eu-Prolog

Reference Manual and Report

April, 1984

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 155

eu-Prolog

April, 1984

by

Eugene Kohlbecker

This material is based on work supported by the National Science Foundation under grant MCS 83-04567.

Table of Contents

Preface 1
1. Introduction 3
2. Syntax 6
3. Top-level Operations 7
4. Core operations 12
5. Primitives 19
6. Meta-circular Interpreter 21
7. Annotated Implementation 23
Bibliography 28

Preface

This report describes the use and implementation of eu-Prolog, a dialect of Prolog. It is a reference manual and user-guide for that system; a prior understanding of the Prolog unification and backtracking procedures are assumed.

eu-Prolog is not a full-blown Prolog system; much of what appears in a standard Prolog system is missing. Rather, this version is an educational tool designed for the teaching of the basics of programming in Prolog. It was completed as a project for Professor John Barnden's course Artificial Intelligence II.

Various implementations of Prolog dialects have been produced in Lisp. For examples, see Wallace's paper [Wallace 1983] and Komorowski's thesis [Komorowski 1981]. Also, Wand has produced an embedding of the Prolog unification and control capabilities in Scheme [Wand 1983]. Instead of being an embedding of Prolog within Scheme, eu-Prolog is a true Prolog that runs on top of Scheme and Franz Lisp [Foderaro *et al.* 1983].

Even though these other implementations and embeddings exist, an implementation of Prolog in Scheme 84 [Friedman *et al.* 1984] is interesting because of the convenience of that language's first-class continuations. Both backtracking and the cut are understood as invocations of continuations; each is seen to be the resumption of an earlier state of the Prolog interpreter. An interesting question for further thought is, are there other places in the Prolog control mechanism that would provide useful continuations for different computations.

The files which contain the eu-Prolog code are in the iuvax directory

"/usiu/eugene/prolog".

They are

prolog.s	system initialization and loading
prolog.l	Franz Lisp functions for eu-Prolog
prolog.o	compiled version of prolog.l
protop.s	read-execute-print loop and top-level command interpreter
exe.s	eu-Prolog interpreter

To load eu-Prolog, enter Scheme 84 and load the file

"/usiu/eugene/prolog/prolog.s".

You should see

```
Scheme 84 version 0 (27-Feb-84)

>>> (load 'prolog.s)
[fasl /usiu/eugene/prolog/prolog.o]
t
>>>
```

To run eu-Prolog, invoke the Scheme 84 **prolog** function:

```
>>> (prolog)

[Return to top level]
eu-Prolog (version 27-Feb-84)

prolog>
```

1. Introduction

This section presents an overview of the eu-Prolog system. It includes a discussion of the conventions used in this report. A brief demonstration of the use of the system is provided.

The portion of the code that is written in Lisp deals with the unification algorithm—it involves only list processing. Originally written in Scheme, it was translated into Lisp so that faster execution speeds could be obtained. However, because it is desirable to use the power of Scheme's first-class continuations and lexical closures, the actual Prolog control mechanism was written in Scheme 84.

To simplify the implementation, a Lisp-like syntax was chosen, eschewing the brackets, commas, and vertical bars of other Prolog systems. This means that the *list* data structure is represented as in Lisp: with parentheses and occasional uses of "." (Lisp pair notation). The truly concerned can, of course, fiddle with the Franz Lisp reader and alter this aspect of the syntax.

Furthermore, all top-level operations and primitives are written in prefix, rather than infix, notation. There are no infix arithmetic operators in eu-Prolog.

As is customary in some Prolog implementations, identifiers that begin with a capital letter are regarded as *variables*. Those with lower case initial letters are reserved as the names of top-level operations, primitives, and user-defined procedures and data.

One of the most important aspects in discussing the behavior of a system is defining the terms that are to be used. The grammar presented in the next section will serve as the formal definition of terms used to describe parts of Prolog expressions. Other technical terms are be defined informally here.

A Prolog procedure definition is an *assertion list*. In eu-Prolog all assertions are created with the ":-" operator. This is true for program text as well as data that are to be entered into the *store*. When a procedure is *called*, the appropriate definition is retrieved from that store.

The assertion that eu-Prolog is currently trying to instantiate is its *goal*. The assertion name is always the head of the list that represents the goal. The assertions in the assertion list are matched, one by one with the goal until one is found whose first clause *unifies* with the goal. The remaining clauses are taken as a list of subgoals that must all be instantiated in order to successfully instantiate the goal. If all the subgoals cannot be instantiated, that assertion fails, and further attempts are made at unifying another head clause with the goal. This means that the order of the assertions in the assertion list is important; Prolog systems conduct their search for valid instantiations in the fixed order specified by the assertion list.

Moreover, the subgoals are processed in order. Should a failure be encountered for any subgoal, the system *backtracks* to the previous subgoal, attempting to find another valid instantiation of it. If one is found, processing continues with the subgoal list in order from the newly instantiated goal. However, it might be that there are no subsequent valid instantiations, and so, the system would backtrack another step. In this fashion, it is possible to backtrack out of the entire subgoal list. If this happens, the assertion has failed, and, as already stated, the search continues with the next assertion in the assertion list.

The formal result of a successful procedure call is a pair: an *environment* and a *failure continuation*. The environment is in the form of a most-general unifier that when applied to the original procedure call will yield a valid instantiation of the relation described by the call. The system never actually prints this environment; instead, for top-level calls—those made directly by the user—the instantiated goal is displayed. The failure continuation is also not printed. It remains in the background, ready to restart the computation should backtracking be triggered. Thus, the internal result of any procedure call is in a form suitable for use with subsequent calls. Only at top-level does it seem that a successful call produces a valid instantiation of that call.

As implied, every call takes place with an environment and a failure continuation in effect at the time of the call. The top-level read-execute-print loop provides these two values for user calls. For calls encountered during the execution of any procedure, the system uses the environment and continuation that resulted from the immediately preceding successful call, extending them both to produce a new result.

If a call cannot be instantiated or if the user requests further top-level attempts at instantiation, the current failure continuation is invoked. This failure continuation is a Scheme procedural abstraction of the backtracking mechanism; invoking it means to backtrack, using the standard search strategy of Prolog systems.

If no more backtracking is possible, that is, all the search-tree branches have been fully explored, then the initial failure continuation is invoked. This special continuation prints the word `fail` on the terminal and gives the user a new top-level prompt.

Using eu-Prolog. After loading eu-Prolog into Scheme 84 and invoking it by calling the Scheme function `(prolog)`, the system is ready to accept procedure definitions. As a simple example of the use of the system, consider the **append** procedure.

```
prolog> (:- (append nil A A))
append

prolog> (:- (append (A . B) C (A . D))
            (append B C D))
append
```

The system is now ready to accept calls to **append**:

```
prolog> (append (e f) (g (h)) Ans)
(append (e f) (g (h)) (e f g (h)))
;
```

The semicolon prompt indicates that there is an awaiting failure continuation, prepared to investigate via backtracking the rest of the search-tree. The user can either **stop**, returning to the top-level read-execute-print loop, or can tell the system to continue, with **go**:

```
; go
fail

prolog>
```

In this example, there were no further valid instantiations to be found. The system wrote the `fail` and supplied another prompt.

Below is another call to **append**, illustrating the system's behavior when there are multiple valid instantiations and the use of **stop**.

```
prolog> (append L1 L2 (a (b c) d))
(append nil (a (b c) d) (a (b c) d))
; go
(append (a) ((b c) d) (a (b c) d))
; go
(append (a (b c)) (d) (a (b c) d))
; stop

prolog>
```

2. Syntax

For those who like this sort of thing, this section presents an extended Backus-Naur Form (BNF) grammar for assertions in this dialect of Prolog. Non-terminal symbols within braces indicate zero or more occurrences of that symbol.

```
assertion ::= (:- simple-clause {clause} )

clause ::= simple-clause
     | (call clause)
     | (not clause)
     | (asserta clause)
     | (assertz clause)
     | (retract clause)
     | (write clause) | (write term)
     | !
     | fail

simple-clause ::= (assertion-name {term} )
     | ( variable {term} )

assertion-name ::= primitive-name | user-defined-name

term ::= constant | variable | data-object

data-object ::= ( {term} ) | ( term {term} . term )

primitive-name ::= add | mult | < | > | atom | is-def
```

All variables are Scheme identifiers that begin with an upper case letter. Any other Scheme atomic symbol is treated as a constant. Data-objects have the same syntax as Lisp lists.

The keywords `call`, `not`, `asserta`, `assertz`, `retract`, `!` (the cut), `fail`, and `write` make up the *core* of eu-Prolog. They perform special operations affecting the store and the flow of control during the execution of a procedure. `write` is a simple output operation.

3. Top-level Operations

This section describes the top-level operations that eu-Prolog recognizes. A top-level command is anything that the user types to the system except for procedure calls. The eu-Prolog *top-level* is indicated by either the "**prolog>** " or semicolon prompt. The semicolon prompt indicates that there is currently a non-top-level failure continuation.

The commands allow creating, deleting, editing, and listing assertions; forcing backtracking and aborting the current search; saving of the store on a file; loading of a file containing prolog assertions; and returning to Scheme 84.

What distinguishes these top-level operations from the core operations of eu-Prolog? Calls to top-level operations may not contain variables other than those used inside procedure definitions, nor can they appear within the body of a procedure; core operations can do both. Top-level operations are not goals to be instantiated; they serve only to provide the user with an interface to the underlying Scheme 84 and host operating system. Core operations are assertions, designed to be evaluated just as if they were defined by a user.

1. Defining assertions and making procedure calls. The "`:-`" operator is eu-Prolog's means of entering both programs (i.e., prolog procedures) and data into the global store. The appropriate syntax is given in the BNF grammar of Section 2, and an example of its use appears in Introduction, Section 1. Here, another example and some discussion are provided.

Consider the procedure `member`. An instantiation of

`(member A B)`

is valid if either `A` is the first element of `B` or if `A` occurs in the tail of `B`. The procedure `member` is entered as

```
prolog> (:- (member A (A . B2)))
member

prolog> (:- (member A (B1 . B2))
            (member A B2))
member

prolog>
```

eu-Prolog responds to each assertion with the name of the procedure or data begin defined. Also, note that the order in which the assertions are made is relevant; as with all Prologs, eu-Prolog will consider the elements of the store in the order they were entered while it is looking for a valid instantiation.

Another important thing to observe is that in the first assertion,

`(:- (member A (A . B2)))`,

there is only one clause. This kind of assertion represents a tautology—it is always valid.

To call a procedure, the clause to be instantiated is typed at the prompt. If eu-Prolog does not recognize the input as either a top-level command or a core operation, it attempts to instantiate the goal. Unsuccessful attempts result in the printing of `fail`; successful ones result in the printing of the valid instantiation and the semicolon prompt to indicate that there is a non-empty backtracking continuation.

Examples of the use of `member` follow.

```
prolog> (member b (b a d c a t))
(member b (b a d c a t))
; go
fail
```

This succeeded once because `b` occurs in the list `(b a d c a t)`. The attempt at finding subsequent valid instantiations made by typing the command `go` failed.

```
prolog> (member a (b a d c a t))
(member a (b a d c a t))
; go
(member a (b a d c a t))
; go
fail
```

This succeeded twice. The next call has no valid instantiations.

```
prolog> (member e (b a d c a t))
fail
```

Prolog procedures can be used to determine valid variable bindings in the returned environment:

```
prolog> (member A (b a d c a t))
(member b (b a d c a t))
; go
(member a (b a d c a t))
; go
(member d (b a d c a t))
; go
(member c (b a d c a t))
; go
(member a (b a d c a t))
; go
(member t (b a d c a t))
; go
fail
prolog> (member a L)
(member a (a . G00147))
; go
(member a (G00149 a . G00152))
; go
(member a (G00149 G00154 a . G00157))
; go
(member a (G00149 G00154 G00159 a . G00162))
; stop
```

The *gensym*'d variables (e.g., **G00149**) in the last example represent eu-Prolog's way of saying that any environment binding of the variable will result in a valid instantiation. Because there are infinitely many valid instantiations of the last call, the example was terminated by the **stop** command.

In summary, "`:-`" is used to define assertions; calls to procedures are made by typing the eu-Prolog clause that is to be instantiated; subsequent attempts at instantiating the same goal are forced by the **go** command; and the discarding of all future backtracking is made by the **stop** command. **go** and **stop** have no effect at top-level:

```
prolog> go
fail

prolog> stop

prolog>
```

2. Manipulating assertions. eu-Prolog provides three commands to alter and verify collections of assertions associated with a given name: `listing`, `edit`, and `-:` (the removal of all assertions).

To list on the terminal all assertions associated with a given name, type the command

`(listing` *name*`)`.

For example,

```
prolog> (listing member)

(member A (A . B2)) :-
(member A (B1 . B2)) :-
   (member A B2)

prolog>
```

The assertions are listed in the order they were made using an infix "`:-`" rather than prefix notation. This has the effect of reducing the number of parentheses needed in the display. The null right-hand side of the first assertion indicates that it is a tautology.

If the *name* has no assertions associated with it in the store, that information is reported:

```
prolog> (listing mem)

[Prolog message---no assertions: mem]

prolog>
```

To alter the set of assertions without removing them altogether and retyping them, an interface to the Franz Lisp structure editor is provided. To access it use the command

`(edit name).`

For example,

```
prolog> (edit member)
edit
# pp

(((member A (A . B2))) ((member A (B1 . B2)) (member A B2)))
# (r A X)

((& ) (&  & ))
# pp

(((member X (X . B2))) ((member X (B1 . B2)) (member X B2)))

# ok
prolog> (listing member)

(member X (X . B2)) :-
(member X (B1 . B2)) :-
   (member X B2)

prolog>
```

The list structure being edited is is the form of a list of assertions, each with the ":-" symbol removed. As long as the user is careful about syntax, the set of assertions can be rearranged, extended, or diminished as the user sees fit.

As in Franz Lisp, if the editor is exited with the **stop** command instead of **ok**, the changes made during the editing session will not be preserved.

If the user desires to remove all the assertions associated with a given name, the command

`(-: name)`

is used. For example,

```
prolog> (-: append)
[Prolog message---removed all assertions: append]

prolog> (listing append)

[Prolog message---no assertions: append]

prolog>
```

3. Saving and loading eu-Prolog *files.* The system provides a means of saving the entire global store in a file and of loading a file which contains eu-Prolog assertions.

To save the entire store, use the command

`(save` *filename*`)`

as in

```
prolog> (save "foo.pro")
foo.pro

prolog>
```

The file will not be pretty-printed, but will be in a form suitable for later use by `load`.

To restore a saved store or to load a file of eu-Prolog assertions created in some other fashion, use the command

`(load` *filename*`)`.

For example,

```
prolog> (load "foo.pro")
foo.pro

prolog>
```

When using either `save` or `load`, it is not necessary to supply a double-quoted string as *filename*. As long as the name contains only characters which are legal within Scheme 84 atoms, the user can type the name without the quotes.

`(load foo.pro)`

works just as well as the string version shown in the example.

4. Exiting eu-Prolog. For returning to the underlying Scheme 84 system, the `exit` command is provided.

```
prolog> exit

[Return to top level]
Scheme 84 version 0 (27-Feb-84)

>>>
```

4. Core operations

This section describes the behavior of the core eu-Prolog operations. Core keywords are recognized prior to any attempt at execution of a primitive or user-defined procedure. Thus, no user definition of a procedure with the same name as a core operation will have any effect.

The eight core operations in eu-Prolog are **asserta**, **assertz**, **retract**, **call**, **not**, the cut **!**, **fail**, and **write**.

1. Store side-effects. There are three core operations which provide side-effects to the global store:

(**asserta** *clause*),
(**assertz** *clause*), and
(**retract** *clause*).

For all three, the variables within the clause are resolved in the current environment to yield an instantiated clause. This instantiated clause is the value used in the subsequent action described below.

The store is organized by *assertion names*—the constant that appears in the head of the list structure in the representation of a fully instantiated simple-clause. For each assertion name, a list of associated assertions is maintained in the store. It is this list, in a pretty format, that is printed when the **listing** top-level command is given. The three core operations described in this subsection allow dynamic creation and modification of assertion lists as either top-level calls or as clauses to be invoked within the body of a procedure.

asserta and **assertz** add tautologies to the store; **asserta** places the assertion

(:- *instantiated-clause*)

on the head of the appropriate assertion list, **assertz** places it on the tail. For example,

```
prolog> (:- (demo A B)
            (asserta (A B)))
demo

prolog> (demo foo bar)
(demo foo bar)
; go
fail

prolog> (listing foo)

(foo bar) :-
```

```
prolog> (assertz (foo 1 2 3))
(assertz (foo 1 2 3))
; go

prolog> (listing foo)

(foo bar) :-
(foo 1 2 3) :-

prolog>
```

`retract` examines the appropriate assertion list in order and removes the first assertion whose head unifies with the instantiated clause. If there is no matching assertion, then the `retract` fails.

```
prolog> (retract (foo bar))
(retract (foo bar))
; go
fail

prolog> (listing foo)

(foo 1 2 3) :-

prolog> (retract (foo 1 2 3 4))
fail

prolog> (listing foo)

(foo 1 2 3) :-

prolog>
```

`asserta` and `assertz` can only be instantiated once; if reached a second time via backtracking, they will fail. However, a `retract` can be satisfied more than once; injudicious use of `retract` can do serious damage to the store. Consider the following example.

```
prolog> (listing append)

(append nil A A) :-
(append (A . B) C (A . D)) :-
   (append B C D)

prolog> (retract (append nil X X))
(retract (append nil X X))
; go
fail
```

```
prolog> (listing append)

(append (A . B) C (A . D)) :-
   (append B C D)
prolog> (asserta (append nil A A))
(asserta (append nil A A))
; go

prolog> (listing append)

(append nil A A) :-
(append (A . B) C (A . D)) :-
   (append B C D)
prolog> (retract (append X Y Z))
(retract (append X Y Z))
; go
(retract (append X Y Z))
; go
fail

prolog> (listing append)

[Prolog message---no assertions: append]

prolog>
```

2. Evaluation operators. There are two core operations which are similar to the function `eval` in Lisp systems; these are

`(call` *clause*`)` and

`(not` *clause*`)`.

The first provides a means of determining the value of a data object, that is, a list, as if it were a call to a prolog procedure; the second, a form of negation.

For both forms, the *clause* is instantiated in the environment in existence at the time the invocation of the form occurs. This *instantiated-clause* is then further processed by the system.

In a `call`, the instantiated-clause is treated as if it were any other goal to be instantiated. The interpreter is invoked as if it were typed at top-level or appeared within the body of a procedure. For example,

```
prolog> (call (append X Y (e f)))
(call (append nil (e f) (e f)))
; go
(call (append (e) (f) (e f)))
; go
(call (append (e f) nil (e f)))
; go
fail
```

The principal use of `call` is in providing a way of passing Prolog procedures as arguments to other procedure invocations. Consider the following example of such a procedure; `arith23` is designed to take the name of an arbitrary binary arithmetic operation and compute the result of applying the operation to the numbers 2 and 3.

```
prolog> (:- (arith23 A B)
            (call (A 2 3 B)))
arith23

prolog> (arith23 add R)
(arith23 add 5)
; go
fail

prolog> (arith23 mult R)
(arith23 mult 6)
; go
fail
```

The core operation **not** reverses the roles of successful instantiation and failure. If a call to the instantiated-clause succeeds, the **not** operation fails—triggers backtracking; if the call fails, then the **not** operation succeeds. It is equivalent to the user-defined procedure

```
(:- (=/ A)
    (call A)
    !
    fail)

(:- (=/ A))
```

The cut "**!**" and **fail** operations are described in the next subsection.

not is incapable of adding any new variable associations to the environment. If the attempt to further instantiate the *instantiated-clause* succeeds, then the entire **not** instantiation must fail. In this case, all variable associations are discarded. If the attempt at further instantiation fails, the **not** succeeds but with no new variable associations. Values that cause failure of the **not** operation are not retained in the environment.

Here are some examples of using **not**.

```
prolog> (not (member a (e f g)))
(not (member a (e f g)))
; go
fail

prolog> (not (member a (a b a)))
fail

prolog> (not (not (member b (a b c b))))
(not (not (member b (a b c b))))
; go
fail

prolog> (not (not (member b (a c))))
fail
```

3. Control Operations. eu-Prolog supplies two core commands for control of the backtracking mechanism: the cut "**!**" and **fail**. After discussing each of these, this section presents representative control structures that may be written with the core operators.

The cut is easiest to describe in terms of continuations. Whenever the eu-Prolog interpreter begins searching an assertion list, looking for a head clause which is unifiable with the goal, it records its state. Then, after finding the matched clause and while processing the subclauses in the body, if a cut is encountered, then the entire backtracking control function is replaced with a new function. However this new controlling function is derived from the previously recorded state; it is the continuation of the original goal clause.

`fail` is used to force backtracking. It is identical in effect to typing the `go` command at a semicolon prompt.

The definition of `=/` in the previous subsection used both the cut and `fail`. The cut, in the context of two-element assertion lists, creates a conditional control structure. The schema is

(:- (foo ...) *test* ! *then-part*)
(:- (foo ...) *else-part*)

so that, if the *test* clause succeeds, instantiation of the *then-part* will take place. However, should backtracking be requested out of the *then-part*, the *test* will not be re-done, nor will the second assertion ever be examined. The original call to `foo` fails. Of course, if the *test-part* failed, then the second assertion is used; the *else-part* is run. This is precisely what happens in evaluating calls to `=/`. If the `(call A)` succeeds, the entire `=/` goal fails; otherwise, the entire goal succeeds, trivially.

Here is another example which further illustrates the use of `fail` without the cut. The procedure `retractall` removes all assertions from the store which match the supplied one, instead of only the first, like `retract`.

```
prolog> (listing retractall)

(retractall N) :-
   (retract N)
   fail
(retractall N) :-
```

The `fail` causes the `retract` to be performed repeatedly until it finally fails because there are no more matching assertions to remove. The second assertion in the definition insures that `retractall` always succeeds.

4. Output. There is one core operation included in eu-Prolog for output. It is

(`write` *object*),

where *object* is either a clause or a term. Any variables within the *object* are resolved within the existing environment, and the result is printed on the terminal. The output from a `write` operation has a single asterisk appearing in the left-most column.

For example,

```
prolog> (:- (count 0))
count

prolog> (listing counter)

(counter) :-
   (count N)
   (retract (count N))
   (add 1 N M)
   (write M)
   (asserta (count M))
(counter) :-
   (counter)

prolog> (counter)
*        1
(counter)
; go
*        2
(counter)
; go
*        3
(counter)
; go
*        4
(counter)
; stop
```

5. Primitives

This section describes the six primitive relations **add**, **mult**, **<**, **>**, **atom**, and **is-def** present in the system. Primitives are always recognized prior to any user-defined relation. So, while it is possible to define a procedure with the same name as a primitive, such a definition will have no effect.

What distinguishes the primitives from the core operations? There is no difference in the way each is used. The separation of these forms into two categories was an implementation decision. The cut, **fail**, **call**, and **not** all deal with the control of the eu-Prolog interpreter; because they directly utilize the aruments of the basic Prolog interpreter, they are core operations. **asserta**, **assertz**, and **retract** were placed in the core because they produce side-effects upon the store; **write** because it was an I/O operation. These last four could have been implemented as primitives.

What determined which primitives were included? Some common lisp primitives like **cons**, **car**, **cdr**, **null**, and **equal** have elementary eu-Prolog definitions in terms as user-defined procedures. But, the arithmetic operations and the atom test do not. In addition, an operation which returned the assertion list of a user defined procedure from the store was needed to implement the meta-circular interpreter described in the next section.

(add *A B C***)**

Succeeds if $A + B = C$. If given any two values, **add** will find the third value that satisfies this equation.

(mult *A B C***)**

Succeeds if $A * B = C$. If given any two values, **mult** will find the third value that satisfies this equation.

(< *A B***)**

Succeeds if $A < B$ and both values are known when **<** is called.

(> *A B***)**

Succeeds if $A > B$ and both values are known when **>** is called.

(is-def *Goal Procedure***)**

Succeeds if the assertion list *Procedure* is associated in the store with the assertion name of *Goal*. In practice, *Goal* is usually known, so that calling **is-def** has the effect of associating in the environment the variable *Procedure* with the assertion list that corresponds to *Goal*. The form of the assertion list is the same as that used by the top-level command **edit**. An example of the use of **is-def** is provided in the next section.

The arithmetic primitives behave anomalously; most prolog procedures can be successfully called when any number of the variables have associations in the current environment. The calls to **member** and **append** already presented in this report are examples. However, when the looked-for variable association is a number, eu-Prolog has no means of representing answers from a set of cardinality greater than one. For example,

```
(add A 3 5)
```

succeeds because eu-Prolog can find the one association for `A` that validly instantiates the goal. But

```
(add A B 5)
```

fails because there are several possible associations for the pair of variables `A` and `B`. eu-Prolog cannot represent sets of numerical values.

6. Meta-circular Interpreter

Komorowski provides a meta-circular interpreter for Prolog [Komorowski 1981]. This section presents that interpreter transcribed into eu-Prolog. It utilizes the unification facility of the underlying Prolog system as its own means of unifying clauses.

The top-level procedure is `topexe`.

```
(:- (topexe Goal)
    (is-def Goal Procedure)
    (exe Goal Procedure))
```

The primitive `is-def` is used to retrieve the definition of a eu-Prolog procedure previously defined by the user. `is-def` is careful enough so that the actual definition that is returned contains unique variable names, guaranteed not to occur in any other definition.

The syntax of the returned definition is similar to that used in defining it except that the "`:-`" is not present. Furthermore, it includes all clauses defined with the same procedure name as `Goal`. For example, assume that **append** has been entered in the eu-Prolog store as

```
(:- (append nil A A))

(:- (append (A . B) C (A . B1))
    (append B C B1))
```

Then the call

```
                (is-def (append (a) (b) C) Procedure))
```

returns an environment in which `Procedure` is bound to the list

```
(((append nil G101 G101))
 ((append (G101 . G102) G103 (G101 . G104))
  (append G102 G103 G104)))
```

where the `Gxxx` variables are not otherwise used in either the store or the environment.

After successfully retrieving a procedure definition, `topexe` calls `exe`.

```
(:- (exe Goal Procedure)
    (is-found Goal Procedure Body)
    (exebody Body))
```

`is-found` matches the `Goal` with one of the clauses in the definition `Procedure`, returning with `Body` bound to the body of the matched clause. This relies on the eu-Prolog unification algorithm.

```
(:- (is-found Goal ((Goal . Body) . Rest-of-Procedure) Body))

(:- (is-found Goal (Wrong . Rest-of-Procedure) Body)
    (is-found Goal Rest-of-Procedure Body))
```

exebody separates the first term in the **Body** from the rest of them, recursively calls **topexe** to determine an environment in which the first term is satisfied, and then uses that environment to instantiate the rest of the terms.

```
(:- (exebody nil))

(:- (exebody (First . Rest))
    (topexe First)
    (exebody Rest))
```

It is helpful to remember that a successful Prolog procedure call actually returns two things: an instantiating environment and a continuation which will direct the computation if it is told to look for a different environment. In the implementation of eu-Prolog, is was decided not to print these environments or continuations; the call itself, with all of its variables resolved in the returned environment is printed instead. Furthermore, while the continuation is not seen, it is waiting in the background, ready to be invoked should a failure be detected.

7. Annotated Implementation

This section presents the main sections of the Scheme 84 source code of this implementation.

The unification algorithm and the mechanics of applying a most-general unifier to a clause are well-known and will not be shown. See the book by Nilsson [Nilsson 1980] for a discussion of these matters. In the implementation of eu-Prolog they are written in Franz Lisp. Also in lisp are certain low-level operations for manipulating the store and the environments. The store is implemented using Franz Lisp property lists; environments are association lists.

The code displayed in this section is written in Scheme 84.

The top-level routines with which the user directly interacts are presented first.

```
(define prolog-top-level        ; prolog top-level prompt
   (lambda ()
      (newline)
      (print "prolog> ")
      (prolog-read-eval)))
```

Since there are actually two top-level prompts, "`prolog> `" and the semicolon, it was convenient to separate the prompt-printing from the rest of the read-eval-print loop.

```
(define prolog-read-eval        ; prolog read-eval code
   (lambda ()
      (let ([e (read)])
           (case (type-input e)
                 [goal (evaluate e)]
                 [backtrack (force-backtracking)]
                 [stop (set! force-backtracking
                                  (lambda () (writeln 'fail)))]
                 [exit (return-to-Scheme)]
                 [edit (prolog-edit (body e))]
                 [save (prolog-save (file-name e))]
                 [load (prolog-load (file-name e))]
                 [assert (assert (rest e))]
                 [listing (prolog-pp (body e))]
                 [remove (remove-assertions (body e))]
                 [else (prolog-error e "illegal command")])
           (prolog-top-level))))
```

The original `force-backtracking` function is set to the same value that a `stop` command restores it, a thunk that prints the word "fail".

When the `type-input` function determines that the user's entry is not one of the top-level operations, the `evaluate` function is invoked, beginning the attempt at instantiating a goal. It sets up the appropriate continuations and environment structure and then calls the actual prolog interpreter `prolog-eval`.

```
(define evaluate
   (let ([default-cont (lambda (ignored)
                           (writeln 'fail)
                           (prolog-top-level))])
        (lambda (e)
           (prolog-eval e nil default-cont
              (lambda (result)
                 (if (fail? result)
                     (begin (writeln 'fail)
                            (set! force-backtracking
                               (lambda () (default-cont nil)))
                            (prolog-top-level))
                     (begin (writeln
                              (apply-unifier (env-pt result) e))
                            (set! force-backtracking
                               (lambda ()
                                  ((cont-pt result) nil)))
                            (print "; ")
                            (prolog-read-eval))))
              default-cont))))
```

The arguments to `prolog-eval` are

`e`	the goal clause,
`r`	the current environment,
`back`	the backtracking continuation,
`k`	the result continuation, and
`cutter`	the continuation needed should a cut occur.

The *result continuation* makes up the bulk of the code in `evaluate`; it specifies what the system is to do with a result, whether it is in the form of a failure or a valid instantiation of the goal. If the goal cannot be instantiated, the default continuation is invoked. If a valid instantiation is found, the result continuation displays it and sets the `force-backtracking` function to the value needed to resume this particular computation, should it be requested.

At top-level, both the backtracking and cut continuations are always the default—trigger backtracking if possible, otherwise write out "`fail`".

`prolog-eval` is the main interpreter. It does a dispatch on the type of goal. Core commands are recognized directly; primitives and user-defined procedures require more processing.

```
(define prolog-eval
   (lambda (e r back k cutter)
      (case (call-type e)
            [cut (k (cons r cutter))]
            [fail (k 'fail)]
            [asserta (begin (asserta (apply-unifier r (body e)))
                            (k (cons r back)))]
            [assertz (begin (assertz (apply-unifier r (body e)))
                            (k (cons r back)))]
            [retract
               (k (if (retract (apply-unifier r (body e)))
                      (cons r
                         (lambda (ignored)
                             (prolog-eval e r back k cutter)))
                      'fail))]
            [print
               (begin (writeln "*       "
                               (apply-unifier r (body e)))
                      (k (cons r back)))]
            [call (prolog-eval (apply-unifier r (body e))
                               r back k cutter)]
            [not (prolog-eval (apply-unifier r (body e)) r back
                     (lambda (result)
                        (k (if (fail? result)
                               (cons r back)
                               'fail)))
                     cutter)]
            [clause
              (k (cond [(primitive? e)
                         (eval-prim (apply-unifier r e) r back)]
                        [(user-defined? e)
                         (do-asserts (apply-unifier r e) r
                                     (assert-list e))]
                        [t 'fail]))]
            [else (prolog-error e "illegal clause")])))
```

In all non-error cases, either the atom `fail` or an environment-continuation pair is returned to the result continuation. A non-core goal is recognized as a simple clause, termed a `clause` in the code.

Primitives are handled by `eval-prim`. It does a dispatch on the particular primitive, calling the Franz Lisp function which does the actual work. The result is either the failure atom or a pair made up of the environment obtained while processing the primitive and the unchanged backtracking continuation `back`.

User-defined procedures are evaluated by retrieving the correct assertion list from the store and invoking `do-asserts`. Its arguments are

`e`	the original goal with variables resolved in `r`,
`r`	the environment in existence when the call was made, and
`al`	the assertion list corresponding to the goal.

```
(define do-asserts
   (lambda (e r al)
    (call/cc (lambda (cut)
      (if (null? al)
          (cut 'fail)
          (begin
             (call/cc
                (lambda (back)
                   (let* ([first-assert
                             (unique-idents (first al))]
                          [newrib (unify e (lhs first-assert))])
                         (if (fail? newrib)
                             (back 'fail)
                             (cut (do-body
                                    (rhs first-assert)
                                    (extend-env r newrib)
                                    back cut))))))
             (do-asserts e r (rest al))))))))
```

The `call/cc` expression grabs the continuation that is to become the cut continuation; that continuation is the return to `prolog-eval`. If the assertion list is empty, a failure can be reported immediately. Otherwise, the backtracking continuation is marked and an attempt is made to match the head of the first assertion, extracted by `lhs`, with the goal `e`. Success means that interpretation will continue with a clause-by-clause instantiation of the body of that assertion by the function `do-body`. Failure forces the consideration of the rest of the assertion list.

The arguments to **do-body** are

b	the subgoals, the list of clauses in the body of an assertion,
r	the environment with associations made while unifying the head clause with the goal,
back	the backtrack continuation, into the midst of checking the assertion list, and
cutter	the cut continuation, out of processing the goal altogether.

```
(define do-body
   (lambda (b r back cutter)
      (if (null? b)
          (cons r back)
          (prolog-eval (first b) r back
             (lambda (result)
                (if (fail? result)
                    (back 'fail)
                    (do-body
                       (rest b)
                       (env-pt result)
                       (cont-pt result)
                       cutter)))
             cutter))))
```

If the subgoal list **b** is empty, a result is immediately returned to **do-asserts** and on back to **k** waiting in **prolog-eval**. Otherwise, **prolog-eval** is recursively called on the first subgoal, with the current environment and the appropriate continuations. Failure forces backtracking, and success causes the rest of the subgoal list to be processed.

Bibliography

Clocksin, W. F. and C. S. Mellish. *Programming in Prolog.* Berlin: Springer-Verlag, 1981.

Foderaro, John K., Keith L. Sklower, and Kevin Layer. *The FRANZ LISP Manual.* June 1983.

Friedman, Daniel P., Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand. "Scheme 84 Reference Manual", version 0. Indiana University Computer Science Department Technical Report No. 153. Bloomington, February 1984.

Komorowski, Henryk Jan. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.* Linköping, Sweden: Software Systems Research Center, Linköping University, 1981.

McDermott, Drew. "The Prolog Phenomenon". *SIGART Newsletter* **72**, pages 16–20.

Nilsson, Nils J. *Principles of Artificial Intelligence.* Palo Alto: Tioga Publishing Company, 1980. pages 140–144.

Wallace, Richard S. "An Easy Implementation of PiL (Prolog in Lisp)". *SIGART Newsletter* **85** (July 1983), pages 29–32.

Wand, Mitchell. "A Semantic Algebra for Logic Programming". Indiana University Computer Science Department Technical Report No. 148. Bloomington, August 1983.

Warren David H. D. and Luis M. Pereira. "Prolog—the Language and its Implementation Compared with Lisp". *SIGART Newsletter* **64**, pages 109–115.