# Compiling Prolog to Forth

**Article** · January 1987

CITATIONS
2

READS
697

**1 author:**

Lou Odette
Singapore-MIT Alliance
**14** PUBLICATIONS **632** CITATIONS

SEE PROFILE

# Compiling Prolog to Forth

*L. L. Odette*

*Applied Expert Systems, Inc.*
*5 Cambridge Center*
*Cambridge, MA 02142*

## *Abstract*

The fact that the focus of a Prolog computation is the structure of the program leads directly to a view of a Prolog compiler as a procedure that takes a collection of Prolog clauses and produces a description of their structure that just happens to be executable. Forth lends itself naturally to the description of both structures and processes. In fact, some hold that Forth programming involves creating the parts of speech required to describe an application. This article proposes that for this reason, Forth is a very good language for prototyping Prolog compilers. A simple object language for a Prolog to Forth compiler is presented and discussed.

## *Introduction*

A narrow definition of logic is the study of the arguments valid by virtue of their structure. Having taken this view, a rule language (mechanical theorem prover) needs two elements: a validation process and an internal representation of the argument structure. Rule languages can be distinguished on the basis of how much structure they admit. Expert-2 [PAR84], for example, is a mechanical theorem prover for propositional logic where complex arguments are described only in terms of the atomic propositions that make them up. The requirements of the internal representation of an atomic proposition are met by a token for the proposition (e.g., a pointer to the string representing the proposition's text). Prolog, on the other hand, is a mechanical theorem prover for predicate logic, which attaches significance to the internal structure of the atomic propositions: for example, the predicate name and the number and structure of its formal parameters.

It follows that an interpreter for Prolog must run its validation process over more complex data structures than those used by an interpreter for propositional logic. This added complexity tends to limit the performance of Prolog interpreters, certainly relative to interpreters for propositional languages. The thrust of Prolog compilation is to combine the validation process and the clause structure, so that the internal representation of each atomic proposition *is* the program that realizes the validation process over the proposition. In essence, a compiled Prolog clause is an executable description of the clause, which is why Forth is an ideal language for implementing and experimenting with Prolog compilers.

This paper introduces a set of Forth words which form the basis of a Prolog Virtual Machine (PVM). The instructions of the virtual machine are of two types: those that alter the flow of control and those that denote the structures in Prolog clauses. Compilation to the virtual machine instructions becomes a simple matter of composing a description of the clause, which can easily be done by hand. Implementation of the virtual machine is a straightforward Forth programming task.

The compiler technology presented here is based on the simple compiler described by Bowen, Byrd, and Clocksin [BOW83]. Code for the compiler (in Prolog) is given in an appendix, as is the Forth code for the virtual machine. The compiler code may be of use to Forth programmers

interested in building compilers in Prolog. The Forth code may be of use to anyone interested in incorporating Prolog in Forth applications or experimenting with extensions of the Prolog language. The elegance of the Forth solution to compiling Prolog should be of interest to both Forth and non-Forth programmers alike.

## *Introduction to Prolog*

Prolog is a simple language with a straightforward syntax and program structure (Figure 1).

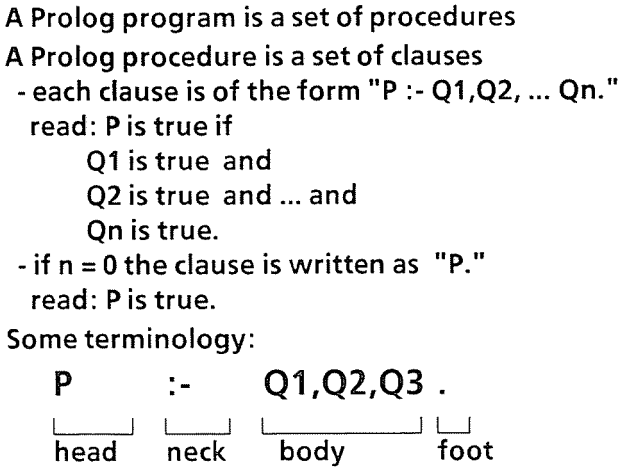



Figure 1. Prolog at a Glance (I).

Its declarative semantics is also straightforward. Each procedure represents the definition of a predicate. For example, the sex of individuals may be specified by the predicates male and female, as defined by the Prolog clauses:

```
male(isaac).
male(lot).
female(milcah).
```

Predicate definitions may be conditional, as in the following clauses:

```
son(X,Y) :- parent(Y,X),male(X).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
```

The first clause is read as "X is a son of Y if Y is a parent of X and X is male." The second clause is read as "X is a grandparent of Y if X is a parent of Z and Z is a parent of Y." The terms **X**, **Y**, and **Z** in these definitions are logical variables, meaning they reference some unknown individual. The scope of a variable reference is the clause it is used in.

What is unusual about Prolog is its procedural semantics—in particular, the search mechanism underlying procedure invocation (which may result in backtracking) and the means for passing information between procedures via unification (pattern matching). Prolog procedures execute much like Forth or any other conventional language with the exception that any procedure call could possibly invoke more than one procedure or even none at all. The Prolog machinery needs to search through the candidate procedures. One way to visualize Prolog procedure execution is as a search tree, or a proof tree in the case of successful search (Figure 2).

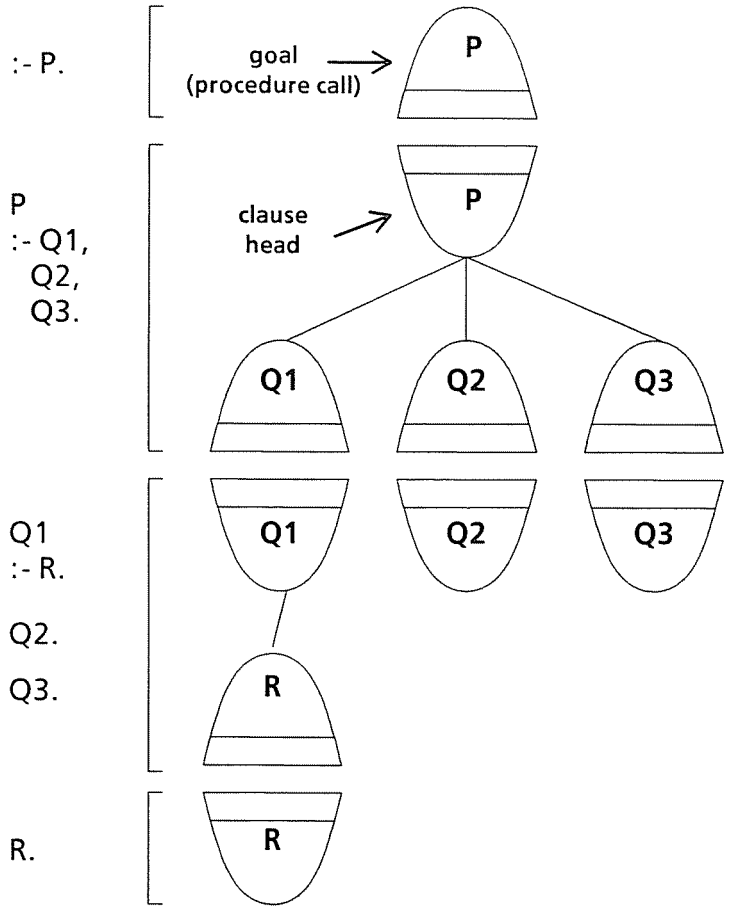



Figure 2. Proof Tree for the Prolog Procedure **P**.

Given the Prolog program on the left, successful execution of the procedure **P**, as invoked by the goal **:- P.** can be represented by the tree on the right. Each upper half circle represents a procedure call while the lower half circle represents a matching procedure. The Prolog machine must search through the program, matching the call against candidate procedures. The expense of the search and the associated pattern matching limit the Prolog performance. (The tree diagram has been called a Ferguson diagram [VAN84].)

Prolog procedures can also have parameters (Figure 3). Unlike parameters in conventional languages, Prolog parameters are neither strictly input nor output parameters. Rather, the role played by a parameter depends on the procedure call, and one of the very unusual things about Prolog parameters is that they can be both input and output. This aspect of parameters is a side effect of one of the more interesting of the ideas about computing that have been realized in the Prolog language. The idea is "call by description." Each parameter of a procedure is a description, as is each argument supplied by a procedure call. Descriptions can be more or less general depending on whether they contain variables or not.

On procedure invocation, the argument terms of the caller (the goal) are matched with the parameter terms of the called procedure. The pattern matching process (called unification) tests whether two terms can be matched by binding some of the variables in the terms. In a sense, unification is an attempt to find a view of the two descriptions under which they describe the same

thing. In Prolog, a successful unification of two terms results in the most general description covered by both original descriptions, which may be a specialization of the originals (Figure 4).

**Prolog procedures can have terms as parameters**

**A term may be:**
- **a constant**
- **a variable**
- **a structure**

**Constants are atomic objects**

**Variables stand for arbitrary objects**
**(by convention variable names begin with an uppercase letter)**

**Structures consist of a functor applied to terms as arguments**
**(eg. "p(a,b)" )**

**Some terminology:**

**p** **(a,b)**

**functor** **arity = 2**

Figure 3. Prolog at a Glance (II).

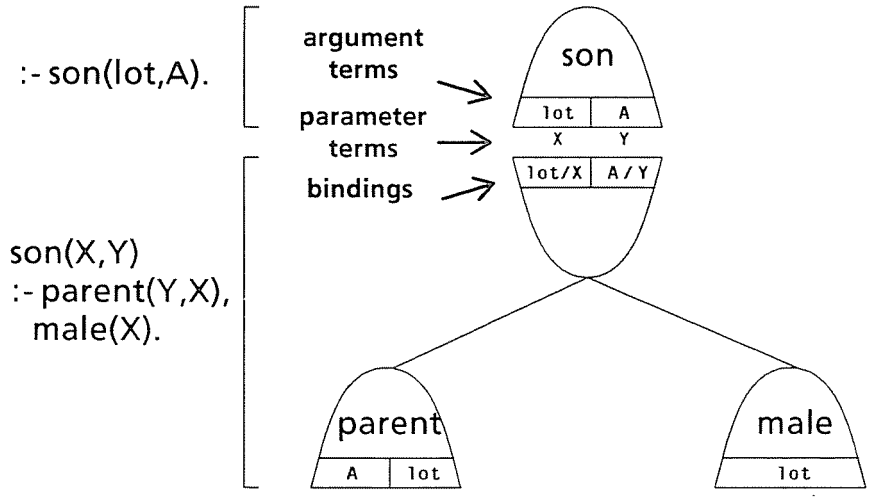



Figure 4. Procedure Invocation by the Goal **:- son(lot,A).**

Read the bindings **a/b** as "a is substituted for b." Following the first successful unification of the goal with the head of the procedure, the variable **X** in the procedure has been specialized to the constant **lot**. The variable **A** in the goal may be specialized by subsequent unification of the subgoals with other procedures.

The pattern matching procedure involved in unification can be expensive, primarily because so many cases need to be considered (Figure 5).

| | Constant Cp | Variable Xp | Structure Sp |
|---|---|---|---|
| Constant Ca | Succeed Ca = Cp | Succeed Xp = Ca | Fail |
| Variable Xa | Succeed Xa = Cp | Succeed Xa = Xp | Succeed Xa = Sp |
| Structure Sa | Fail | Succeed Xp = Sa | Succeed if * |

* Sa,Sp have same functor and arity
corresponding arguments of Sa,Sp unify

Figure 5. Cases Considered by the Unification Procedure.

Subscripts refer to the arguments passed by the caller (e.g., a structure **Sa**) and the parameters of the procedure (e.g., a structure **Sp**). Variables are de-referenced prior to comparisons, meaning if a variable has been bound, it is replaced by the bound value prior to comparison. Unification may recurse on structures.

When the structure analysis done by unification is delayed until run time, as in an interpreter, performance suffers, and, as was pointed out previously, searching for candidate matching clauses on a procedure call can also be expensive. These two observations lead to the basic compilation strategy for Prolog:

## Strategy for Compiling Prolog

### 1) Specialize unification for each clause.

- unification involves an analysis of structure, so move as much of the analysis as possible from run-time to compile-time.

### 2) Reduce the set of candidate clauses.

- index clauses by their structure. Common indices are main functor, arity and type of first parameter.

The focus of this paper is primarily the implementation of the first strategy because it is relatively easy to see how to approach the implementation of the latter. For example, if all procedures with the same main functor were chained together and accessed through the pfa of the main functor word, there would be a substantial reduction in the search space.

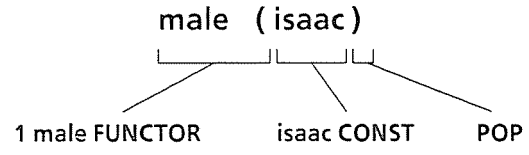
The approach taken here is to break down the process of compiler building into two steps. In the first step, a compiler is described that compiles Prolog to the instruction set of a Prolog Virtual Machine (PVM). The PVM used here has several advantages. First, it is easy to understand and implement because the set of instructions is small (there are only seven instructions). In addition, the compilation procedure is straightforward because there is essentially a one-to-one correspondence between clause structure and the object (PVM) code. Moreover, the PVM is a stack machine, which reduces the complexity of the compiler because issues like register allocation need not be considered. Finally, this PVM serves as a good introduction to the Warren Abstract Machine [WAR83] and the current literature on Prolog compilation.

With the first step being the construction of the compiler, the second step becomes implementing the PVM. This method is a common approach to compiler building, with speed being traded off against the advantages of portability and more compact code.

## *Prolog Compilation Step 1. Compile Prolog to PVM Instructions*

In explaining the compilation procedure, we delay considering full program compilation and look first at the compilation of Prolog structures. The object of the compilation is ultimately to compose a description of the Prolog structure using Forth words, which also happen to be instructions of the PVM. The descriptive words that are needed are the names of the types of Prolog terms—the unstructured terms such as variables and constants, and the structured terms like lists. The term types suggest using PVM instructions named `VAR` and `CONST` for unstructured terms and `FUNCTOR` for structured terms, with an instruction like `POP` used to indicate the termination of a structured term description. These instructions will eventually be implemented as Forth words.

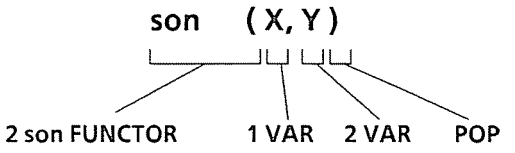
Given these four instructions, the procedure for compiling Prolog structures to PVM instructions is simply to compose a description of the structure using `CONST`, `VAR`, `FUNCTOR` and `POP`. Two examples of the compilation of Prolog structures follow. What is important to note is the near one-to-one correspondence between the Prolog objects that comprise the structure and the PVM instructions the structure compiles into.




Example 1. Compilation of the Structure **male(isaac)**.

The PVM code describes the term **male(isaac)** as a structure with functor = male and arity = 1, whose single formal parameter is a constant = isaac. The PVM instruction `POP` terminates the description. Parameters to PVM instructions are indicated in the familiar Forth reverse Polish syntax.

For a technical reason, the object code does not reference logical variables by their names in the source code. The reason is that variables occurring in a clause must be unique to each use of the clause. Thus, there can be no unique reference to the variable "X." With each procedure invocation, new procedure variables are created and associated with the procedure's stack frame. The compiler renames variables as they appear in a clause—first variable, second variable, etc.—to be used as indices into the area allocated for a procedure's variables. Thus, variables are referenced by number (index) in the object code.




Example 2. Compilation of the Structure **son(X,Y)**.

The PVM code describes the term **son(X,Y)** as a structure with functor = son, arity = 2. The two formal parameters of the structure are variables referenced by an index into an array of variables.

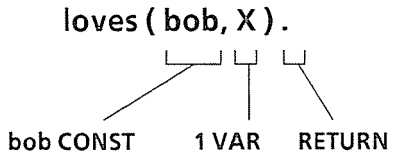
The procedure for compiling Prolog programs is quite similar to the procedure outlined for compiling structures; however, there are some additional steps and some subtleties. The chief additional step is marking transfer of control via the PVM instructions `CALL`, `ENTER`, and `RETURN`. The chief subtlety is the difference between the way the PVM instructions operate in the head and the body of a clause. In the head of a clause, the PVM instructions perform the operations of unification, as specialized for that clause. In the body of a clause, PVM instructions must prepare arguments for a procedure call. In other words, PVM instructions must operate in at least two modes: "match" mode in the head of a clause and "arg" mode in the body. This fact previews some implementation issues. The reason for mentioning it here is to explain the motivation behind the different forms of description used in the head and body of a clause.

A second subtlety is the effect of clause indexing on the compilation of the head of a clause. It is assumed here that clauses can be indexed by their main functor and arity: if the procedure **son/2** (i.e., functor = son, arity = 2) is being invoked, then candidate clauses can be found by looking, say, at the pfa of the word `son` and following a chain of pointers to the **son/2** clauses. The whole Prolog program does not need to be searched, and the functor and arity of the clause can be left off the description of the clause head.

With these additional facts in mind, we first consider the compilation of Prolog clauses without bodies—unit clauses.

### Compilation of Unit Clauses

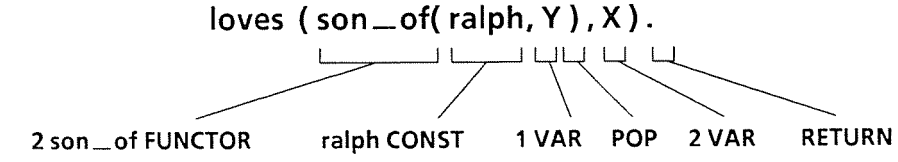
The chief differences between the compiled forms of structures and unit clauses are the indication of transfer of control in the Prolog program with the word `RETURN` and the fact that the functor and arity of the clause are not part of the object code emitted by the compiler. For example, extending the description of the compilation procedure, the clause **loves(bob,X).** is compiled as follows. First the compiler notes that its functor/arity is **loves/2** (this indicates where to store the compiled code) and that it has a single variable **X** and a single constant **bob**. Next a description of the clause is composed as before:




This is the program (description) for **loves(bob,X).** that is stored with the collection of clauses for functor/arity = **loves/2**.

A more complicated example is the clause **loves(son_of(ralph,Y),X).** There are two variables, one constant and a structure in this clause, and the PVM code emitted by the compiler is

```
2 son_of FUNCTOR ralph CONSTANT 1 VAR POP 2 VAR RETURN .
```




Lists may be represented by a structured term with functor/arity = **cons/2** (Figure 6). The first parameter of **cons/2** references the first element of the list, and the second parameter of **cons/2** references the rest of the list. Other representations of lists could be used to save both space and time at the (slight) cost of increasing the PVM instruction set.

| External Form | Internal Form |
|---|---|
| [] | nil |
| [a] | cons(a,nil) |
| [a\|[]] | cons(a,nil) |
| [a,b] | cons(a,cons(b,nil)) |
| [a\|[b]] | cons(a,cons(b,nil)) |
| [a\|b] | cons(a,b) |

Figure 6. Prolog List Syntax.

Prolog has several syntactic forms for lists. Generally, a list is enclosed by square brackets. The empty list [ ] is a constant, and the character | separates the beginning of a list from the rest of the list. There is a single internal representation of a list that, in the examples here, is a structure of functor = cons, arity = 2.

As a final example of the compiled form of a unit clause, consider **append([a,b],L,[a,b|L]).** This clause has only one variable. The PVM code emitted by the compiler is

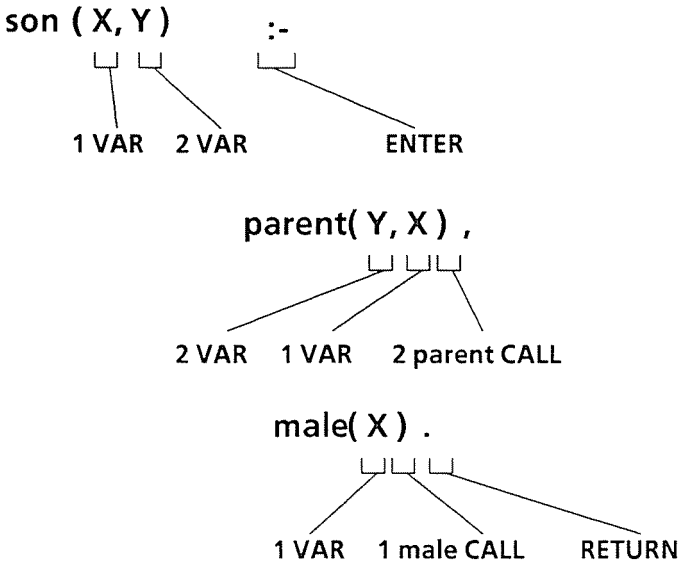
```
2 cons FUNCTOR a CONST 2 cons FUNCTOR b CONST nil CONST POP POP
   1 VAR
2 cons FUNCTOR a CONST 2 cons FUNCTOR b CONST 1 VAR POP POP
   RETURN
```

The compiled code can be read as a description of the structure of the clause **append([a,b],L,[a,b|L]).** With the PVM instruction set implemented in Forth, the description will constitute the program that is executed when **append/3** is called.

**Compilation of Non-Unit Clauses**

Compilation of non-unit clauses requires two additional PVM instructions: `ENTER` and `CALL`. The word `ENTER` is the object code representation of the "neck" (**:-**) of a clause. Its chief purpose is to switch the PVM execution mode and adjust certain pointers. The PVM instruction `CALL` takes a reference to the clause to be called as its argument. Its purpose is to transfer control to the called procedure and to save control information.

CALL is compiled following a description of the procedure arguments. As mentioned earlier, the compilation of a call is slightly different from the compilation of a structure. For example, consider the clause **son(X,Y) :- parent(Y,X),male(X).** The head of the clause is compiled as for unit clauses, the neck of the clause is marked in the object code by the instruction ENTER, and a procedure call is compiled after a description of the arguments to the procedure.




As a final example, consider the clause **append([X | L1],L2,[X | L3]) :- append(L1,L2,L3).** The clause has four variables. The PVM code for this clause is

```
2 cons FUNCTOR 1 VAR 2 VAR POP
  3 VAR
2 cons FUNCTOR 1 VAR 4 VAR POP
  ENTER
2 VAR 3 VAR 4 VAR 3 append CALL RETURN
```

## *Prolog Compilation Step 2: Implement the Prolog Machine*

Once the instructions of the Prolog machine have been named, and it is clear how to compile Prolog clauses to Prolog machine code, what remains is the implementation of the machine. This section describes the simulation of the Prolog machine in software.

There are three main components of the simulation. The first is the internal representation of Prolog terms (e.g., constants, variables, and structures) and of references to these objects. The second component concerns the structure of the stacks required to support Prolog computation. The final component is the procedural semantics of the PVM instructions—what the instructions do. Side issues like the memory map, implementation registers and scratch stacks will be touched on but in less depth.

### Internal Representation of Terms and References to Terms

First consider references to Prolog terms. As previously mentioned, there are three primitive types of Prolog terms. One internal representation could be a 32-bit cell with the 2 high-order bits indicating the type of the term and the remaining bits containing a pointer to the term (this is just

a generalization of the idea of pointer). The two fields of the reference are called the "tag" and the "val," following Clocksin [CLO85]. The following represents a sufficient set of references to primitive objects:

| Tag | Val | Purpose |
|---|---|---|
| 1 | pointer to a variable binding | variable |
| 2 | pointer to a constant record | constant |
| 3 | pointer to a structure record | structured term |

For the sake of efficiency, it may be desirable to increase the number of types of terms that can be referenced. For example, it can be worthwhile to have a special type of reference for integers even though integers could very well be referenced like any atom. Similarly, one might wish to reference lists as distinct from general structures and unbound variables as distinct from bound variables.

The simplest of the internal representations of terms is the representation of variables. Variables and references to variables are identical. The val field of a variable reference points to a reference to a Prolog term. An unbound variable is often indicated by a reference structure that has the tag field of a variable and a val field that points to itself.

The internal representations of Prolog constants and structures are distinct from the representations of references to these types of terms. Both constants and structures are represented by different kinds of records, with distinct fields holding relevant information about the term. For example, the record representing a structure holds the information about its functor and arity, as well as references to its parameter terms.

Constant and structure representations are built in different areas of memory. Structures reside exclusively in an area of memory called the structure stack. This stack constitutes the necessary dynamic memory allocation required for Prolog computation and simplifies the garbage collection problem because stacks grow and shrink with the computation.

Constants also reside in a special area of memory. For a simple Forth implementation of Prolog, we can identify the Forth dictionary with the constant space (presumably Prolog would exist in a separate Forth vocabulary). Each Prolog constant is then represented by a Forth word, the header storing the name string and the parameter field storing other information. This particular implementation scheme leaves the garbage collection of constants unresolved, which may be a problem.

The Forth dictionary can also be the place where Prolog programs are stored. Indexing of clauses in a particular procedure can be done through the main functor of the procedure. One simple way to do this is to chain procedures by arity, with the pointer to this chain stored in the pfa of the main functor word. Clause records can then be chained off the procedure records. With this structure, procedure invocation begins with a search down the procedure links, and backtracking resumes a search down the clause links.

Garbage collection of procedures may be necessary if there is significant data base manipulation in a Prolog program. This could be accommodated here by allocating space for procedures from a heap [DRE85].

In summary, there are four kinds of record structure:

*Constant record* (2 fields). The first field is the name string of the constant, and the second field is a pointer to a chain of procedure records. For our purposes, a constant record is a Forth word, the header containing the name string, link field, etc., and the first cell of the parameter field pointing down the procedure chain (i.e., the Forth code used to build and initialize constant records is `CREATE 0 ,`).

*Structure record* (3 fields). The first field is a pointer to the constant naming the functor, the second field holds the number of arguments of the structure, and the third is a variable length field containing the references to the formal arguments of the structure. Prolog structure records are built by `FUNCTOR` descriptions in space allocated from the structure stack.

*Procedure record* (3 fields). The first field is a pointer to the next procedure in the chain having the same functor but different arity, the second field holds the arity of the procedure, and the third field contains a pointer to a chain of clause records. Space for procedure records can be allocated from the Forth dictionary or from a heap.

*Clause record* (3 fields). The first field is a pointer to the next clause record, the second field holds a number indicating how many variables are in the clause, and the third is a variable length field that contains the code itself (effectively the Forth parameter field). Space for clause records can be allocated from the Forth dictionary or from a heap.

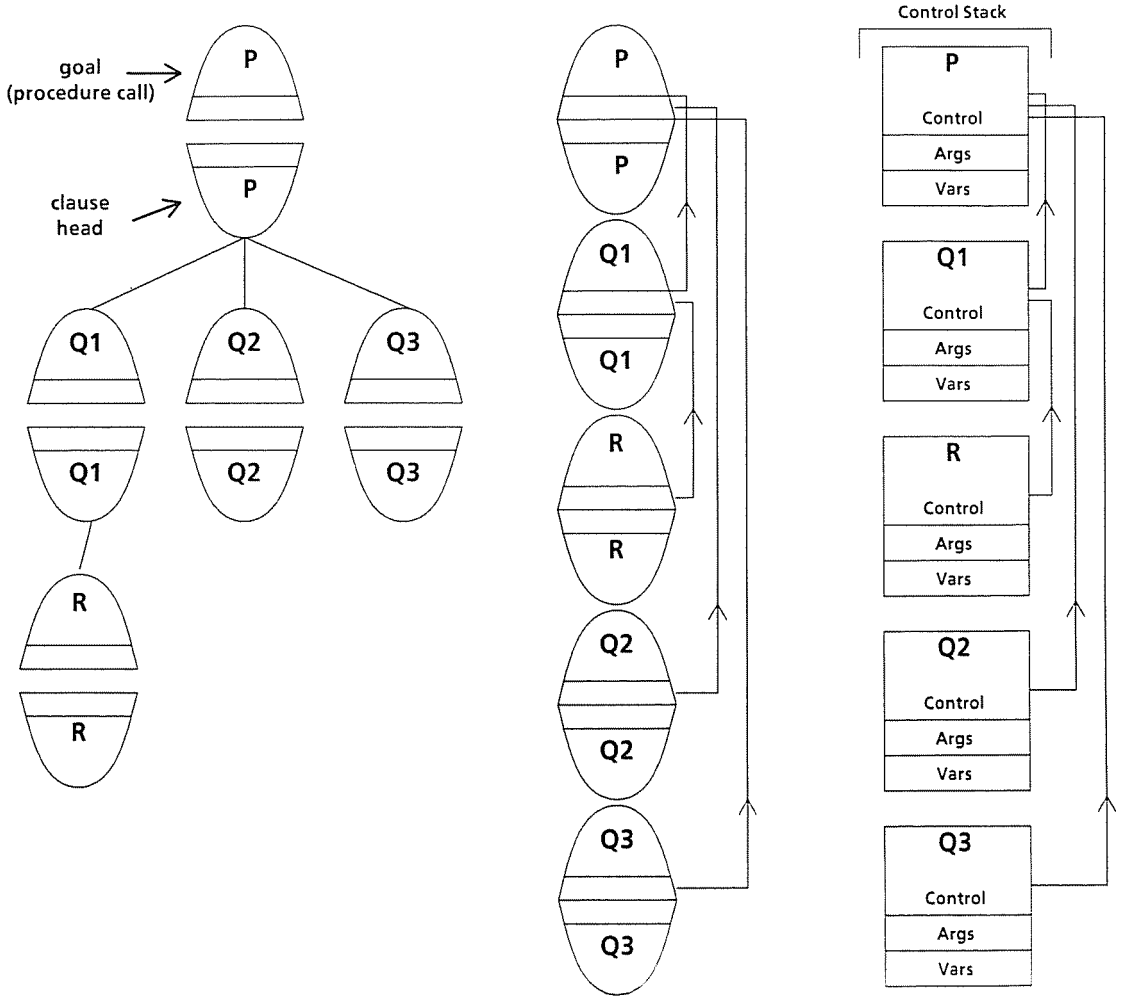



Figure 7. Structure of the Control Stack.

The control stack (right-hand side) constitutes a trace of the procedure calls during a Prolog computation and therefore is a representation of the Prolog proof tree (left-hand side), with each stack frame corresponding to a procedure call. The stack frame holds control information, procedure arguments and the clause variables. In practice, it is possible to reclaim space on the control stack during a computation.

### Stack Structure

There are two main stacks in a Prolog machine. The first is the structure stack that holds any temporary structures created during the computation. This is a straightforward stack requiring only a pointer to its top. The second of the Prolog stacks (control stack) holds state information, the arguments passed to procedures and procedure variables. This stack is essentially a linear version of the proof tree traced out during a Prolog computation (Figure 7).

The Prolog stacks must generally be large relative to the usual Forth stacks because, in the case of the structure stack, the stack is the mechanism for dynamic memory allocation, and, in the case of the control stack, nondeterminism requires that all state information be saved in case backtracking is necessary. Thus, a procedure return does not necessarily pop the control stack, and the stack can grow quite deep.

### Implementation of PVM Instructions

The following section describes in detail the workings of software simulations of the PVM instructions. The first issue is execution modes. As alluded to previously, the PVM instructions `CONST`, `VAR` and `FUNCTOR` operate in modes, the two main modes being "match" and "arg." There is a third mode called "copy" that is a variant of "arg" mode.

The instructions operate in match mode in the head of a clause, matching the parameters of the clause with the arguments passed to the procedure on the control stack. Instructions operate in arg mode in the body of a clause, placing arguments on the control stack prior to a procedure call. Modes are switched by the instructions `CALL`, `ENTER` and `RETURN` (Figure 8).

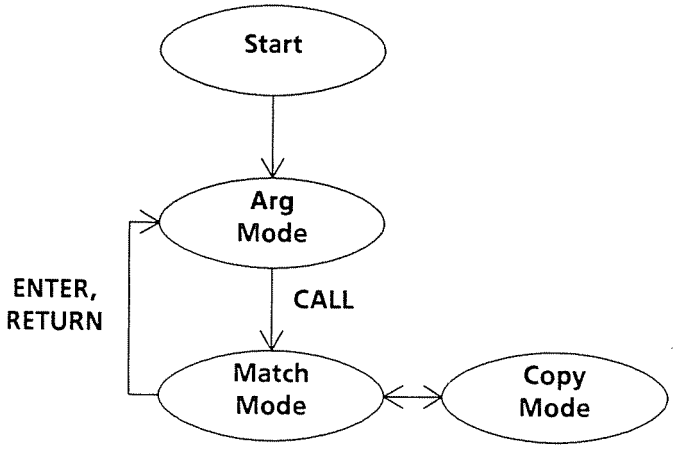



Figure 8. Mode Switching in the Prolog Machine.

A procedure invoked from top level would begin executing in arg mode, placing arguments on the control stack prior to a call. The call (PVM instruction `CALL`) switches the mode to match, and the arguments are matched with the parameters in the head of the clause. If the match is successful, the body of the clause is entered (PVM instruction `ENTER`), the mode is switched to arg, and arguments are placed on the control stack prior to the first call in the body. The mode is also switched to arg on a procedure return (PVM instruction `RETURN`). This is only strictly necessary when returning from unit clauses.

### `CONST`, `VAR` and `FUNCTOR` in Arg Mode

In the discussion to follow, operation of PVM instructions in each mode will be considered according to the mode sequence pictured in Figure 8—first arg mode, then match mode, and finally copy mode. To begin, we look again at how a procedure call (goal) compiles, focusing now on what the code does. For example, the goal **parent(haran,X)** compiles to the following PVM instructions.

```
haran          CONST        % push reference to haran
n              VAR          % push reference to variable
2 parent       CALL         % call parent/2
```

If this procedure call is successful, the variable **X** is bound to the child of **haran**.

At this stage of execution, the PVM is in arg mode, and the effect of PVM instructions is to place references to arguments on the control stack. An argument pointer is maintained to indicate where the arguments are to go. The actions of PVM instructions in arg mode are described as follows.

| Instruction | Parameter(s) | Description (arg mode) |
|---|---|---|
| CONST | C: pointer to a atom | push reference to C on control stack<br>advance arg pointer, continue |
| VAR | I: index into environment | dereference Ith variable<br>push result on control stack<br>advance arg pointer, continue |
| FUNCTOR | F: pointer to an atom<br>N: integer | build F/N on structure stack<br>push reference to it on control stack,<br>push copy of arg pointer,<br>reset arg pointer to 1st parameter of F/N,<br>continue |
| POP | none | pop arg pointer, continue |

Thus, in executing the goal **parent(haran,X)**, the control stack has two argument references on it, the argument pointer indicating the first of these, just before the procedure call is made (Figure 9).

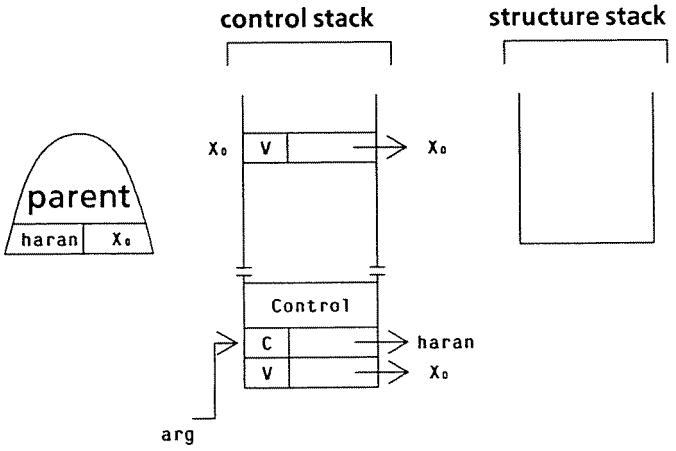



Figure 9. Stacks before CALL.

References to the arguments have been loaded on the control stack, and an argument pointer is set to the first argument reference. The C in the tag field of the first argument indicates that it references a constant; its val field points to the constant. The V in the tag field of the second argument indicates that it references a variable; its val field points up earlier in the control stack to the original variable reference. The fact that the val field of this earlier variable reference points to itself denotes that the variable is unbound.

The PVM instruction sequence `2 parent CALL` results in a search through the procedure records from **parent**, looking for procedures whose arity is 2. If one is found, the execution mode is switched to match, control is transferred to the procedure code, and the pattern matching process begins. Transfer of control instructions are detailed later.

A more complicated example, one that involves the structure stack, is the code for the goal **derivative(sin(a),a,Y)**. If this procedure is successful, it results in the binding of the variable **Y** to the derivative of **sin(a)** with respect to **a**.

```
1 sin          FUNCTOR  % create sin/1, push reference, reset arg
                          pointer
a              CONST    % push reference to a
               POP      % restore arg pointer to derivative/3 from
                          sin/1
a              CONST    % push reference to a
n              VAR      % de-reference var, push reference
3 derivative CALL
```

Thus, in executing this goal, the control stack has three argument references on it just before the procedure call is made, and the argument pointer indicates the first of these (Figure 10). The first argument references a structure on the structure stack.

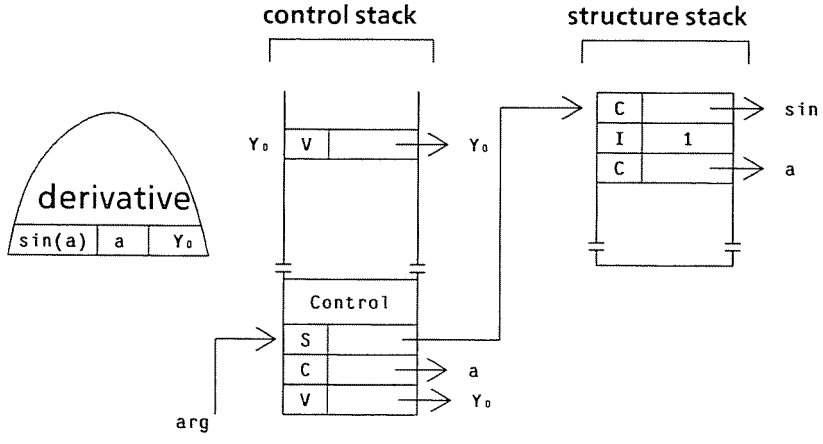



Figure 10. Stacks before `CALL`.

References to the arguments have been loaded on the control stack, and an argument pointer is set to the first argument reference. The `S` in the tag field of the first argument indicates that it references a structure; its val field points to the structure, which is located in the structure stack. The structure was built by the sequence of instructions `1 sin FUNCTOR a CONST POP` operating in arg mode.

As before, the PVM instruction sequence `3 derivative CALL` results in a search through the procedure records from the word `derivative`, looking for any procedures whose arity is 3. If one is found, the execution mode is switched to match, control is transferred to the procedure code, and the pattern matching process begins.

### `CONST`, `VAR` and `FUNCTOR` in Match Mode

The `CALL` instruction switches the execution mode of the PVM to match before transferring control. In this mode, the PVM instructions of the compiled form effect the matching between the argument and the parameters of the clause. For any instruction, if the argument is an unbound

variable, the instructions immediately bind that variable to the appropriate term. For arguments that are other than unbound variables, the actions of PVM instructions in match mode are described as follows.

| Instruction | Parameter(s) | Description (match mode) |
|---|---|---|
| CONST | C: pointer to a atom | if arg not a constant, fail<br>else if arg not = C, fail<br>else<br>advance arg pointer, continue |
| VAR | I: index into environment | dereference Ith variable<br>if result is an unbound variable,<br>bind to arg, advance arg pnter, continue<br>else if result type not = arg type, fail<br>else unify result and argument<br>if unificiation not successful, fail<br>else advance arg pointer, continue |
| FUNCTOR | F: pointer to an atom<br>N: integer | if arg not a structure, fail<br>else if functor of arg not = F<br>or arity of arg not = N, fail<br>else push copy of arg pointer,<br>reset arg pointer to 1st parameter of arg,<br>continue |
| POP | none | pop arg pointer, continue |

As an example of match mode operation, consider the compiled forms of unit clauses **parent(haran,lot).** and **parent(abraham,isaac).**

```
haran     CONST     % match 1st arg with haran
lot       CONST     % match 2nd arg with lot
          RETURN

abraham   CONST     % match 1st arg with abraham
isaac     CONST     % match 2nd arg with isaac
          RETURN
```

The operation of this code in realizing the unification is straightforward (Figure 11). If any of the matches fail, backtracking is invoked.

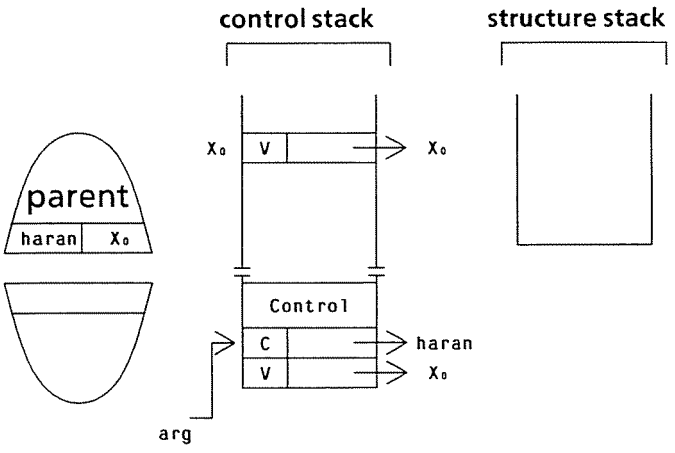



Figure 11a. After `CALL`, before `haran CONST`.

Before the beginning of execution of the PVM code for the procedure, the control stack contains the arguments, and an argument pointer indicates the first of them. The PVM code `haran CONST` will check whether the first argument references the constant **haran**. In the case illustrated, the first argument does reference **haran** so the match succeeds, the argument pointer is advanced, and execution continues.

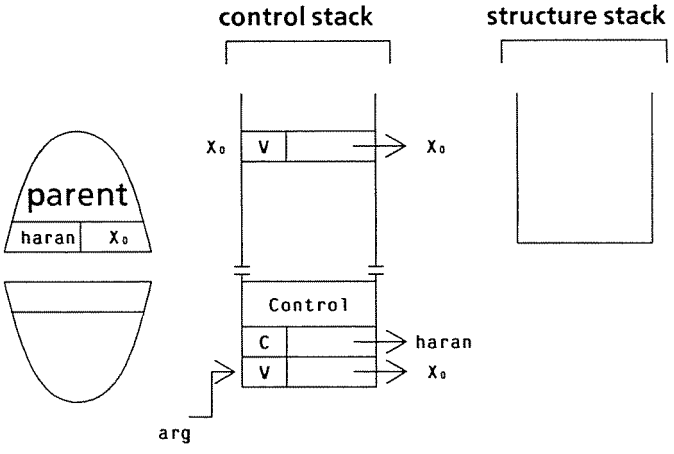



Figure 11b. After `haran CONST`, before `lot CONST`.

The argument pointer now points to the second argument, which references an unbound variable. The PVM code `lot CONST` notes that the argument is an unbound variable and therefore binds it by making it a reference to the constant **lot**.

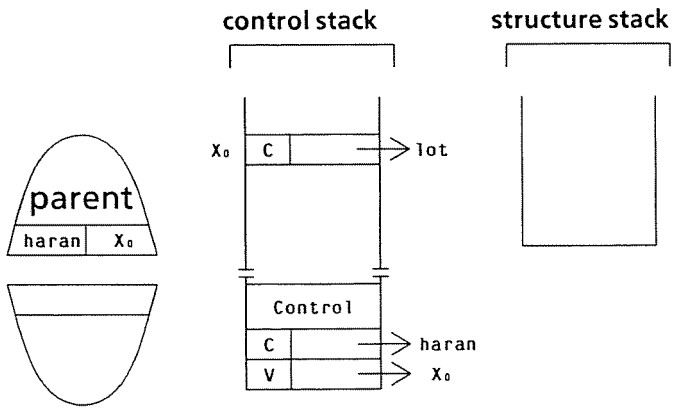



Figure 11c. After `lot CONST`, before `RETURN`.

Note that the variable referenced by the second argument has been replaced by a reference to the constant **lot**. At this point the stack frame for the procedure could be reclaimed if no more alternatives remained. Otherwise argument and control information must be maintained for this procedure in the event that the computation backtracks to this point.

Two additional illustrations of unit-clause PVM code follow. The first example is the code for the clause **derivative(sin(X),X,cos(X)).**, which states that the derivative of the sine of any argument with respect to that argument is the cosine of that argument. This clause compiles to:

```
1 sin  FUNCTOR  % match 1st arg with sin/1, reset arg pointer
1      VAR      % match 1st parameter of sin/1 with first var
       POP      % restore arg pointer to derivative/3 from sin/1
1      VAR      % match 2nd arg with first var
1 cos  FUNCTOR  % match 3rd arg with cos/1, reset arg pointer
1      VAR      % match 1st parameter of cos/1 with first var
       POP      % restore arg pointer to derivative/3 from cos/1
       RETURN
```

The second example, which contains nested structures, is the clause

**derivative(**(sin(X),2),X,*(2,*(sin(X),cos(X))))).**

This clause states that the derivative of the square of the sine of some argument is twice the product of the sine and the cosine. Using infix notation the clause would read

**derivative(sin(X)**2,X,2*sin(X)cos(X)).**

The clause compiles to:

```
2 **   FUNCTOR  % match 1st arg with **/2, reset arg pointer
1 sin  FUNCTOR  % match 1st parameter of **/2 with sin/1, reset arg pointer
1      VAR      % match 1st parameter of sin/1 with first var
       POP      % restore arg pointer to **/2 from sin/1
2      CONST    % match 2nd parameter of **/2 with "2"
       POP      % restore arg pointer to derivative/3 from **/2
1      VAR      % match 2nd arg with first var
2 *    FUNCTOR  % match 3rd arg with */2, reset arg pointer
```

```
2       CONST    % match 1st parameter of */2 with "2"
2 *     FUNCTOR  % match 2nd parameter of */2 with */2, reset arg pointer
1 sin   FUNCTOR  % match 1st parameter of */2 with sin/1, reset arg pointer
1       VAR      % match 1st parameter of sin/1 with first var
        POP      % restore arg pointer to **/2 from sin/1
1 cos   FUNCTOR  % match 2nd parameter of */2 with cos/1, reset arg pointer
1       VAR      % match 1st parameter of cos/1 with first var
        POP      % restore arg pointer to **/2 from cos/1
        POP      % restore arg pointer to **/2 from **/2
        POP      % restore arg pointer to derivative/3 from **/2
        RETURN
```

### CONST, VAR and FUNCTOR in Copy Mode

The remaining complication that must be dealt with in respect to the operation of the PVM instructions CONST, VAR and FUNCTOR is operation in copy mode. Copy mode is entered when an argument is an unbound variable and the corresponding parameter is a structure. In this case, the structure must be built and placed on the structure stack, and the variable reference must be replaced by a reference to the structure. The process of building the structure is similar to what takes place in the body of a clause except that, in this case, the structure building code is in the clause head—thus the need for a different mode. The operation of the PVM instructions in this mode is described in the following table.

| Instruction | Parameter(s) | Description (copy mode) |
|---|---|---|
| CONST | C: pointer to a atom | copy C reference to structure stack,<br>advance arg pointer, continue |
| VAR | I: index into environment | dereference Ith variable<br>if result is an unbound variable,<br>create new unbound var on struct. stack<br>bind referenced var to new var<br>else<br>copy reference to structure stack<br>then advance arg pointer, continue |
| FUNCTOR | F: pointer to an atom<br>N: integer | build F/N on structure stack<br>push copy of arg pointer,<br>reset arg pointer to 1st parameter of struct,<br>continue |
| POP | none | pop arg pointer, continue |

As an example of copy mode operation, consider the clause

**derivative(sin(X),X,cos(X)).**

as called by

**derivative(sin(a),a,Y).**

Pictures of the stacks through execution are given in Figure 12.

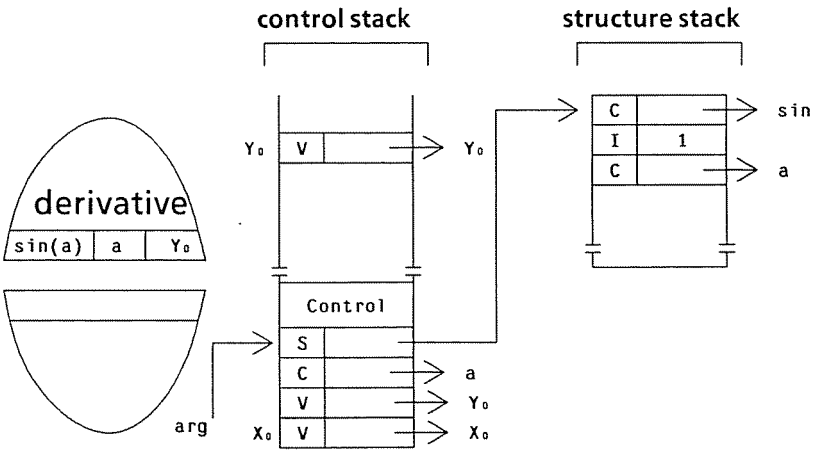



Figure 12a. After `CALL`, before `1 sin FUNCTOR`.

Before the beginning of execution of the PVM code for this procedure, the control stack contains the arguments, and an argument pointer indicates the first of them. Because the clause contains one variable, space has been allocated on the control stack following the procedure arguments, and the variable has been initialized as unbound. The PVM code `1 sin FUNCTOR` will check whether the first argument references a structure with functor = sin and arity = 1. In the case illustrated, the first argument does reference such a structure so the match succeeds, a copy of the argument pointer is saved, and the argument pointer is set to point to the first parameter of the structure.

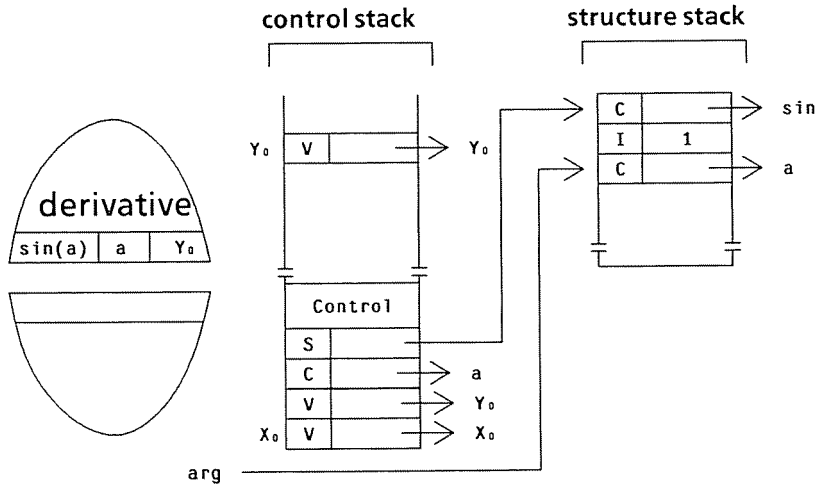



Figure 12b. After `1 sin FUNCTOR`, before `1 VAR`.

The PVM code `1 VAR` will de-reference the procedure's first variable and compare the result with the reference pointed to by the argument pointer. At this stage of the computation, the variable is unbound, and the argument reference is to the constant `a`. The variable is then bound to `a` (its reference is changed to `a`). The argument pointer is advanced.

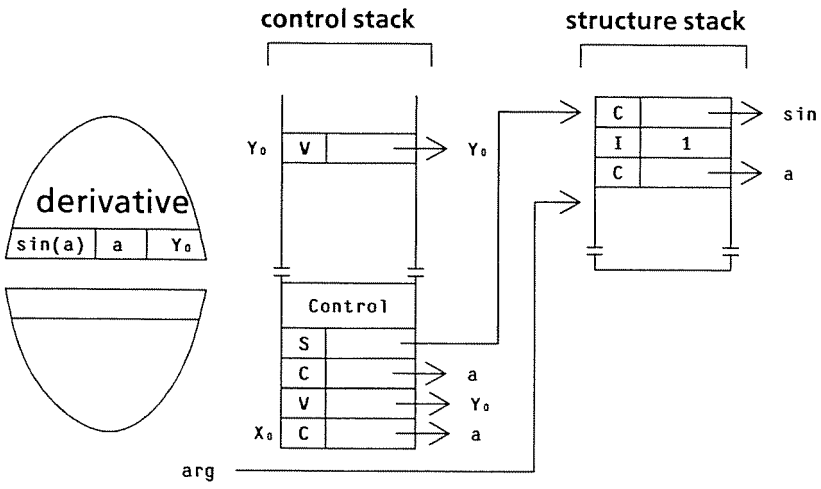



Figure 12c. After 1 VAR, before POP.

The PVM code POP will restore the argument pointer to the value it had before FUNCTOR was executed. Note that the cell allocated for the first variable of the procedure now references the constant a.

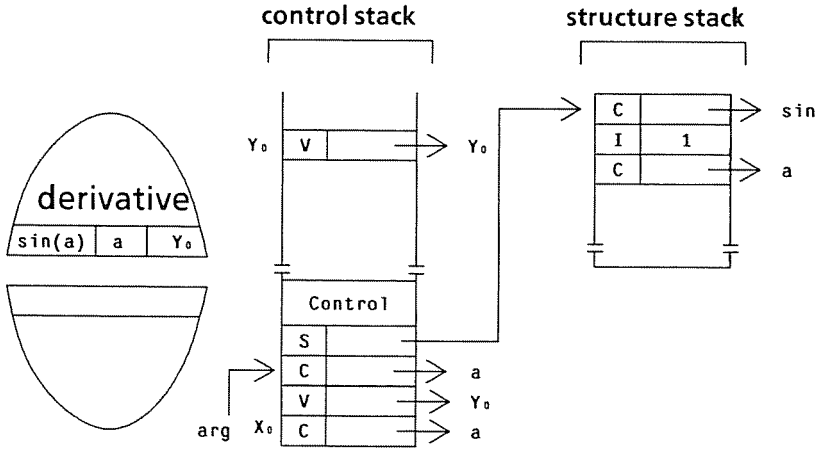



Figure 12d. After POP, before 1 VAR.

The PVM code 1 VAR will consult the term referenced at the memory location of the first procedure variable and compare the result with the reference pointed to by the argument pointer. At this stage of the computation, the variable is bound to the constant a and the argument reference is to the constant a; therefore, the variable and the argument will match. The argument pointer is advanced.

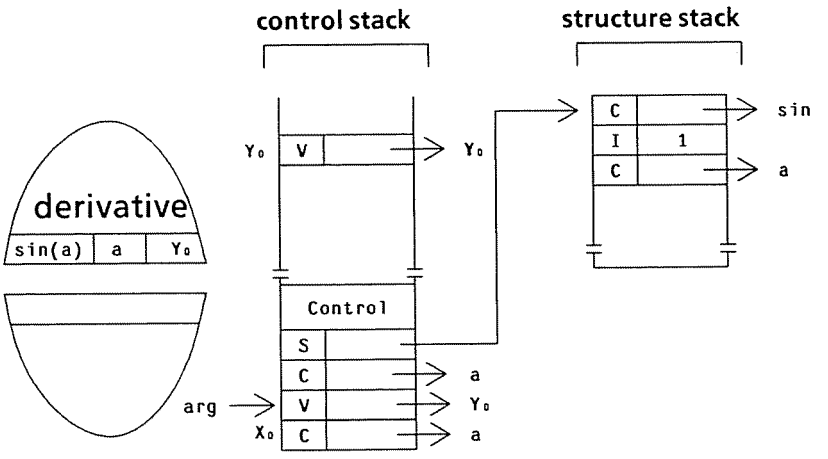



Figure 12e. After 1 VAR, before 1 cos FUNCTOR.

The PVM FUNCTOR instruction will notice that the next argument references an unbound variable. The mode will be switched to copy, and a structure will be constructed on the structure stack. The structure is known to have functor = cos and arity = 1; therefore, space for the structure can be allocated, and the corresponding structure reference can replace the unbound variable reference.

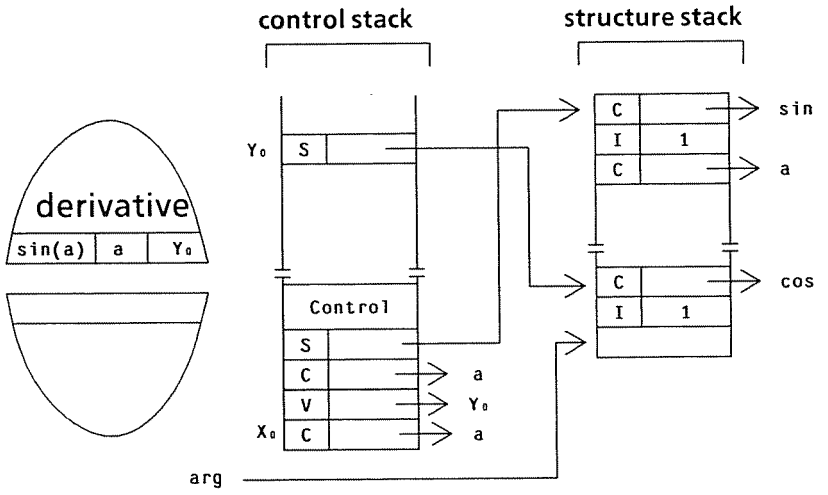



Figure 12f. After 1 cos FUNCTOR, before 1 VAR.

Once space for the structure has been allocated and the variable bound, a copy of the argument pointer is saved, and the pointer is reset to the first parameter position of the new structure. Following PVM code will cause references to Prolog terms to be placed at the positions indicated by the argument pointer.

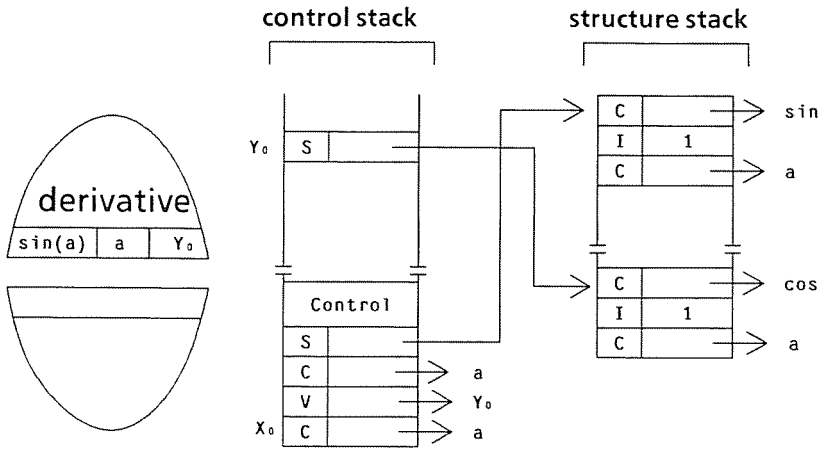



Figure 12g. After 1 VAR POP.

The first variable is again de-referenced and copied to the position indicated by the argument pointer. Execution of POP will restore the argument pointer to its value before execution of FUNCTOR and change the execution mode back to match.

In closing this section, some final comments on references to Prolog terms and the binding of Prolog variables are in order.

- The only Prolog terms that live in the control stack are variables.
- Structures live only in the structure stack. No subterm of a structure exists in the control stack.
- Variables in the control stack can be bound only to constants, terms in the structure stack, or variables occurring earlier in the control stack.

Maintaining this discipline facilitates the restoration of the state of the Prolog computation in case backtracking is required. The fact that structures live completely and only in the structure stack means that they can be readily disposed of on backtracking simply by changing the pointer to the top of the structure stack. Similarly, variable-variable binding is required to be from the most recent variable to the least recent variable, both in the control and the structure stacks. This binding discipline simplifies backtracking and means as well that the control stack frame for deterministic procedures may be reclaimed without creating dangling pointers.

There must also be a mechanism that will note the binding of variables that have been created before the most recent backtrack point because, on backtracking, the bindings of these variables must be undone. The mechanism is a special stack called the trail. On binding a variable that lives earlier than the most recent backtrack point, a pointer to the variable is pushed on the trail. The trail stack pointer is part of the control information saved with a control frame, thereby providing the necessary information to reset variables on backtracking.

### Transfer of Control Instructions CALL, ENTER and RETURN

The instructions of the PVM that remain to be described are the flow of control instructions CALL, ENTER and RETURN. Most of what these instructions do has been described previously and is summarized in the following table.

| Instruction | Parameter(s) | Description |
|---|---|---|
| CALL | F: pointer to an atom<br>N: integer | find first clause with functor F arity N,<br>if found, allocate space for variables,<br>copy control information to control stack,<br>if current clause has remaining alternatives,<br>update backtrack pointer,<br>copy backtrack info to control stack<br>set execution mode to "match"<br>transfer control to clause<br>else fail |
| ENTER | none | set execution mode to "arg",<br>adjust stack frame pointers |
| RETURN | none | if deterministic, reclaim control stack frame<br>set execution mode to "arg"<br>transfer control back to caller |

As an example of the compilation of a full clause, consider

**son(X,Y) :- parent(Y,X),male(X).**

which compiles to:

```
1          VAR     % match 1st arg with first var
2          VAR     % match 2nd arg with second var
           ENTER   % set execution mode to arg
2          VAR     % de-reference then copy 2nd var to control stack
1          VAR     % de-reference then copy 1st var to control stack
2 parent   CALL    % transfer control to parent/2 or backtrack
1          VAR     % de-reference then copy 1st var to control stack
1 male     CALL    % transfer control to male/1 or backtrack
           RETURN  % reclaim stack area, return control to caller
```

Several Prolog machine implementation registers are needed to support the computation (Figure 13). These registers contain pointers into the code, pointers to the control, structure and trail stacks, a flag indicating the execution mode, and the argument pointer. Some of the registers are saved by the instructions `CALL` and `ENTER` and then restored by `RETURN` and the backtracking mechanism. `CALL` and `RETURN` always save and restore the program counter and a pointer to the control stack frame of the current procedure. These are the first two registers in the table that follows. If the procedure is deterministic, these are the only two registers saved. If a procedure is non-deterministic—there are remaining alternatives (as indicated by the link on the code record)—`CALL` saves the contents of all six registers in the table. These six constitute sufficient information to restore the execution state on backtracking (Figure 14).

| Register | Description |
|---|---|
| RC | pointer to code;<br>the return point in the calling procedure |
| RF | pointer to the control stack;<br>stack frame of the calling procedure |
| BC | pointer to a procedure;<br>next procedure on backtracking |
| BF | pointer to the control stack;<br>last choice point |
| SS | pointer to structure stack;<br>reset to this value on backtracking |
| TS | pointer to trail stack;<br>reset variables on here on backtracking |

Figure 13. Prolog State Registers.

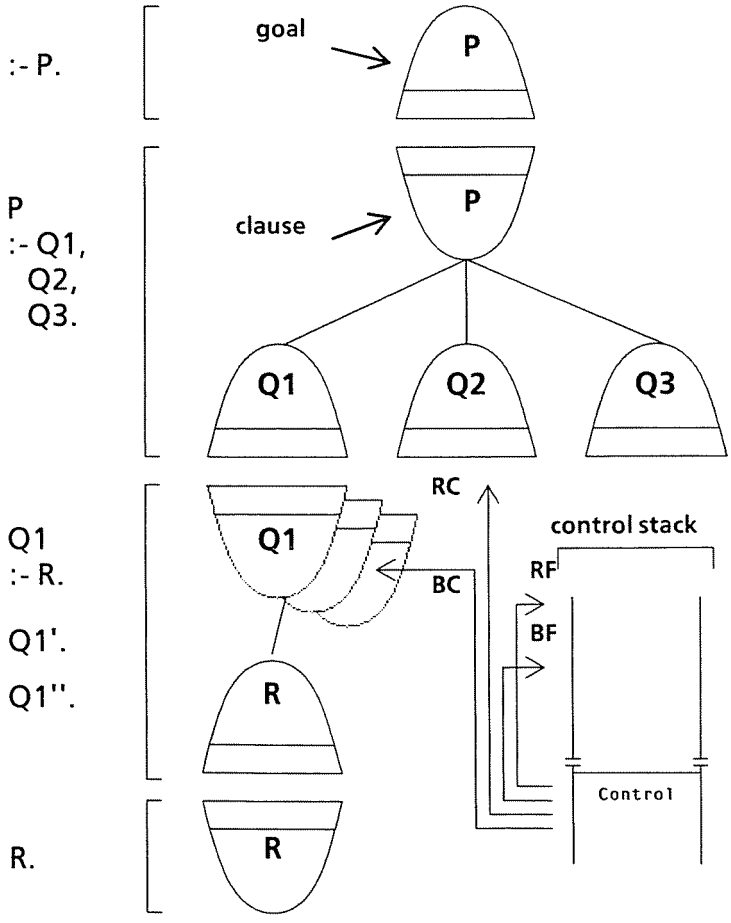



Figure 14. Control Information Saved by `CALL`.

This figure indicates the main registers saved by `CALL` in the case of a call to a procedure that has more than one clause. Also saved are the trail and structure stack pointers.

**Backtracking**

Backtracking occurs on failure to find a procedure with the correct functor and arity (`CALL`), on failure to match the arguments of a call and the parameters of a procedure (`CONST`, `VAR`, `FUNCTOR`), or on explicit invocation via the predicate **fail**. The following events are triggered by backtracking:

- Go back to most recent choice point (set current frame to contents of `BF` register).
- If there is only one remaining alternative clause, update the choice point (restore `BF` from (new) current frame if necessary).
- Garbage collect the structure stack (restore `SS` from (new) current frame).
- Re-initialize variables where necessary (restore `TS` from (new) current frame; unbind any trailed variables).
- Transfer control to the next alternative clause (reset program counter from `BC` in (new) current frame).

## *PVM Implementation in Forth*

A Forth word set that implements the PVM instructions described here is given in Appendix A. This code supplies most of the functionality required by the Prolog machine. The word set described in the appendix has been built in MicroMotion MasterForth and runs on the Apple Macintosh. Using colon words exclusively, the compiled Prolog runs over ten times faster than an earlier Prolog interpreter in Forth [ODE87].

An optimized version of the PVM (see optimization below) has been ported to the NC4000P Forth engine where it runs the naive reverse benchmark at 6K LIPS with a clock rate of 4 MHz. (The Logical Inference Per Second measures, in effect, the procedure call rate.) At a clock rate of 10 MHz., it is estimated that this version would achieve performance equivalent to the fastest compiled Prolog (Quintus Prolog) running on the VAX 11/780 [ODE86].

There are some minor differences between the PVM described here and the version simulated by the Forth code in the appendix. The most significant difference is the interpretation of the argument of the PVM instruction `VAR`. The Forth code for `VAR` uses a byte offset from the start of the stack frame to locate a variable. By contrast, the PVM `VAR` instruction described above uses an index into a table of variables. The latter convention makes the description of the PVM less complicated; the former makes the PVM execution somewhat faster.

With some extra work, the PVM to Forth compiler (the word `ASSERTZ` in screen 40) could calculate the byte offset from the table index. For example, in a clause with two parameters, the first variable is allocated on the control stack after the control information (12 bytes) and the arguments (2 arguments times 4 bytes/argument), so its offset from the start of the stack frame would be 12 + 8 = 20 bytes.

A second difference between the general PVM and the Forth simulation lies in the way references to Prolog objects are tagged. Since a small model Forth is assumed in the simulation, all pointers are 16 bits, and therefore the high order 16 bits of the object reference are free to be used for the tag. This makes both tagging and testing tags very simple. A version for a large model Forth would require more complicated code for these operations.

## *The Compiler*

**Basics**

The Prolog compiler whose (Prolog) code appears in Appendix B accepts a restricted Prolog syntax (Figure 15). The most important restriction of the syntax is that all predicates be expressed

in functional form. Extension of the compiler to accept other operator positions does require a significant effort, although a straightforward path to the more general syntax would be to build a preprocessor that transforms all predicates into functional form. The output of this program could then be used as the compiler input, and the compiler per se would not have to be modified. There are other parts of the usual grammar that are not recognized by the grammar used here (e.g., strings), but adding them requires only simple modifications.

```
<horn_clause> ::= <atmf>. | <atmf> :- <atmfs>.

<atmfs> ::= <atmf> {,<atmf>}

<atmf> ::= <atom_name> | <atom_name> ( <args> )

<args> ::= <simple_term> | <simple_term> , <args>

<simple_term> ::= <atom_name> ( <args> ) |
                  <variable> |
                  <constant> |
                  <list> |
                  (<simple_term>) |
                  (<conjunction>)

<conjunction> ::= <simple_term> , <simple_term>
                  <simple_term> , <conjunction>

<atom_name> ::= <lower case identifier>

<variable> ::= <identifier starting with uppercase or "_">

<constant> ::= <atom_name> | <integer>

<list> ::= [ <simple_term> ] |
           [ <simple_term> {, <simple_term> } | <list> ]
```

Figure 15. Grammar Accepted by the Compiler.

The input to this compiler is a list of tokens for a single clause, terminated by the token . . The tokens are the names of each constant, variable and functor, along with parentheses, quotes, punctuation and the clause neck (:- ). The tokenizer is not described here, but it is relatively easy to construct (see [CLO81], p. 86 ). Note that the grammar accepted by the compiler recognizes structured terms with spaces between the functor name and the left parenthesis bracketing the functor arguments. One approach to improving on this is to annotate the identifiers produced by the tokenizer, thereby indicating to the compiler that an atom followed immediately by a ( is a functor name.

The compiler is implemented as a grammar, using the grammar rule facility provided in most Prologs [CLO81]. The grammar consists of a collection of rules that define the strings of symbols that are valid sentences of the language. Grammar rules may also provide for some analysis of the sentence, often transforming it into a structure which is meant to clarify its meaning. The grammar presented here analyzes the input string in this manner, transforming it into code for the Prolog machine.

**Optimizations**

The compiler and PVM have been simplified as much as possible for the purpose of exposition; however, there are a number of modifications that will increase execution efficiency (at the expense of increasing the complexity of the compiler and adding words to the Forth vocabulary). For example, the code density could be reduced by putting all object references for each clause into a table. Then, instead of each type word taking an object reference as its argument, it could just take an index into the reference table. Type words could then be specialized by index, e.g., `1CONSTANT`, `2CONSTANT`, etc. The result would be that, in the code, only one cell is required for most primitive object descriptions instead of two. The cost is the time required to extract the references from the table.

The PVM instructions may be specialized in other ways. For example, the constant **nil** could be described by a special word, such as `CONSTNIL`, thereby saving both time and space in the reference table. A special functor description word for **cons/2** is also desirable since lists are a very common structure. Similarly, unnamed variables could be described by a special PVM instruction such as `VOID`.

Specialization of variable descriptions also provides a number of opportunities to increase efficiency. Instead of initializing variables on entry into a procedure, variables could be initialized on first appearance in the clause and compiled to a PVM instruction called, for example, `FIRST.VAR`. In match mode, such a special description would also save the check to determine the binding of a variable. Consecutive unnamed variables might also be compiled to a single word of one argument.

Finally, one might consider combining `CALL-RETURN` pairs into a single description and compiling the cfa of special-purpose functions directly. Directions for further extensions to the word set are suggested in [CLO85] and [WAR83].

## *Mixing Prolog and Forth*

With the design described here, Forth and Prolog can be mixed freely because the Prolog machine is simulated directly in Forth. Prolog computations can be launched from Forth and Forth computations launched from Prolog. One way to mix the two would be to have the compiler recognize a distinguished functor (possibly **forth**) that would cause the Forth code enclosed in the following parentheses to be compiled in-line in the Prolog clause. For example, the definition of a Prolog procedure that takes a list L and, as a side effect, prints the time taken for a naive reverse of the list might look like:

```
test(L) :- forth(0 COUNT ! START.TIMER),
    nrev(L,L1),
    forth(STOP.TIMER COUNT @ CR ." number of microseconds ").
```

This would compile to the equivalent Forth:

```
1 VAR ENTER
0 COUNT ! START.TIMER
1 VAR 2 VAR 2 nrev CALL
STOP.TIMER COUNT @ CR ." number of microseconds " .
RETURN
```

With this approach, there is no overhead involved in mixed language programming; however, there is some ugliness in the interface. Another possibility is to provide a facility for the declaration of a Prolog interface to Forth. The syntax of such a declaration could be

**forth_predicate(<Forth word>,<Prolog predicate>)**

where the predicate has +'s and -'s in its argument positions to indicate input and output arguments respectively. For example, the declaration

**forth_predicate('TEST',test(+,+,-))**

would specify that a call to the Prolog procedure **test/3** would compile to code that would place the first two arguments on the Forth data stack, execute the Forth word TEST, bind/compare the top of the data stack with the third argument of the call and then either fail or succeed on the basis of the comparison. The cost of this approach is the overhead involved in transferring values between the Prolog control stack and the Forth data stack.

## *Conclusion*

There are two major paths for extensions to the work reported here. The first leads to a very attractive delivery vehicle for real-time expert systems—Forth for the procedural component, Prolog for representation and reasoning. Forth's strengths in real-time applications are well known. Thus, the facility and efficiency with which abstract machines can be simulated in Forth makes the language an ideal platform on which to deliver real-time knowledge-based systems. A marriage of Prolog and Forth is currently being used for this purpose [PAL86]. Given a Forth engine, compiled Prolog on such a platform competes in performance with anything currently available and is likely to be superior in price-performance for a long time to come.

Extensions to the current work on the path leading to a fully formed delivery vehicle include additional PVM instructions of the sort mentioned above under optimizations, better indexing of clauses and more efficient use of control stack space. It would also be worthwhile to simulate the Warren Abstract Machine [WAR83] in Forth to understand the trade-offs between machine complexity and speed. Collaboration with Forth engine vendors could result in hardware features supporting high level languages built on a Forth platform. The Forth engine could even evolve into a Prolog engine.

The second path for extensions begins exploration of issues in computation only touched on by commercial Prolog implementations. The general thrust of the exploration is the extension of unification towards more sophisticated treatment of the objects to be unified. For example, unification in standard Prolog is based solely on the syntactic structure of terms. The language is untyped, and there is no notion of evaluation or co-reference (other than for logical variables). There are, however, numerous illustrations of how type systems (mapping terms into a user-supplied type lattice) can provide tremendous leverage in solving hard problems [WAL85]. Furthermore, absence of evaluation or co-reference in the unifier means that the terms **2 + 2** and **4** won't unify—not satisfactory behavior in an intelligent system.

There are thus proposals to extend Prolog in these and other directions ([KOR83], [SHA83]), and such extensions can be added on top of standard Prolog. Nevertheless, extending the language while keeping it efficient requires extending the underlying virtual machine. One of the interesting facets of the research by Warren, Clocksin and coworkers on Prolog compilation is the emphasis on reducing the problem of compiling Prolog to the problem of finding a concise clause description language. One research question, then, is how to elaborate the clause description language to handle the Prolog extensions in a natural way. The next question is how to build it.

This article is in the spirit of the earlier work on Prolog compilation, taking the position that compiled Prolog is an executable clause description and arguing, therefore, that Forth is a good choice for a PVM implementation language. Forth is an even better choice for compiler prototyping of the type required for exploratory Prolog extensions. Therefore, both of the research questions, developing descriptive locutions and simulating the underlying machine, can be tackled naturally in Forth.

| Extension | Effort | Reference |
|---|---|---|
| Intelligent Backtracking | small | [KUM86] |
| Sorted Logic | small | [WAL85] |
| Logic with Equality | medium | [KOR83] |
| Parallel Logic Programming | large | [CON81] |
| Concept Unification | large | [KAH86] |

Figure 16. Areas of Prolog Machine Extension.

Some specific areas of extension are indicated in Figure 16 along with estimates of the magnitude of the effort involved to extend the Forth code of Appendix A into an effective testbed. Intelligent backtracking may be one of the more straightforward extensions to implement [KUM86], and there is ample scope to develop and test new ideas in this area. Sorts and Kinds are clearly a powerful representation feature and might be implemented naturally by combining this Prolog with the object-oriented systems already available in Forth. Logic with equality would likely require more work than the former extensions, but the task is well bounded with significant theoretical issues to explore. Parallel logic programming requires significant effort, but there may be interesting applications to data acquisition and process control. Concept unification is the most ambitious extension—the general idea involves simulating the capability that people exhibit, for example, to unify the concepts of shoe and hammer in situations where the goal is to put a nail into the wall. Only by trying to make some of these extensions work will enough insight be gained to understand their value.

## *References*

[BOW83] Bowen, D. L., Byrd, L. M., and Clocksin, W. F. (1983). A portable Prolog compiler. Proceedings of the Logic Programming Workshop '83, pp. 74-83. Albuteira, Portugal.

[CLO85] Clocksin, W. F. (1985). Design and simulation of a sequential Prolog machine. *New Generation Computing* 3:101-20.

[CLO81] Clocksin, W. F., and Mellish, C. S. (1981). *Programming in Prolog*. Berlin: Springer-Verlag.

[CON81] Conery, J. S., and Kibler, D. F. (1981). Parallel interpretation of logic programs. In *Proceedings of the conference on functional programming languages and computer architecture*, pp. 163-70. ACM.

[DRE85] Dress, W. B. (1985). A FORTH implementation of the heap data structure for memory management. *J. Forth Appl. and Res.* 3(3): 39-50.

[KAH86] Kahn, K. M. (1986). Uniform: a language based upon unifications which unifies (much of) LISP, Prolog and ACT 1. In *Logic programming. Functions, relations, and equations*, eds. D. DeGroot and G. Lindstrom, pp. 411-38. Englewood Cliffs, NJ: Prentice-Hall.

[KOR83] Kornfeld, W. A. (1983). Equality for Prolog. *Proceedings IJCAI*, pp. 514-19. Los Altos, CA: Morgan Kaufmann.

[KUM86] Kumar, V., and Lin, Y. (1986). An intelligent backtracking scheme for Prolog. Technical report AI TR86-41, University of Texas at Austin.

[ODE86] Odette, L. L., and Wilkinson, W. (1986). Prolog at 20K LIPS on the Novix? *FORML conference proceedings*. San Jose, CA: Forth Interest Group.

[ODE87] Odette, L. L., and Paloski, W. H. (1987). Use of a Forth-based Prolog for real-time expert systems. II. A full Prolog interpreter embedded in Forth. *J. Forth Appl. and Res.*, this issue.

[PAL86] Paloski, W., Odette, L., Krever, A. (1986). A Prolog in Forth for real-time expert systems. *J. Forth Appl. and Res.* 4(2):307.

[PAR84] Park, J. (1984). *MVP-FORTH expert system toolkit*. Mountain View, CA: Mountain View Press.

[SHA83] Shapiro, E. Y. (1983). Logic programs with uncertainties: a tool for implementing rule-based systems. *Proceedings IJCAI*, pp. 529-32. Los Altos, CA: Morgan Kaufmann.

[VAN84] van Emden, M. H. (1984). An interpreting algorithm for Prolog programs. In *Implementations of Prolog*, ed. J. A. Campbell. Chichester, England: Ellis Horwood.

[WAL85] Walther, C. (1985). A mechanical solution of Schubert's steamroller by many-sorted resolution. *Artificial Intelligence* 26(2): 217-25.

[WAR83] Warren, D. H. D. (1983). An abstract Prolog instruction set. Technical note 300, SRI International, Menlo Park, CA.

## *Acknowledgements*

My thanks to Bill Paloski and Martin Tracy for helpful comments on the manuscript.

Manuscript received August 1986.

## *Appendix A: Forth Implementation of the Prolog Virtual Machine*

**Glossary**

The Forth words used to simulate the Prolog machine (screens 29-35) are described below.

`CALL (arity ^atom - )` Takes an integer (arity) and a pointer to the pfa of a constant record and searches the procedure records for a procedure with the specified arity. Backtracks if unsuccessful. Otherwise, if there are multiple applicable clauses, backtracking information is saved: a pointer to the next clause record, a pointer to the next most recent choice-point frame in the control stack and the trail and global stack-pointers. Sets the execution mode to match. Also saves the top of the Forth return stack in the next Prolog control stack frame and initializes the argument pointer to the first argument position in the next control frame (the first memory location after the 12 bytes of control information).

`ENTER (- )` Commits to execution of the body of a clause. Sets execution mode to arg and adjusts control stack pointers.

`RETURN (- )` Indicates success exit. Gets the return pointer saved by `CALL` and pushes it on the Forth return stack. Sets execution mode to arg and adjusts control stack pointers, reclaiming control stack frames where possible.

`CONST (^atom - )` Match mode: tests whether the next argument is a constant whose name is given by atom. Backtracks if they don't match; continues with Forth execution otherwise. Arg mode: builds an argument with tag = 2 and val = ^atom.

`VAR (n - )` Match mode: de-references the variable specified by the input index and then, if the variable is unbound, binds it to the argument; otherwise `VAR` tests whether the argument matches the binding. Backtracks if they don't match; continues with Forth execution otherwise. Arg mode: copies the variable binding to the next argument position.

`FUNCTOR (arity ^atom - )` Match mode: tests whether the next argument is a structured term with the specified arity, whose functor is named by atom. Backtracks if they don't match; otherwise saves the argument pointer and resets it to point to the first argument of the structured term (i.e., to point to the first cell of the argument field of the structure record pointed to by the input argument). Forth execution continues from this point. Arg mode: allocates space for a structure record from the global stack and then sets the first two fields of the record (i.e., functor name and arity). Builds an argument with tag = 3 and val = `^structure`. Saves the argument pointer and resets it to point to the first cell of the argument field of the structure record. Forth execution continues from this point.

`POP (- )` Restores the argument pointer in both execution modes.

## *Appendix B: Prolog-PVM Compiler*

```
/*  =====     =====     =====     =====     =====  */
% COMPILER

% All args are output
% NVars is the number of vars in the clause
% Code is the object code
% Sent is a list of tokens, terminated by the token '.'
% The variable Sent is bound by the procedure "read_in"

compile_clause(Pred/Arity,Nvars,Code) :-
    read_in(Sent),
    horn_clause(Pred/Arity,Vars,[ ],Cd,Sent,_),
    nrev(Cd,Code),memberchkN(-1,Vars,-1,Nvars).

/*  =====     =====     =====     =====     =====  */
% Old is the input code sequence ([ ] here)

horn_clause(Pred/Arity,Vars,Old,['RETURN'|New]) -->
    atmf(Vars,Old,[_,_,_|N1],Pred/Arity),
    (['.'],{New=N1}|
    [':-'],atmfs(Vars,['ENTER'|N1],New ) ).

/*  =====     =====     =====     =====     =====  */

% atmfs are terms that aren't numbers or variables
%  Arg1: list of vars in the clause
%  Arg2: input code sequence
%  Arg3: output code sequence

atmfs(Vars,Old,New ) -->
    atmf(Vars,Old,N1,_),
    (['.'],{New=N1}|[','],atmfs(Vars,N1,New) ).

atmf(Vars,Old,['CALL',Pred,0|Old],Pred/0) -->
    atom_name(Pred).

atmf(Vars,Old,['CALL',Pred,Arity|N1],Pred/Arity) -->
    atom_name(Pred),['('],args(Vars,Old,N1,0,Arity),[')'].

/*  =====     =====     =====     =====     =====  */

% simple_term parses terms, lists and structured terms in
%       functional form
% This is basically a case analysis and is meant to be
%       deterministic, thus the cut
%  Arg1: Variable symbol list (difference list - built
%          during compilation)
%  Arg2: Input Code list (input)
%  Arg3: Output Code list (output)
```

```
simple__term(__,__,__) -->
['.'],!,fail.

simple__term(Vars,Old,['POP'|N1]) -->
    atom__name(Pred),['('],
    args(Vars,['FUNCTOR',Pred,Arity|Old],N1,0,Arity),[')'],!.

simple__term(Vars,Old,New ) -->
    variable(Vars,Old,New),!.

simple__term(__,Old,New ) -->
    constant(Old,New),!.

simple__term(Vars,Old,New ) -->
    ['('],(simple__term(Vars,Old,New ) ;
    conjunction(Vars,Old,New ) ),[')'],!.

simple__term(Vars,Old,New ) -->
    list(Vars,Old,New ),!.

conjunction(Vars,Old,New ) -->
    simple__term(Vars,Old,N1 ),[','],
    (simple__term(Vars,N1,New ) | conjunction(Vars,N1,New ) ).

/*  =====     =====     =====     =====     =====  */

% args parses arguments
%  Arg1: - Arg3: see arguments for simple__term
%  Arg4: arg count (input)
%  Arg5: arg count (output)

args(Vars,Old,New,Num0,Num1 ) -->
    simple__term(Vars,Old,N1),
    ([','],{Num is Num0+1},args(Vars,N1,New,Num,Num1) |
    {New = N1,Num1 is Num0+1}).


/*  =====     =====     =====     =====     =====  */

% char(N,Atom,M)   - M is the N-th character of Atom
% ascii(M,N)       - the ascii code for M is N

atom__name(Name) -->
    [Name],{char(1,Name,M),
    ascii(M,N),ascii(a,__a),ascii(z,__z),__a =< N,N =< __z},!.

variable(Vars,Old,['VAR',Num |Old]) -->
    [Var],{char(1,Var,L),ascii(L,M),
    (ascii('__',M) ;
    ascii('A',__a),ascii('Z',__z),__a =< M,M =< __z),
    memberchkN(Var,Vars,0,Num)},!.
```

```
constant(Old,[Inst,X |Old]) -->
    atom__name(X),{Inst = 'CONST'}|
    [X],{integer(X),Inst = 'INTEGER'}.

list(Vars,Old,New) -->
    ['[',']'],{New=['CONST',nil|Old]} |
    ['['],simple__term(Vars,['FUNCTOR',cons,2|Old],N1),
    list__tail(Vars,N1,New ) .

/*  =====     =====     =====     =====     =====  */
/* Auxiliary relations */

list__tail(Vars,Old,['POP','CONST',nil|Old]) -->
    [']'].

list__tail(Vars,Old,['POP'|New]) -->
    ['|'],simple__term(Vars,Old,New),[']'].

list__tail(Vars,Old,New) -->
    [','],
    simple__term(Vars,['FUNCTOR',cons,2|Old],N1) ,
    list__tail(Vars,N1,New) .

% memberchkN

memberchkN(X,[X|__],M,N) :- N is M+1,!.
memberchkN(X,[__|T],L,N) :- M is L+1,memberchkN(X,T,M,N).

% naive reverse

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

% concatenate

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

*Appendix C: Forth Implementation of PVM*

```
SCR#  2
\  Primitive Macro Facility                              LLO 6/16/86
\  At compile time, copy the code over in-line
\  May not work for control structures
\  RUN-SYMBOLS 16 + @ returns the cfa of EXIT
: ;M [COMPILE] ; IMMEDIATE
   DOES> BEGIN DUP @ DUP RUN-SYMBOLS 16 + @ = NOT
         WHILE , 2+
         REPEAT 2DROP ;  IMMEDIATE
: :M : ;


SCR#  3
\ Stack Definitions                                      LLO 6/16/86
\ ^ARG.STACK is used for argument pointers
\ pushed by FUNCTOR, popped by POP

CREATE STRUCTURE.STACK  6000 ALLOT      \ structure stack
CREATE CONTROL.STACK    6000 ALLOT      \ control stack
CREATE TRAIL            1000 ALLOT      \ trail stack
CREATE ^ARG.STACK       1000 ALLOT      \ arg pointer stack


SCR#  4
\ Prolog Machine Registers                               LLO 6/16/86
\ VALUE is a multiple cfa word
\ access value by name (ie. X vs. X @)
\ store value indicated by IS  ( ie. 2 IS X vs. 2 X !)

0 VALUE CF                              \ Current   Frame
0 VALUE NF                              \ Next      Frame
0 VALUE BF                              \ Backtrack Frame
0 VALUE SS                              \ Structure Stack
0 VALUE TS                              \ Trail  Stack


SCR#  5
\ Register to Register Operations                        LLO 6/16/86

: BF>CF   BF IS CF ;            \ on backtracking
: NF>CF   NF IS CF ;            \ on procedure call
: NF>BF   NF IS BF ;            \ on choice point call
: CF>NF   CF IS NF ;            \ on deterministic return
```

```
SCR#  6
\ Context Save Operations                                   LLO 6/16/86
\ IP> and >IP correspond to R> and >R

:M RP>Stack ( -- )  IP> NF      ! ;M
:  BP>Stack ( n -- )    NF 2+   ! ;
:  CF>Stack ( -- )    CF NF 4  + ! ;
:  BF>Stack ( -- )    BF NF 6  + ! ;
:  SS>Stack ( -- )    SS NF 8  + ! ;
:  TS>Stack ( -- )    TS NF 10 + ! ;


SCR#  7
\ Context Restore Operations                                LLO 6/16/86
:M StackNF>RP ( -- ) IP> DROP NF @ >IP  ;M  \ ret from unit
:M Stack>RP   ( -- ) IP> DROP CF @ >IP  ;M
:  Stack>BP ( -- n)  CF 2+   @          ;
:  Stack>CF ( -- )    CF 4  + @ IS CF   ;
:  Stack>BF ( -- )    CF 6  + @ IS BF   ;
:  Stack>GS ( -- )    CF 8  + @ IS SS   ;
:  Stack>TS ( -- )    CF 10 + @ IS TS   ;


SCR#  8
\ Temporaries, Equates, Tags                                LLO 6/16/86
\ Temporary Variables
0 VALUE ARG                             \ Argument Pointer
0 VALUE NVARS                           \ cache
0 VALUE ARITY                           \ cache
  VARIABLE ARG.FLG                      \ arg-match mode flag
  VARIABLE COPY.FLG                     \ copy mode flag
\ Equates
 4  CONSTANT BYTES/CELL
12  CONSTANT BYTES/FRAME
\ Tags  -  Here the tag is the high order 16 bits
 1  CONSTANT VAR.TYPE
 2  CONSTANT CONST.TYPE
 3  CONSTANT FUNCT.TYPE


SCR#  9
\ Record Manipulation                                       LLO 6/16/86
\ Code records
: CODE>ARITY   ( ^code -- n )    2+  C@ ;
: CODE>NVARS   ( ^code -- n )    3 + C@ ;
: CODE>PROC    ( ^code -- addr ) 4 +    ;

\ Procedure Records
: PROC>CODE  ( ^clause -- ^code )  4 + @ ;   \ procedure > code
: PROC>ARITY ( ^clause -- n ) 2+ @ ;         \ procedure > arity
```

```
SCR# 10
\ Record Manipulation                                    LLO 7/20/86
\ Structure Records
: STRUC>FUNCTOR ( ^functor -- ^atom )     @ ;     \ func > name
: STRUC>ARITY   ( ^functor -- arity ) 2+ @ ;     \ func > arity
: STRUC>ARGS    ( ^functor -- ^args ) 4 +  ;     \ func > args

\ Term References
: >TYPE         ( term.ref -- type  )     @ ;     \ ref > type
: >POINT        ( term.ref -- ^term ) 2+ @ ;     \ ref > pointer


SCR# 11
\ Procedure Search                                       LLO 6/16/86
: FIND.PROC ( n pfa -- ^code | flag )
\ Find a procedure of given arity and functor
\ arity = n, functor = pfa
\ Return FALSE if not found
\ Return pointer to first code record if found
  BEGIN
  @ DUP                          \ get pointer to clause records
  IF 2DUP PROC>ARITY =           \ compare arity
     IF TRUE ELSE FALSE THEN     \ if = then we're done
  ELSE  TRUE THEN                \ if link = 0 then we're done
  UNTIL SWAP DROP DUP            \ clean stack
  IF PROC>CODE TRUE THEN ;       \ convert to code record pointer


SCR# 12
\ Variable Manipulation                                  LLO 6/16/86
: CREATE.UNBOUND.VAR ( addr -- )
\ Create an unbound variable at addr
  VAR.TYPE OVER 2! ;     \ store
: INIT.VARS ( -- )
\ create unbound variables in the control frame
  NVARS ?DUP
  IF    NF BYTES/FRAME +
        ARITY BYTES/CELL * +   DUP ROT BYTES/CELL * + SWAP
        DO    I CREATE.UNBOUND.VAR   BYTES/CELL +LOOP  THEN ;
: RESET.VARS ( top.TS bottom.TS -- )
\ reset the variables on the trail stack
  2DUP =  IF 2DROP  ELSE DO I @ VAR.TYPE OVER 2! 2 +LOOP  THEN ;
```

```
SCR# 13
\ Saving State                                          LLO 6/16/86
:M  RESET.RP ( ^code -- )
\ Reset the return stack pointer
  IP> DROP CODE>PROC >IP ;M
: SAVE.BACKTRACK ( ^code -- ^code )
\ Save appropriate information at a backtrack point
  DUP @ DUP BP>Stack     \ get link to next code record
  IF BF>Stack            \ if a choice point, save old BF
     NF>BF               \ current frame is new backtrack frame
     SS>Stack TS>Stack   \ save stack pointers
  THEN ;


SCR# 14
\ Adjust Pointer to Next Frame                          LLO 6/16/86
: SET.NF ( ^code -- ^code )
\ reset NF register
  CODE>ARITY BYTES/CELL * BYTES/FRAME + CF + IS BF ;
: RESET.NF ( -- )
\ reset NF pointer
  NF BYTES/FRAME +
  ARITY NVARS + BYTES/CELL * +
  IS NF ;                        \ adjust next frame pointer
: INIT.^ARG ( -- )
\ init ARG register
  NF BYTES/FRAME + IS ARG ;


SCR# 15
\ General Backtracking                                  LLO 6/16/86
: BACKTRACK ( -- )
\ Restore context
\ BF>CF                          \ make the BF current
\ Stack>BP DUP RESET.RP          \ get pointer to code record
\ @ DUP BP>Stack 0=              \ get link to next code record
\ IF Stack>BF THEN               \ reset BF if this not choice pt
\ Stack>PF Stack>GS              \ restore PF and SS pointers
\ TS Stack>TS                    \ restore TS
\ TS RESET.VARS                  \ reset vars on the trail
\ INIT.^ARG
\ SET.NF                         \ reset NF
  ." not implemented " ;
```

```
SCR# 16
\ Backtracking on Unification Failure
: RETRY ( -- )
\ Restore context
  CF                            \ get a copy of CF
  BF>CF                         \ make the BF current
  Stack>BP DUP RESET.RP         \ get pointer to code record
  DUP CODE>NVARS IS NVARS
  @ DUP BP>Stack 0=             \ get link to next code record
  IF Stack>BF THEN              \ reset BF if this not choice pt
  Stack>GS                      \ restore SS pointer
  TS Stack>TS                   \ restore TS
  TS RESET.VARS                 \ reset vars on the trail
  INIT.VARS  INIT.^ARG
  IS CF    ;                    \ restore CF


SCR# 17
\ Binding Trail                                          ... 7/26/86
\ keep track of variable bindings that may need
\ to be reset on backtracking

: >TRAIL ( ^var -- )
\ put a variable cell address on the trail
  TS !  TS 2+ IS TS ;

: >TRAIL? ( ^var -- ^var )
\ trail a variable if necessary
  DUP BF U<  OVER SS U<  OR
  IF DUP >TRAIL THEN ;


SCR# 18
\ Creating and Fetching a Binding                        ... 7/26/86
: BIND ( ^term type ^var -- )
\ bind a variable
  >TRAIL? 2! ;           \ smash variable cell

: >ULT.BINDING ( ^term0 -- ^term1 )
\ completely dereference a variable binding
  BEGIN
  DUP >TYPE VAR.TYPE =                  \ check for variable
   IF DUP >POINT OVER =                 \ check if unbound var
     IF TRUE ELSE >POINT FALSE THEN     \ chase pointer if bound
   ELSE TRUE THEN                       \ not variable then done
  UNTIL ;
```

```
SCR# 19
\  Save and Restore Arguments                              LLO 6/16/86

: POP.ARG ( -- ^term )
\ pop an argument off control/structure stack
\ leave term pointer on data stack
  ARG DUP BYTES/CELL + IS ARG ;
: PUSH.ARG ( ^term type -- )
\ push an argument onto the control/structure stack
  ARG 2!  ARG BYTES/CELL + IS ARG ;


SCR# 20
\ Save and Restore Arg Pointers                           ... 7/26/86
: PUSH.^ARG ( addr -- )
\ save then reset argument pointer
  2 ^ARG.STACK  ARG OVER @ !  +!  IS ARG ;
: POP.^ARG ( -- )
\ restore argument pointer
  ^ARG.STACK -2 OVER +!   @ @ IS ARG  ;

: ARG&TYPE ( -- ^term type )
\ pop an argument, dereference it if necessary, return with type
  POP.ARG >ULT.BINDING DUP >TYPE ;


SCR# 21
\ Unify Variable (match mode)                             LLO 7/20/86
\ see VAR, screen 33
: UNIFY.VAR  ( ^var -- flag )
\ unify the variable with an argument
  ARG&TYPE  VAR.TYPE =          \ get the argument
  IF 2DUP U> IF SWAP THEN 2+ !  \ bind variables
  ELSE 2@ ROT BIND THEN  TRUE ; \ bind it to the variable


SCR# 22
\ Unify Constant                                          ... 7/22/86
\ see VAR, screen 33
: UNIFY.CONST ( ^const -- flag )
\ unify the constant with an argument
  ARG&TYPE  CASE                    \ get the argument
  VAR.TYPE
   OF >R 2@ R> BIND TRUE ENDOF
  CONST.TYPE
   OF >POINT SWAP >POINT =          \ compare pointers
       IF TRUE ELSE FALSE THEN      \ succeed if pointers match
  ENDOF
  SWAP FALSE ENDCASE   ;            \ nothing else recognized
```

```
SCR# 23
\ Unify Structure                                    LLO 7/20/86
\ see VAR, screen 33

: UNIFY.FUNCT ( ^functor -- flag )
\ unify the functor with an argument
  ARG&TYPE  CASE                                \ get the argument
  VAR.TYPE   OF >R 2@ R> BIND        TRUE ENDOF
  FUNCT.TYPE OF ." Not Implemented " TRUE ENDOF
  SWAP FALSE ENDCASE ;


SCR# 24
\ Build a Term Reference (arg mode)                  LLO 7/20/86

: REF>ARG ( ^term -- )
\ builds an argument from a term reference
  DUP >TYPE VAR.TYPE =          \ check for variable
  IF VAR.TYPE                   \ make a new variable
  ELSE 2@ THEN  PUSH.ARG ;      \ otherwise copy
: VAR>ARG  ( n -- )
\ builds an argument from a variable reference
  COPY.FLG @
  IF NF ELSE CF THEN  +         \ get address of variable
  >ULT.BINDING  REF>ARG  ;      \ get binding for argument


SCR# 25
\ Build a Structure Record (copy, arg modes)         LLO 7/20/86

: COPY.FUNCT ( arity ^atom -- )
\ build a functor record on the structure stack
  -1 ARG.FLG +!                 \ increment the counter
  OVER BYTES/CELL * 4 +         \ compute # of bytes in record
  SS DUP ROT + IS SS            \ allocate space
  DUP 4 +  PUSH.^ARG            \ reset arg pointer for rest
  2! ;                          \ fill head of functor record


SCR# 26
\ Match a Structure (match mode)                     LLO 7/20/86
\ see FUNCTOR, screen 35
: MATCH.FUNCT ( arity ^atom ^term -- flag )
\ match functor with an argument
\ ARG is reset if functor (name )  and arity match
\ remainder of match is handled by code in clause head
  >POINT                         \ get pointer to record
  DUP STRUC>FUNCTOR ROT =        \ check functor match
   IF DUP STRUC>ARITY ROT =      \ check arity match
    IF STRUC>ARGS PUSH.^ARG TRUE \ reset arg pointer, return T
    ELSE DROP FALSE THEN
   ELSE 2DROP FALSE THEN ;       \ return false if no match
```

```
SCR# 27
\ Copy a Structure (copy mode)                          LLO 7/20/86
\ see FUNCTOR, screen 35
: MATCH.VAR ( arity ^atom ^var -- flag )
\ Builds a structure record and binds it to the variable
\ The structure args are built by the remainder of the head code
  COPY.FLG ON                    \ copy variable from NF not CF
  SS FUNCT.TYPE ROT BIND         \ bind the variable to funct rec
  COPY.FUNCT    TRUE ;           \ set to copy functor, return T


SCR# 28
\ Prolog Virtual Machine Instructions
\ Note: VAR takes a byte offset from the start of a frame
\ (base address) to compute the address of the variable.
\ ARG.FLG = 0, COPY.FLG = 0 indicates match mode
\ ARG.FLG not 0, COPY.FLG = 0 indicates arg mode
\ ARG.FLG not 0, COPY.FLG not 0 indicates copy mode
\ in copy mode, nesting is handled by decrementing ARG.FLG
\ at the start of a structure, and incrementing on exit
\ (via POP).
\ COPY.FLG is used to indicate which frame pointer (CF or NF)
\ should be used for the base address of the variable.


SCR# 29
\ Prolog Machine Instruction    CALL                    LLO 6/16/86
: CALL ( n cfa -- )
\ Call a Prolog procedure
  ARG.FLG OFF                    \ swith mode to "match"
  FIND.PROC                      \ get pointer to procedure rec
  IF DUP CODE>NVARS IS NVARS     \ cache NVARS
     DUP CODE>ARITY IS ARITY     \ cache ARITY
     INIT.VARS RP>Stack          \ init vars and set up stack
     SAVE.BACKTRACK
     INIT.^ARG                   \ set argument pointer
     4 + >IP EXIT                \ pass control to procedure
  ELSE BACKTRACK THEN   ;        \ backtrack if no procedure


SCR# 30
\ Prolog Machine Instruction    ENTER                   LLO 6/16/86

: ENTER ( -- )
\ Enter a procedure and begin execution of the body
  ARG.FLG ON                     \ switch mode to "arg"
  COPY.FLG OFF
  CF>Stack                       \ adjust frame pointers
  NF>CF RESET.NF                 \ adjust next frame pointer
  INIT.^ARG ;                    \ set arg pointer for next call
```

```
SCR# 31
\ Prolog Machine Instruction ENTER                    LLO 6/16/86
: RETURN ( -- )
\ Return from a procedure
  ARG.FLG @                        \ check mode
  IF Stack>RP  CF BF U>
   IF CF>NF THEN                   \ reclaim if not backtrack pnt
   Stack>CF                        \ adjust frame pointers
  ELSE StackNF>RP  BF NF =         \ if ret from unit clause
   IF CF>Stack                     \ save parent frame pointer
      RESET.NF                     \ reset frame pointer
   THEN
  THEN ARG.FLG ON                  \ turn off matcher
  INIT.^ARG ;                      \ set arg pointer for next call


SCR# 32
\ Prolog Machine Instruction  CONST                   LLO 6/16/86
: CONST ( ^atom -- )
\ match or copy a constant
  ARG.FLG @
  IF CONST.TYPE PUSH.ARG                  \ push arg in arg mode
  ELSE ARG&TYPE VAR.TYPE =                \ get first argument
      IF CONST.TYPE SWAP BIND             \ if variable, bind it
      ELSE >POINT = NOT                   \ otherwise must be EQ
       IF IP> DROP    NF BF =
          IF RETRY                        \ retry if new call
          ELSE BACKTRACK THEN             \ backtrack if not
       THEN
      THEN                                \ else continue
  THEN          ;


SCR# 33
\ Prolog Machine Instruction  VAR
: VAR ( n -- )
  ARG.FLG @
  IF   VAR>ARG                    \ get binding for argument
  ELSE NF + >ULT.BINDING          \ get the variable bindings
   DUP >TYPE  CASE                \ case analysis on type
     VAR.TYPE   OF UNIFY.VAR   ENDOF
   CONST.TYPE   OF UNIFY.CONST ENDOF
   FUNCT.TYPE   OF UNIFY.FUNCT ENDOF
   ENDCASE      NOT
   IF R> DROP  NF BF =            \ if match not successful
      IF RETRY                    \ retry
      ELSE BACKTRACK THEN         \ or backtrack
   THEN
  THEN ;                          \ build an argument
```

```
SCR# 34
\ Prolog Machine Instruction  POP                    LLO 7/20/86

: POP ( -- )
\ pop from a FUNCTOR
  POP.^ARG                      \ restore argument pointer
  ARG.FLG @                     \ look for "arg" mode
  IF 1 ARG.FLG +! THEN ;        \ decrement counter


SCR# 35
\ Prolog Machine Instruction  FUNCTOR                LLO 7/20/86
: FUNCTOR ( arity ^atom -- )
\ Compiler object indicating a structure
  ARG.FLG @
  IF SS FUNCT.TYPE PUSH.ARG   COPY.FUNCT
  ELSE ARG&TYPE CASE
     VAR.TYPE OF MATCH.VAR   ENDOF
   FUNCT.TYPE OF MATCH.FUNCT ENDOF
   FALSE SWAP ENDCASE
   NOT IF R> DROP NF BF =
        IF RETRY ELSE BACKTRACK THEN
       THEN
  THEN ;


SCR# 36
\ Support Routines
\
\ ASSERTZ is the PVM to Forth Compiler.
\ It requires as parameters the number of variables
\ in the clause, the arity of the clause, and the pfa
\ of the clause functor.
\ If the PVM word set were extended, with different words
\ for instructions in the head and body, much of the compilation
\ to the extended word set could be handled by ASSERTZ, since
\ it can tell the difference between the head and body of a
\ clause.  Thus the Prolog-PVM compiler could stay simple.
\


SCR# 37
\ Prolog Print Words
  DEFER PRINT.TERM
: PRINT.CONST ( ^const -- )
\ print a constant
  >POINT BODY> .ID ;
: PRINT.VAR ( ^var -- )
\ print a variable
  >POINT BASE @ >R
  ASCII _ EMIT HEX U.   R> BASE ! ;
```

```
SCR# 38

: PRINT.FUNCT ( ^term -- )
\ print a structure
  >POINT DUP @ BODY> .ID  ASCII ( EMIT
  2+ DUP 2+ SWAP @ BYTES/CELL * OVER + SWAP
  DO I PRINT.TERM BYTES/CELL +LOOP  ASCII ) EMIT ;
: <PRINT.TERM> ( ^term -- )
\ print a Prolog term
  >ULT.BINDING DUP >TYPE CASE
  CONST.TYPE OF PRINT.CONST  ENDOF
    VAR.TYPE OF PRINT.VAR    ENDOF
  FUNCT.TYPE OF PRINT.FUNCT  ENDOF  ENDCASE ;
' <PRINT.TERM> IS PRINT.TERM


SCR# 39
\ Auxiliary Words                                    LLO 6/16/86
: !CODE.DATA ( nvars arity -- )
\ pack the number of variables (nvars) and arity
\ then enclose
  256 * + ,  ;              \ arity is in high order byte
: \LINK ( start.addr -- end.addr )
\ traverse links to end
  BEGIN DUP @ IF @ FALSE ELSE TRUE THEN UNTIL ;


SCR# 40
\ PVM to Forth Compiler                          LLO 6/16/86
: :ASSERTZ ( nvars arity pred -- )
\ add a clause to the Prolog data base
  2DUP FIND.PROC                \ find proc record
  IF \LINK                      \ if found, get last clause rec
     HERE SWAP ! 0 ,            \ install links
     DROP !CODE.DATA            \ install nvars, arity
  ELSE \LINK                    \ if not found get last proc
     HERE SWAP ! 0 ,            \ install links
     DUP , HERE 2+ ,            \ install arity and link to clause
     0 , !CODE.DATA             \ set clause link
  THEN STATE ON  ;              \ compile remainder of input
: ;ASSERT ( -- )
  STATE OFF ; IMMEDIATE         \ turn off compilation
```

```
SCR# 41
\ Initialization words for test routines              LLO 6/16/86
: INIT.^ARG.STACK
\ init argument pointer stack
  ^ARG.STACK DUP 2+ SWAP ! ;
: INIT.STACKS
\ initialize stacks prior to CALL
  STRUCTURE.STACK IS SS
  TRAIL IS TS
  CONTROL.STACK IS CF
  CF IS NF  0 IS BF
  ARG.FLG OFF COPY.FLG OFF
  CONTROL.STACK 3000 0 FILL     INIT.^ARG.STACK ;
: TEST INIT.STACKS CR ;


SCR# 42
\ Test Procedures
\ Create Dictionary Entries for all Test Words

CREATE REV   0 ,       CREATE APPEND 0 ,
CREATE NIL   0 ,       CREATE cons   0 ,
CREATE bob   0 ,       CREATE art    0 ,    CREATE fred 0 ,


SCR# 43
\ APPEND and REV    (Note arguments of VAR)
1 3 APPEND :ASSERTZ NIL CONST 24 VAR 24 VAR RETURN ;ASSERT
4 3 APPEND :ASSERTZ 2 cons FUNCTOR 24 VAR 28 VAR POP
                    32 VAR
                    2 cons FUNCTOR 24 VAR 36 VAR POP
            ENTER   28 VAR 32 VAR 36 VAR 3 APPEND CALL
            RETURN  ;ASSERT
0 2 REV    :ASSERTZ NIL CONST NIL CONST RETURN ;ASSERT
4 2 REV    :ASSERTZ 2 cons FUNCTOR 20 VAR 24 VAR POP 28 VAR
            ENTER   24 VAR 32 VAR 2 REV CALL
                    32 VAR
                    2 cons FUNCTOR 20 VAR NIL CONST POP 28 VAR
                    3 APPEND CALL
            RETURN  ;ASSERT


SCR# 44
\ Test of Append
\ use: TEST 0 TEST1 CALL
\ check result with CONTROL.STACK 12 + PRINT.TERM
CREATE TEST1  0 ,
\ append([bob,fred],[art],L).
1 0 TEST1 :ASSERTZ  ENTER
             2 cons FUNCTOR bob CONST
             2 cons FUNCTOR fred CONST NIL CONST POP POP
             2 cons FUNCTOR art  CONST NIL CONST POP
             12 VAR 3 APPEND CALL   RETURN ;ASSERT
```

```
SCR# 45
\ Test of REV
\ use: TEST Ø TEST2 CALL
\ check result with CONTROL.STACK 12 + PRINT.TERM
CREATE TEST2 Ø ,
\ rev([bob,art,fred],L).
1 Ø TEST2 :ASSERTZ  ENTER
            2 cons FUNCTOR bob CONST
            2 cons FUNCTOR art  CONST
            2 cons FUNCTOR fred CONST NIL CONST
                   POP POP POP
            12 VAR 2 REV CALL RETURN ;ASSERT
```