdot x = e \) must be commutative. One standard approach to
solving the problem is to show that the following set of clauses is unsatisfiable. In this formulation of the problem, \( p(x,y,z) \) may be thought of as asserting that \( x * y = z \). Although an equality representation of the problem is probably preferable, this formulation does illustrate the points that we wish to convey in this section.

1. \( p(x,e,x) \) e is a right identity
2. \( p(e,x,x) \) e is a left identity
3. \( p(x,g(x),e) \) \( g(x) \) is the right inverse of \( x \)
4. \( p(g(x),x,e) \) \( g(x) \) is the left inverse of \( x \)
5. If \( p(x,y,xy) \) & \( p(y,z,yz) \) & \( p(xy,z,xyz) \) then \( p(x,yz,xyz) \)
6. If \( p(x,y,xy) \) & \( p(y,z,yz) \) & \( p(xy,z,xyz) \) then \( p(xy,z,xyz) \)
7. \( p(x,y,f(x,y)) \) closure
   the operation is well-defined
8. If \( p(x,y,zl) \) & \( p(x,y,z2) \) then \( z1 = z2 \)
   equality axioms
9. \( x = x \)
10. If \( x = y \) then \( y = x \)
11. If \( x = y \) & \( y = z \) then \( x = z \)
    equality substitution for \( g \) and \( f \)
12. If \( x = y \) then \( g(x) = g(y) \)
13. If \( x = y \) then \( f(x,v) = f(y,v) \)
14. If \( x = y \) then \( f(v,x) = f(v,y) \)
    equality substitution for \( p \)
15. If \( x = y \) & \( p(x,v2,v3) \) then \( p(y,v2,v3) \)
16. If \( x = y \) & \( p(v1,x,v3) \) then \( p(v1,y,v3) \)
17. If \( x = y \) & \( p(v1,v2,x) \) then \( p(v1,v2,y) \)
18. \( p(x,x,e) \) special hypothesis
    denial of theorem:
19. \( p(a,b,c) \) There are elements \( a, b, \) and \( c \) such that
20. \( -p(b,a,c) \) \( a * b = c \) but \( b * a \) is not equal to \( c \)

Hyperresolution is one of the more useful inference mechanisms used in classical theorem provers[3,4,5]. In our discussion here, we will be considering the use of hyperresolution with set of support (although it is known that this approach is not refutation complete, it is a widely used and effective strategy). Furthermore, let us confine the discussion to Horn sets, which simply means that each clause can contain at most one positive literal. Normally, an inference rule like hyperresolution is implemented such that a positive unit clause is passed to the inference mechanism. The inference routine must locate a "nucleus" clause (e.g., one of the associativity axioms) which contains a negative literal that can be unified with the specified positive unit. Once such a nucleus is located, the search continues by trying to remove the remaining antecedent literals with other positive unit clauses. Once all of the negative literals have
been removed, a hyperresolvent is inferred (composed of the remaining positive literal).

For purposes of discussion, let us suppose that only axiom 19 of the example is included in the set of support. The following basic steps outline the conversion of a set of clauses intended for interpretation by a theorem prover into Warren assembler language.

1. First, all of the clauses which are axioms that are not in the set-of-support are transformed into a set of "pseudo-Prolog" clauses that are then compiled into Warren assembler language.

2. Inferences proceed by selecting a single clause from the set-of-support. This clause is "asserted", which means that it is added to the code resulting from step 1. That is, if the selected unit were p(a, b, c), then it would be added to the routine p/3. The code given below corresponds to the output of these first two steps.

3. Then, the positive unit p(a, b, c) is converted to an invocation of the routine notp/3 with the given three arguments. Each generated hyperresolvent will be passed to the foreign subroutine newclause/1, which will simplify the clause and perform subsumption tests. If the clause is the null clause, the problem has been solved. Otherwise, if the generated clause is useful (passes screening operations such as forward subsumption, weighting, etc.), it will be added to the set of support.

4. Finally, the given clause is removed from the set of support, and the algorithm returns to step 2.

This extremely cursory overview of the process should allow the reader to understand the following encoding of the clauses which follows. Note that in the comments above each routine we have included a "pseudo-Prolog" version of the routine. The reader is cautioned to think of it only as a loose description. To properly present these routines in a "high-level" representation would require extending the Prolog language to allow the programmer to explicitly control "occurs checking". The reader is particularly cautioned to note that the operator "=" in the comments is not intended to be the built-in operator of Prolog.

```
% notp/3

% notp/3 through the code for C3 comes from axiom 5.
% The code for C4 through C6 comes from axiom 6.
% The code for C7 and C8 comes from axiom 8.
% The code for C9 is from axiom 15.
% The code for C10 is from axiom 18.
% The code for C11 is from axiom 17.
% The code for C12 is from axiom 20.
% notp(X,Y,Z) :- p(X,Y,Z), newclause(p(X,Y,Z)).
% notp(Y,Z) :- p(X,X,Y), p(X,Y,Z), newclause(p(X,Y,Z)).
% notp(X,Y,Z,XYZ) :- p(X,Y,Z), p(Y,Z), newclause(p(X,Y,Z)).
% notp(Y,Z,XYZ) :- p(X,X,Y), p(X,Y,Z), newclause(p(X,Y,Z)).
% notp(X,Y,Z,XYZ) :- p(X,Y,Z), p(Y,Z), newclause(p(X,Y,Z)).
% notp(X,Y,Z,XYZ) :- p(X,Y,Z), p(Y,Z), newclause(p(X,Y,Z)).
% notp(X,Y,Z,XYZ) :- p(X,Y,Z), p(Y,Z), newclause(p(X,Y,Z)).
% notp(X,Y,Z,XYZ) :- p(X,Y,Z), p(Y,Z), newclause(p(X,Y,Z)).
% notp(X,Y,Z,XYZ) :- p(X,Y,Z), p(Y,Z), newclause(p(X,Y,Z)).
% notp(X,Y,Z,XYZ) :- p(X,Y,Z), p(Y,Z), newclause(p(X,Y,Z)).
%```

\% notp(V1,X,V3) :- X = Y, newclause(p(V1,Y,V3)).
\% notp(V1,V2,X) :- X = Y, newclause(p(V1,V2,Y)).
\% notp(b,a,c) :- newclause(null).

\% notp/3  try_me_else
allocate C2a,3 \% create choice point
      5 \% allocate an environment
get_y_variable Y4,A0 \% If p(x, yz) &
get_x_variable A0,A1 \% y, yz) &
get_y_variable Y2,A2 \% xy) &
put_y_variable Y1,A1 \% p(y, yz) &
put_y_variable Y3,A2 \% yz) &
call p/3,5

put_y_value Y4,A0 \% p(x, yz) &
put_unsafe_value Y3,A1 \% yz, yz)
call Y0,A2 \% xy) &

allocate C2
allocate an environment
get_y_variable Y1,A1 \% If p(y, yz) &
get_x_variable A1,A0 \% y, yz) &
get_y_variable Y3,A2 \% yz) &
put_y_variable Y4,A0 \% p(x, yz) &
put_y_variable Y2,A2 \% xy) &
call p/3,5

put_unsafe_value Y4,A0 \% p(x, yz) &
put_y_value Y3,A1 \% yz, yz)
call Y0,A2 \% xy) &

allocate C3
allocate an environment
get_y_variable Y3,A1 \% If p(x, yz) &
get_y_variable Y0,A2 \% xy) &
put_y_variable Y4,A1 \% p(x, yz) &
put_y_variable  Y2,A2  %  xy) &
call  p/3,5

put_unsafe_value  Y4,A0  %  p(y,
put_y_variable  Y1,A1  %  z,
put_y_variable  Y3,A2  %  yz
   call  p/3,3  %  ) then

put_structure  p/3,A0  %  Build new clause
unify_y_local_value  Y2  %  p(xy,
unify_y_local_value  Y1  %  z,
unify_y_value  Y0  %  xyz)
deallocate
execute_foreign  newclause/1

C4a  retry_me_else  C5a  %  update choice point
   allocate  5  %  allocate an environment
get_y_variable  Y2,A0  %  1f  p(x,
get_x_variable  A0,A1  %  y,
get_y_variable  Y4,A2  %  xy) &
put_y_variable  Y3,A1  %  p(y,z,
put_y_variable  Y1,A2  %  yz) &
call  p/3,5

put_y_value  Y4,A0  %  p(xy,
put_unsafe_value  Y3,A1  %  z,
put_y_variable  Y0,A2  %  xyz
   call  p/3,3  %  ) then

put_structure  p/3,A0  %  Build new clause
unify_y_value  Y2  %  p(x,
unify_y_local_value  Y1  %  yz,
unify_y_local_value  Y0  %  xyz)
deallocate
execute_foreign  newclause/1

C5a  retry_me_else  C6a  %  update choice point
   allocate  5  %  allocate an environment
get_y_variable  Y3,A1  %  1f  p(y,z
get_x_variable  A1,A0  %  ,
get_y_variable  Y1,A2  %  yz) &
put_y_variable  Y2,A0  %  p(x,y.
put_y_variable  Y4,A2  %  xy) &
call  p/3,5

put_unsafe_value  Y4,A0  %  p(xy,
put_y_value  Y3,A1  %  z,
put_y_variable  Y0,A2  %  xyz
   call  p/3,3  %  ) then

put_structure  p/3,A0  %  Build new clause
unify_y_local_value  Y2  %  p(x,
unify_y_value Y1 % yz, 
unify_y_local_value Y0 % xyz)
deallocate execute_foreign newclause/1

C6a retry_me_else C7a % update choice point
allocate 5 % allocate an environment
get_y_variable Y0,A2 % If p(xy,z,xyz) &
get_x_variable A2,A0 %
get_y_variable Y3,A1 %

put_y_variable Y2,A0 % p(x,
put_y_variable Y4,A1 % y,xy) &
call p/3,5,no_occurs

put_unsafe_value Y4,A0 % p(y,
put_y_variable Y3,A1 % y,
put_y_variable Y1,A2 % yz
call p/3,3 % ) then

put_structure p/3,A0 % Build new clause
unify_y_local_value Y2 % p(x,
unify_y_local_value Y1 % yz,
unify_y_value Y0 % xyz)
deallocate execute_foreign newclause/1

C7a retry_me_else C8a % update choice point
allocate 2 % allocate an environment
get_y_variable Y1,A2 % If p(x,y,z1) &

put_y_variable Y0,A2 % p(x,y,z2) then
call p/3,2

put_structure =~/2,A0 % Build new clause
unify_y_value Y1 % =~/2(z1,
unify_y_local_value Y0 % z2)
deallocate execute_foreign newclause/1

C8a retry_me_else C9a % update choice point
allocate 2 % allocate an environment
get_y_variable Y0,A2 % If p(x,y,z2) &

put_y_variable Y1,A2 % p(x,y,z1) then
call p/3,2

put_structure =~/2,A0 % Build new clause
unify_y_local_value Y1 % =~/2(z1,
unify_y_value Y0 % z2)
deallocate execute_foreign newclause/1

C9a retry_me_else C10a % update choice point
C9 allocate 3 % allocate an environment
get_y_variable Y1,A1 % If p(x,v2,
get_y_variable Y0,A2 % v3) &
put_y_variable Y2,A1 % =/2(x,y)
call =/2,3
put_structure p/3,A0 % Build new clause
unify_y_local_value Y2 % p(y,
unify_y_value Y1 % v2,
unify_y_value Y0 % v3)
deallocate
eexecute_foreign newclause/1

C10a retry_else_else C11a % update choice point
C10 allocate 3 % allocate an environment
get_y_variable Y2,A0 % If p(v1,x,
get_y_variable Y0,A2 % v3) &
put_x_value A1,A0 % =/2(x,y)
put_y_variable Y1,A1 % =/2(x,y)
call =/2,3
put_structure p/3,A0 % Build new clause
unify_y_value Y2 % p(v1,
unify_y_value Y1 % v2,
unify_y_local_value Y0 % y,
deallocate
eexecute_foreign newclause/1

C11a retry_else_else C12a % update choice point
C11 allocate 3 % allocate an environment
get_y_variable Y2,A0 % If p(v1,
get_y_variable Y1,A1 % v2,x) &
put_x_value A2,A0 % =/2(x,y)
put_y_variable Y0,A1 % =/2(x,y)
call =/2,3
put_structure p/3,A0 % Build new clause
unify_y_value Y2 % p(v1,
unify_y_value Y1 % v2,
unify_y_local_value Y0 % y)
deallocate
eexecute_foreign newclause/1

C12a trust_else Else Fail % delete choice point
c12 get_constant b/0,A0 % If p(b,
get_constant a/0,A1 % a,
get_constant c/0,A2 % c) then
% generate the null clause
put_constant null/0,A0
eexecute_foreign newclause/1
not=/2
not=/2
The code for C14 through C15 comes from axiom 11.
The code for C16 comes from axiom 12.
The code for C17 comes from axiom 13.
The code for C18 comes from axiom 14.
The code for C19 comes from axiom 15.
The code for C20 comes from axiom 18.
The code for C21 comes from axiom 17.

not=(X,Y) :- newclause(Y = X).
not=(X,Y) :- Y = Z, newclause(X = Z).
not=(Y,Z) :- X = Y, newclause(X = Z).
not=(X,Y) :- newclause(g(X) = g(Y)).
not=(X,Y) :- newclause(f(X,V) = f(Y,V)).
not=(X,Y) :- newclause(f(V,X) = f(V,Y)).
not=(X,Y) :- p(X,V2,V3), newclause(p(V1,Y,V3)).
not=(X,Y) :- p(V1,X,V3), newclause(p(V1,Y,V3)).
not=(X,Y) :- p(V1,V2,X), newclause(p(V1,V2,Y)).

not=/2 try_me_else
put_x_value C14a,2 % create choice point
put x_value A0,A2 % save reference to x
put_structure =/2,A0 % build =/2(x,y)
unify_x_value A1 % x
unify_x_value A2 % y
execute_foreign
newclause/1

C14
allocate
get_y_variable Y1,A0 % allocate an environment

put_x_value A1,A0 % =/2(y,
put_y_variable Y0,A1 % z)
call =/2,2

put_structure =/2,A0 % build =/2(x,
unify_y_value Y1 % x
unify_y_local_value Y0 % z)
deallocate
execute_foreign
newclause/1

C15
allocate
get_y_variable Y0,A1 % allocate an environment

put_x_value A0,A1 % =/2(x,y)
put_y_variable Y1,A0 % x
call =/2,2

put_structure =/2,A0 % build =/2(y,
unify_y_local_value Y1 % x,
unify_y_value...newclause/1

desallocate

execute_foreign

C18a retry_if_else...newclause/1
C16 put_structure...unify_y_value

executeforeign

C17a retry_if_else...newclause/1
C17 put_structure...unify_y_value

executeforeign

C18a retry_if_else...newclause/1
C18 put_structure...unify_y_value

executeforeign

C19a retry_if_else...newclause/1
C19 allocate

call

put_structure...unify_y_value
deallocate

execute_foreign

C20a % update choice point
C17a % allocate an environment

allocate

call

put_structure...unify_y_value
deallocate

execute_foreign

C20a % update choice point
C17a % allocate an environment

allocate
C20a retry_me_else
C20 allocate
get_y_variable
put_y_variable
put_y_variable
put_y_variable
put_x_value
put_y_variable
put_y_variable
call
put_structure
unify_y_local_value
unify_y_local_value
unify_y_local_value
deallocate
execute_foreign
C21a % update choice point
C21 allocate
get_y_variable
put_x_value
put_y_variable
put_y_variable
put_y_variable
call
put_structure
unify_y_local_value
unify_y_local_value
unify_y_local_value
deallocate
execute_foreign
C21a % update choice point
C21 allocate
get_y_variable
put_x_value
put_y_variable
put_y_variable
put_y_variable
call
put_structure
unify_y_local_value
unify_y_local_value
unify_y_local_value
deallocate
execute_foreign
newclause/1
% newclause/1

The code for C22 comes from axiom 1.
The code for C23 comes from axiom 2.
The code for C24 comes from axiom 3.
The code for C25 comes from axiom 4.
The code for C26 comes from axiom 5.
The code for C27 comes from axiom 6.
The code for C28 comes from axiom 7.
The code for C29 comes from axiom 8.
The code for C30 comes from axiom 9.

p(X,e,X).
p(e,X,X).
p(X,g(X),e).
p(g(X),X,e).
p(X,Y,f(X,Y)).
p(X,X,e).
p(a,b,c).

p/3 switch_on_term C22a,L1,null,L2
% The following routine encodes axiom 9.
We are aware that minor modifications to an existing dialect of logic programming would allow a fairly natural intermediate encoding (i.e., would represent the axioms at a level somewhere between the original axioms and the Warren assembler language). We are interested in such research, but have no concrete proposals to make at this time.

### 12.2. Half-Match Unification

Warren's original instruction set can also be extended to support "half-match" unification (i.e., unification in which only variables in one of the two terms can be instantiated). These opcodes are extremely useful for compiling code that does subsumption checks or demodulation (see[2] for descriptions of these operations). The added opcodes are as follows:

- `m_get_x_value`  
- `m_get_y_value`  
- `m_get_constant`  
- `m_get_nil`  
- `m_get_structure`  
- `m_get_list`  
- `m_unify_x_value`  
- `m_unify_y_value`  
- `m_unify_constant`  
- `m_unify_nil`  

These opcodes are all designed to verify that two formulas match identically, rather than testing to determine whether or not the formulas can be unified. For example,

\[ m_get_list \ A2 \]

verifies that A2 references a formula which is a nonempty list (setting `nextarg` to allow subsequent processing in read mode to inspect the elements in the list).

The above discussion is necessarily brief. However, we do believe that it can be used as a basis for arguing that the Warren abstract machine will dramatically impact the attainable inference rates for theorem-proving systems.

### 13. A Comparison with Warren's Original Description

The reader should be well aware that the description of the Warren Abstract Machine offered in this document differs in substantial aspects from both the description offered in Warren's original paper[1] and the implementation in Quintus Prolog. Since this document is designed both as a tutorial on the abstract machine and as a reference for the portable implementation produced at Argonne National Laboratory, some of the differences should be discussed. The most outstanding differences between the description in this document (and the implementation based upon it) and that in Warren's original report are as follows:

1. In Warren's description, the notion of *temporary variable* plays a much larger conceptual role. He carefully distinguishes those instances in which an argument register is to be thought of as playing the role of a
temporary variable. He introduces two conceptually distinct register sets, the X registers and the A registers. The fact that the sets are identical is viewed as an implementation optimization. We have chosen to ignore the distinction. Rather, we view the use of an argument register to play the role of a temporary variable as a convenient conceptual approach towards implementing Prolog on the Warren Abstract Machine, but do not view it as a part of the description of the actual machine. In early versions of our actual implementation, the assembler enforced the Warren distinction. In later versions, the A and X notations are completely interchangeable.

2. In our implementation, logical formulas in the local and global stacks are tagged objects, while an "argument" (i.e., the contents of an argument register, an argument of a list or an argument of a structure) is a reference to such a tagged object. In Warren's description of the machine, all references to formulas contain a copy of the tag of the object which is being referenced (actually, this is not precisely accurate, but does convey the essence of the situation). This allows many comparisons and checks to take place without forcing an access to the referenced object (to inspect the tag). An implementation for a machine in which a single word can contain both the pointer (the reference) and the tag will benefit by utilizing Warren's optimization. However, since our implementation is designed for portability, we could not make the assumption that adding the tag would still allow an argument to reside in a "machine register". In this case, the added tags may take substantially more memory and actually slow down execution (since the arguments cannot easily be coerced into registers).

3. We have added the instructions required to support the "occurs check" and "half matches" (i.e., unifications in which only one of the two formulas can be instantiated).

4. We have utilized different register names (see below).

As an aid to the reader when he compares this document against Warren's original report we have prepared the following table listing differences in the naming of machine registers:
14. Performance

A few final comments on performance are in order. The abstract machine described in this document has been implemented in C and ported to a number of machines. Executing on a VAX 11/780 it attains between 5K and 6K lips (logical inferences per second) on a determinate concatenate benchmark[5]. Quintus Prolog attains roughly four times this execution rate. The factors which contribute to this difference are as follows:

1. Warren utilizes a modified instruction set which, although based directly on the ideas described in this document, includes features that substantially improve performance.
2. Our implementation is done in C for portability.
3. Our data structures are a generalization of Warren's, which allow an extension of the machine to support opcodes for coordination of multiple processes working simultaneously. In particular, the machine is being extended to support both OR-parallelism and determinate AND-parallelism. The decision to create an implementation that would support these extensions did introduce some overhead into the implementation for uniprocessors.

The factors are listed in decreasing order of significance. We believe that the first factor could be eliminated with some effort (by modifying the instruction set in much the way Warren has in Quintus Prolog), producing an implementation that was between one and a half and two times as fast as the current implementation. We lose some execution speed due to having coded in C rather than assembler language. We estimate the loss to be less than a factor of two. In our opinion, the last factor contributes very little to the overall difference in speed. It introduces, perhaps, a 10% overhead into the execution rate. All of these estimates are offered only as very crude approximations.

The fact that a factor of two is attainable by alteration of the instruction set may well lead the reader to reflect upon the value of learning the version

<table>
<thead>
<tr>
<th>Warren's Description</th>
<th>Our Name</th>
<th>Warren's Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Pointer</td>
<td>neztinst</td>
<td>P</td>
</tr>
<tr>
<td>Continuation Pointer</td>
<td>continst</td>
<td>CP</td>
</tr>
<tr>
<td>Last Environment</td>
<td>currenv</td>
<td>E</td>
</tr>
<tr>
<td>Last Choice Point</td>
<td>lastchkpt</td>
<td>B</td>
</tr>
<tr>
<td>Top of Trail</td>
<td>TPOS</td>
<td>TR</td>
</tr>
<tr>
<td>Top of Heap</td>
<td>GPOS</td>
<td>H</td>
</tr>
<tr>
<td>Structure Pointer</td>
<td>neztarg</td>
<td>S</td>
</tr>
</tbody>
</table>
described in this document. It is our belief that the differences between the instruction set described in this document and the more optimal one used in Quintus Prolog can be "hidden" by a somewhat more complex assembler. That is, a program coded in Warren assembler language as described in this document could be assembled into a somewhat extended set of machine operations to attain the added performance. It is quite probable that we will at some future date extend our interpreter to handle the extended instruction set; however, our intent is that this should not require recoding of any programs written in Warren assembler. Hence, we consider it important that those researchers who choose to compile higher-level languages for use on our interpreter make the target of compilation Warren assembler language (rather than the actual object encoding of the machine language supported by the current interpreter).

15. Summary

We have attempted in this document to convey the basic ideas behind the Warren machine. We believe that the machine represents a substantial breakthrough in the design of high-performance inference engines and may well form the basis of many future advanced logic systems.

Acknowledgement

We are grateful for the comments we received from both David H. D. Warren and Fernando Pereira. Our intellectual debt to the achievements of David Warren cannot be overstated. However, neither individual has edited the contents of this document, and the reader should view the original definition of the Warren machine[1] as the basic reference work.
Appendix A
The Warren Abstract Machine Instruction Set

allocate n
Execution of this instruction causes a new environment to be allocated on the local stack. The current value of both continst and current, the continuation address for successful completion and the address of the current environment, are stored in the new environment. The new environment will include n valuecells, which we will refer to as Y0, Y1,...Yn-1. Finally, current will be reset to reference the new environment.

call_foreign proc,n
This instruction is similar to the call instruction, with the exception that proc is here a "foreign subroutine". That is, proc is coded in a language other than Warren assembler (usually). This instruction is an extension to Warren's defined instruction set.

call procedure,n
Execution of this instruction causes an invocation of the designated procedure, after "trimming" the current environment to contain only n valuecells. Since not all of the valuecells allocated for an environment always need to be maintained throughout the entire computation for an alternative, valuecells can be trimmed from the end of an environment. This ability depends on the fact that the valuecells are allocated at the end of an environment in the local stack, and the ones that can be released the earliest are kept at the end. Thus, a programmer can carefully organize his use of the Y0, Y1,... to allow a gradual reduction in the size of the environment. In our example, we did not actually trim any valuecells before the complete deallocation of the environment.

The actual transfer of control to the designated procedure is accomplished by first setting continst to reference the instruction immediately following the call (i.e., setting the return address), and then altering neztinst to reference the first instruction in the designated procedure.

deallocate
Execution of this instruction deallocates the current environment in the local stack. More precisely, continst and current are reset from the values in the environment being released.

execute procedure
This instruction simply resets neztinst to reference the first instruction in the designated procedure. Thus, when control is returned from the executed procedure, it will not return to the procedure in which the execute occurred, but rather to the caller of the procedure that issued the execute. It is used only when the last subroutine in a procedure is invoked.

execute_foreign proc
This instruction is similar to the execute instruction, with the exception that proc is here a "foreign subroutine". This instruction is an extension to Warren's defined instruction set.
The *get_constant* instruction takes two operands. The first designates a constant, and the second an argument register. The instruction attempts to "match" the constant with the incoming argument. If the argument register references an unbound variable, the valuecell will be bound to the designated constant. If the argument register references the same constant, no action occurs. In either of these cases, the *get_constant* succeeds and the next instruction will be the one immediately following the *get_constant*. If the match does not succeed, backtracking will occur.

*get_list* \( Ai \)

Execution of a *get_list* instruction causes the contents of \( Ai \) to be dereferenced. If the resulting reference is to an unbound valuecell, a skeletal list will be created in the global stack, \( \text{nextarg} \) will be set to reference the location of the head in the new list (i.e., the address of where the reference to the head must be inserted into the skeletal list), and the mode will be set to \textit{write mode}. If the dereferenced value points to a list, \( \text{nextarg} \) will be set to reference the location of the head in the list, and the mode will be set to \textit{read mode}. If the dereferenced value is to neither an unbound valuecell nor to a list, then backtracking occurs.

*get_nil* \( Ai \)

Execution of this instruction causes the value in \( Ai \) to be dereferenced. If the dereferenced value is to an uninstantiated valuecell, the valuecell will be bound to \textit{nil}, the empty list. Otherwise, if the dereferenced value is not the empty list, backtracking will occur.

*get_structure* \( \text{func}, Ai \)

Execution of this instruction causes the value in \( Ai \) to be dereferenced. If the result is a variable (and, hence, unbound), then a new skeletal structure is constructed in the global stack, and the variable is bound to the new structure. This will cause \( \text{nextarg} \) to be set to where the first argument should be inserted into the skeletal structure, and the machine will continue in \textit{write mode}. If the dereferenced value does not reference a variable, then it is checked to see if it is a structure with \textit{func} as the function symbol. If so, \( \text{nextarg} \) is set to reference the first argument of the structure, and the machine will continue in \textit{read mode}. If the dereferenced value is neither a variable nor a structure with a function symbol equal to \textit{func}, then backtracking will occur.

*get_value* \( Al, Aj \)

Execution of this instruction causes a unification of the two formulas referenced by \( Al \) and \( Aj \). If the unification succeeds, \( Al \) will be altered to reference the fully dereferenced value of the unification. If the formulas cannot be unified, backtracking will occur.

*get_variable* \( Ai, Aj \)

This instruction copies the contents of \( Aj \) into \( Ai \).

*get_value* \( Yi, Aj \)

Execution of this instruction causes the formulas referenced by \( Yi \) and \( Aj \) to be unified. Backtracking will occur, if the formulas cannot be unified.
get_y_variable Yi,Aj
Execution of this instruction binds Yi to the logical formula referenced by Aj.

proceed
The proceed instruction is used to return from an alternative that did not require invoking a subroutine (i.e., an alternative represented by a unit clause). The next instruction executed will be designated by the contents of the continst register.

put_constant const,Ai
Execution of this instruction puts a reference to the constant const into Ai.

put_list Ai
Execution of this instruction causes a skeletal list (that is, a list in which neither the head nor the tail has yet been defined) to be created in the global stack. The argument register Ai is set to reference the skeletal list. Furthermore, a register called neztarg is set to reference the spot in the list into which a reference to the head of the list being constructed should be stored. The register neztarg is used only when processing lists and structures and points to the next argument (in a list, the head or tail) to be processed. Finally, the machine is put into write mode.

put_nil Ai
Execution of this instruction sets Ai to reference "nil", the empty list.

put_structure func,Ai
Execution of this instruction causes a skeletal structure to be allocated in the global stack, with the function symbol for the new structure being func. Ai is set to reference the new structure, and the machine is set to write mode.

put_unsafe_value Yi,Aj
First, the value stored in Yi is dereferenced. If the dereferenced value is to a valuecell in the current environment, that valuecell is bound to a new uninstantiated valuecell in the global stack, and Aj is set to reference the new valuecell. Otherwise, the dereferenced value is inserted into Aj (i.e., Aj is set to reference the logical formula given by the dereferenced value of Yi).

put_x_value Ai,Aj
Execution of this instruction copies the value off Ai into Aj (i.e., it is identical to the get_x_variable instruction).

put_y_variable Ai,Aj
Execution of this instruction causes a new valuecell to be allocated in the global stack. Then a reference to the new valuecell is inserted into both Ai and Aj.

put_y_value Yi,Aj
Execution of this instruction sets Aj to reference the logical formula to which Yi is bound. If Yi is uninstantiated, but was initialized by a put_y_variable instruction, then Aj will be set to reference Yi itself.
put_y_variable Yl,Aj
Execution of this instruction sets Aj to reference Yl.

retry L
This instruction causes the next alternative in the current choice point to be set to the instruction immediately following the retry, and nextinst is set to L (causing a branch to L to occur).

retry_me_else label
In a procedure that contains several alternatives, this instruction must precede the code for all but the first and last alternatives (i.e., it precedes the middle alternatives). It causes the "next alternative" in the choice point to be set at the address given as its operand. Execution continues with the instruction immediately following the retry_me_else.

switch_on_constant n,table
This instruction generates a hash table for access to clauses that have designated constants occurring as the first argument. The first argument gives the number of entries allocated in the table. The second specifies a list of 2-tuples. Each tuple gives a constant and the address associated with the constant. Execution of this instruction causes an examination of the constant referenced by A0. If the constant occurs in an entry in the hash table, a branch to the corresponding address occurs. Otherwise, backtracking occurs.

switch_on_structure n,table
This instruction generates a hash table for access to clauses that have designated functions occurring in the first argument. The first argument gives the number of entries allocated in the table. The second specifies a list of 2-tuples. Each tuple gives a function symbol and the address associated with the symbol. Execution of this instruction causes an examination of the structure referenced by A0. If the function symbol occurs in an entry in the hash table, a branch to the corresponding address occurs. Otherwise, backtracking occurs.

switch_on_term Lv,Lc,Ll,ls
Execution of this instruction causes a branch which is based on the type of the logical formula referenced by A0. If A0 references a variable, a branch to Lv occurs. If A0 references a constant, a branch to Lc occurs. If A0 references a list, a branch to Ll occurs. If A0 references a structure, a branch to Ls occurs. If any operand is "null", backtracking will occur if the corresponding type is detected.

trust L
This instruction causes the current choice point to be discarded (resetting lastchpt to the address of the previous choice point before deallocating the current choice point). Then nextinst is set to L (causing a branch to L to occur).

trust_me_else fail
In a procedure that contains several alternatives, this instruction should precede the last alternative. It causes the current choice point to be deallocated from the local stack (since there are no more alternatives to consider). Execution continues with the instruction immediately following the trust_me_else_fail.
try Ln
This instruction causes a choice point to be created in the local stack. The alternative address stored in the choice point will be the address of the next instruction after the try, and n argument registers will be stored in the choice point. Finally, nextinst is set to L (causing a branch to L to occur).

try_me_else label,n
This instruction is used only in procedures which include multiple alternatives. It precedes the code for the first alternative. It causes a choice point to be created in the local stack and sets the next alternative (stored in the choice point) to the address given as its operand. The choice point will save values for the first n argument registers. Execution continues with the instruction immediately following the try_me_else.

unify_constant const
If this instruction is executed in read mode, it accesses the argument designated by nextarg. It dereferences that argument and checks the dereferenced value. If the dereferenced value is an uninstantiated valuecell, the valuecell is bound to const. Otherwise, if the dereferenced value is not the constant const, then backtracking occurs. If the machine is in "write mode", a reference to const is put into the argument designated by nextarg. In any event, if backtracking does not occur, nextarg is incremented to the next argument.

unify_nil
This instruction is identical to unify_constant, except that the constant in this case is the empty list.

unify_void n
If the machine is in read mode, nextarg is incremented past n arguments. If it is in write mode, n new valuecells are created in the global stack, and the next n arguments referenced by nextarg are set to reference the created valuecells.

unify_x_local_value Ai
This instruction is identical to unify_x_value, except in the case in which the value in Ai dereferences to a valuecell in the local stack. In this case, a new valuecell is allocated in the global stack, the valuecell in the local stack is bound to the new valuecell, and Ai is set to reference the new valuecell in the global stack.

unify_x_value Ai
If the machine is in write mode the reference in Ai is pushed into the global stack at the location specified by nextarg, and nextarg is incremented (to the next argument in the structure or list being built). If the machine is in read mode, then the argument designated by nextarg is unified with the logical formula referenced by Ai. If the unification fails, backtracking occurs. Otherwise, the dereferenced result is put into Ai. That is, if Ai referenced anything except a valuecell, Ai will not be altered. On the other hand, if Ai referenced a valuecell that is bound to another logical formula, Ai will be altered to reference the formula to which the valuecell was instantiated. If the valuecell is bound to another instantiated valuecell, the chain of valuecells is examined, and Ai is altered to reference either an unbound valuecell or a formula which is not a variable (i.e., Ai is set to the image found by continuing down the chain of bound valuecells).
This process of locating the ultimate image of a reference is called dereferencing. If the unification is successful, nextarg is incremented.

**unify_x_variable Ai**

If the machine is in read mode, the argument referenced by nextarg is inserted into Ai, and nextarg is incremented. If the machine is in write mode, a new uninstantiated valuecell is created in the global stack. A reference to the new valuecell is stored in Ai and in the argument designated by nextarg, and nextarg is incremented.

**unify_y_local_value Yi**

This instruction is identical to unify y value, except in the case in which the value in Yi dereferences to a valuecell in the local stack. In this case, a new valuecell is allocated in the global stack, and the valuecell in the local stack is bound to the new valuecell.

**unify_y_value Yi**

If this instruction is executed in read mode, the next argument designated by nextarg is unified with the logical formula referenced by Yi (which can, of course, cause backtracking to occur). If it is executed in write mode, the value stored in Yi is stored in the argument designated by nextarg. In either case, nextarg is incremented. It is assumed that Yi is bound to a logical formula other than a valuecell in the local stack.

**unify_y_variable Yi**

Execution of this instruction can have either of two effects, depending on the mode. If it is executed in read mode, it accesses the next argument designated by nextarg and binds Yi to the argument. If it is executed in write mode, a new uninstantiated valuecell is allocated in the global stack, Yi is bound to the new valuecell, a reference to the newly-created valuecell is placed at the location designated by nextarg, and nextarg is incremented.

**References**

Answers to Exercises

1.

\begin{align*}
&\text{get\_constant} & 2, A0 \\
&\text{get\_constant} & 17, A1 \\
&\text{get\_constant} & 2400, A2 \\
&\text{proceed}
\end{align*}

2.

\begin{align*}
&\text{try\_me\_else} & C1,3 \\
&\text{get\_constant} & 2, A0 \\
&\text{get\_constant} & 17, A1 \\
&\text{get\_constant} & 2400, A2 \\
&\text{proceed}
\end{align*}

\begin{align*}
&C1: \text{trust\_me\_else\_fail} \\
&\text{get\_constant} & 2, A0 \\
&\text{get\_constant} & 18, A1 \\
&\text{get\_constant} & 1200, A2 \\
&\text{proceed}
\end{align*}

3.

\begin{align*}
&\text{try\_me\_else} & C1,3 \\
&\text{get\_constant} & 2, A0 \\
&\text{get\_constant} & 17, A1 \\
&\text{get\_constant} & 2400, A2 \\
&\text{proceed}
\end{align*}

\begin{align*}
&C1: \text{retry\_me\_else} & C2 \\
&\text{get\_constant} & 2, A0 \\
&\text{get\_constant} & 18, A1 \\
&\text{get\_constant} & 1200, A2 \\
&\text{proceed}
\end{align*}

\begin{align*}
&C2: \text{retry\_me\_else} & C3 \\
&\text{get\_constant} & 1, A0 \\
&\text{get\_constant} & 17, A1 \\
&\text{get\_constant} & 1200, A2 \\
&\text{proceed}
\end{align*}

\begin{align*}
&C3: \text{trust\_me\_else\_fail} \\
&\text{get\_constant} & 2, A0 \\
&\text{get\_constant} & 16, A1 \\
&\text{get\_constant} & 300, A2 \\
&\text{proceed}
\end{align*}

4.

\begin{align*}
&\text{project/3} \text{ try\_me\_else} & 1.1, 3 \\
&\text{get\_nil} & A0 \\
&\text{get\_nil} & A1
\end{align*}
get_nil proceed

L1 trust_me_else_fail

get_list A2 % [[X|Y]|T], nextarg->[X,Y]
unify_x_variable A0 % A4->[X,Y]
unify_x_variable A4 % A5->T
get_list A4 % [X,Y], nextarg->X
unify_x_variable A0 % A6->X
unify_x_variable A7 % A7->[Y,nil]
get_list A7 % nextarg->Y
unify_x_variable A8 % A8->Y
unify_nil

get_list A1 % [X|T1], nextarg->X
unify_x_value A6 % match X
unify_x_variable A9 % A9->T1

get_list A2 % [Y|T2], nextarg->Y
unify_x_value A8 % match Y
unify_x_variable A10 % A10->T2

put_x_value A5,A0 % A0=T
put_x_value A9,A1 % A1=T1
put_x_value A10,A2 % A2=T2
execute project/3 %recurse

5.

connected/2:

switch_on_term Cv,Cc,fail,fail

Cc: switch_on_constant [
    [a,Cca],
    [b,Ccb]
]

Cv: try_me_else Cva2,2
get_constant a,A0
Ccb1: get_constant b,A1
 proceed

Cva2: retry_me_else Cvbl
get_constant a,A0
Ccc1: get_constant c,A1
 proceed

Cvbl: trust_me_else_fail
get_constant b,A0
Ccb: get_constant d,A1
 proceed
8.

\%connected(a,b)  
\%connected(a,c)  
\%connected(b,d)  
\%connected(X,e):- incoming(X)  
\%connected(f,X):- outgoing(X)  
\%connected(g,h)  
\%connected(g,i)

\% The various groups the code divides into are set out above. The 
\% action of the indexing process directs control to the various groups 
\% according to the category of the first parameter.

connected/2:
   switch_on_term  
   Cv0,Cc,Group2,Group2

% code for variable first argument starts here

Cv0:  
   try_me_else  
   get_constant  
   a,A0

% Start of Group 1 for switch_on_constant since it is already known 
% A0=a if control passes to Cca1

Cca1:  
   get_constant  
   b,A1  
   proceed  
   \%unified with (a,b)

Cv1:  
   retry_me_else  
   get_constant  
   a,A0

Cca2:  
   get_constant  
   c,A1  
   proceed  
   \%(a,c)

Cv2:  
   retry_me_else  
   get_constant  
   b,A0

% code for A0=b from switch_on_const

Ccb:  
   get_constant  
   d,A1  
   proceed  
   \%(b,d)

% Start of Group 2

Cv3:  
   retry_me_else  
   Cv4

% Only one clause in Group 2 so no need for try/retry/trust. Therefore 
% Group2 labels a statement in the code for variable arguments.
Group2: get_constant 
execute 

Cv4: retry_else 
get_constant 

% Start of Group 3 
% code for A0=f from switch_on_const 

Ccf: get_variable 
execute 

Cv5: retry_else 
get_constant 

Ccgl: get_constant 
proceed 

Cv3: trust_else_fail 
get_constant 

Cc2: get_constant 
proceed 

% code for constant arguments goes here 
% it must transfer control among the various groups. 

Cc: try 
retry 
trust 

Group1: switch_on_constant 

Group3: switch_on_constant 

% try/retry/trust sequences for (a,b), (a,c) and (g,h), (g,i) 
% in Group1 and Group3 Single clauses need no try/retry/trust 
% so that Ccb and Ccf label clauses in the main code, i.e. 
% (b,d) and (f,X). 

% clauses (a,...) 

Cca: try 
trust 

% clauses (g,...) 

Ccgl: try 
trust 

Cca1,2 % (a,b) 
Cca2 % (a,c) 

Ccgl,2 % (g,h) 
Ccgl2 % (g,i)
Distribution for ANL-84-84

Internal:

J. R. Gabriel (10)
K. L. Kliewer
A. B. Krisciunas
T. Lindholm (10)
E. L. Lusk (20)
P. C. Messina
R. A. Overbeek (40)
D. M. Pablis
T. M. Woods (2)
G. W. Pieper

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (6)

External:

DOE-TIC, for distribution per UC-32 (173)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. L. Bona, U. Chicago
T. L. Brown, U. of Illinois, Urbana
S. Gerhart, Wang Institute, Tynsboro, MA
G. H. Golub, Stanford U.
W. C. Lynch, Xerox Corp., Palo Alto
J. A. Nohel, U. of Wisconsin, Madison
M. F. Wheeler, Rice U.
D. Austin, ER DOE
J. Greenberg, ER-DOE
G. Michael, LLL
C. E. Oliver, ER-DOE