NASA Contractor Report 178393

IMPLEMENTING EMBEDDED ARTIFICIAL INTELLIGENCE RULES WITHIN ALGORITHMIC PROGRAMMING LANGUAGES

(NASA-CR-178393) IMPLEMENTING EMBEDDED ARTIFICIAL INTELLIGENCE RULES WITHIN ALGORITHMIC PROGRAMMING LANGUAGES (VAIR) 51 p CSCL 09B

N88-21686

Unclas

G3/61 0140245

Stefan Feyock

VAIR, INC.
Williamsburg, Virginia

Contract NAS1-18002
March 1988

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

IMPLEMENTING EMBEDDED ARTIFICIAL INTELLIGENCE RULES WITHIN ALGORITHMIC PROGRAMMING LANGUAGES

Problem Description

In recent years the powerful techniques offered by Artificial Intelligence (AI) technology have gained acceptance at an ever-increasing rate. On the other hand, most production software systems continue to be written in traditional programming languages which are not oriented toward AI applications (we will refer to such languages as algorithmic languages in the subsequent discussion). Attempts to close the resulting gap have been provisional and system-specific in nature. In particular, the larger commercial AI systems have the capability to interact with programs written in algorithmic languages implemented on the host machine; the non-AI code is typically invoked as subroutine or coroutine in these situations. The approach taken in the initial phase of this project [2] was similar. A Pascal-based Prolog interpreter [5] was modified by adding an escape predicate, a new built-in predicate that allowed information to be passed to/from algorithmic subroutines. While this solution allowed the seamless integration of algorithmic language-based procedures into Prolog (in particular the applications language interface of the RIM database system [1]), it also exposed a basic limitation of this approach. As indicated, the structure of this system required the AI component to operate as the main program. This was appropriate for the intelligent database application in question, since Prolog can be considered to be an ultrasophisticated database query language with deductive capability. It proved to be inappropriate, however, for many of the other algorithmic language-based applications of interest: programs involving optimization, computer-aided design, simulation, graphics, matrix processing, and a multitude of other applications. Many of these programs could make good use of AI capabilities, but are not structured to run in a subprogram mode.

The STRUTEX program [4], a prototype system for the conceptual design of structures to support point loads in two dimensions, provides an illustration as well as test vehicle for these concepts. STRUTEX is structured as a FORTRAN program that accepts load, surface, and support data from the user (provided in part by means of a mouse), and calls AI rules to make decisions regarding the support structure appropriate to that load. Application programs such as STRUTEX illustrate the widespread need for embedded AI, i.e. the integration of AI and algorithmic languages in a fashion that allows the AI facilities to be called as subprograms from the algorithmic program. It is this need that was addressed by the current phase of the project. The results are as follows:

A Prolog-based AI capability callable in embedded mode from algorithmic programs was created

The developed capability was tested in conjunction with the STRUTEX system

Since Phase 1 of this project achieved the embedding of algorithmic subprograms in an AI system, and Phase 2 embedded AI facilities in algorithmic main programs, the result is a product whose two components supplement each other in a highly synergistic fashion. The addition of embedded AI capabilities to algorithmic programs has already been discussed; the augmentation, however, works in the other direction as well. Thus, the invocation of Prolog from algorithmic language allows Prolog to inherit traditional control structures, in which it is greatly deficient, from these languages. As another example, floating point operations, which are missing from this particular Prolog implementation, can be added by invoking algorithmic subprograms from Prolog. Additional augmentations are limited only by the imagination of the programmer.

TECHNICAL DESCRIPTION

The goal of embedding AI facilities in algorithmic languages was achieved in a manner technically similar to that which achieved the integration of algorithmic languages into Prolog: the addition of the new evaluable predicates import and export to Prolog. Before describing these predicates we will briefly review the conceptually similar escape predicate, which is described in detail in [2].

The escape Predicate

The escape predicate is the heart of the Prolog/RIM interface; moreover, we have noted that this predicate can serve as an interface among a variety of other systems. escape would work as well, for example, as a Prolog/graphics package interface, or a LISP/RIM interface, etc. In fact, the only requirement appears to be lists or list-like structures in the calling language (i.e. the language calling the escape), since otherwise the operations needed to set up and decode escape's parameters are too cumbersome. The fact that few languages besides those oriented toward Artificial Intelligence feature list structures as primitives, rather than as a construct to be defined by the programmer, may account for the fact that the escape mechanism is not a universally implemented feature.

In YRIM, the Prolog/RIM integration described in [2], the escape predicate is added to the Prolog side of the interface; it is installed in Prolog as a new evaluable predicate.

Here is the design of the escape predicate as it was implemented:

```
                          escape(X,Y)
                                 ^ ^
                                 | |
           +---------------------+ +----------------+
           |                                        |
      list containing                     result returned
      information on                      in this argument
      operations to
      be performed
```

The input list X is expected to be a linear list of atoms (symbolic or numeric); the result appears bound to Y, and also has the form of a linear list of atoms. Note that quoted strings are legitimate atoms in Prolog, so passing a list

[floatadd, '37.82', '-10.036']

is a feasible method of implementing real addition in Prolog.

The interface between Pascal and Prolog consists of a set of procedures within the Prolog implementation that move the values of the input list elements to a parameter buffer internal to the Pascal program on the Pascal side of the interface, whence they may be manipulated by the Pascal program as desired. Returning parameters to Prolog is the reverse of this process: the result values are placed in the parameter buffer, and interface routines use these values to create a Prolog list and bind it to the second parameter of *escape*. The reader is again referred to the program documentation for details.

The format [<action_code>, <arg>, ---] is typical for input parameter lists, i.e. parameters to be passed to the *escape* predicate in a list bound to the first parameter. This means that the appropriate format for a Pascal program implementing *escape* is a *case* statement on <action_code>; in other words, the Pascal program is typically an interpreter interpreting commands of the form [<action_code>, <arg>, ---].

Invoking Prolog in Embedded Mode

One of the most important reasons embedded AI is a rare phenomenon is that AI facilities are almost universally implemented as subroutine packages written in the major AI languages LISP and Prolog. Since it may be said of both of these languages that the syntax consists entirely of subroutine calls, these AI packages have the appearance of language extensions, or even of new special-purpose languages.

The point of these observations is that embedding LISP- or Prolog-based AI facilities is tantamount to embedding the entire language interpreter and/or run-time environment. These are large stand-alone programs not designed to run in subroutine mode, and thus present formidable problems to the would-be user who intends to invoke them from non-AI programs. We have been able to develop techniques, however, that allow the Prolog interpreter to interact with algorithmic programs in a manner that implements embedded AI. This interaction is the main result of the present research.

Two factors combined to make it possible to embed Prolog in algorithmic languages, one a straightforward separate compilation capability offered by many language systems, the other a brilliant design feature devised by the Prolog implementors.

Interpreters, regardless of the language interpreted, tend to have similar overall structure; in particular, there is almost inevitably a main interpretation loop having the following general form:

```
loop
  perform housekeeping;
  process next language element;
end loop
```

The first factor referenced above is the VMS Pascal [6] module feature. Prefacing a Pascal program with the keyword module rather than program signals the compiler that the program is a separately compiled unit whose internal facilities (data and subroutines) may be made available (by prepending the phrase [global]) to other programs. Such a separate compilation capability, while not a part of standard Pascal, is almost universal in modern Pascal systems running on microcomputers as well as mainframes. We may therefore use it with little concern that portability and general usefulness will be compromised.

Since the Prolog interpreter can trivially be made into a module, and since procedures within it can therefore be made available to calling programs, it is straightforward to insert a procedure like this:

```
[global] procedure test;
         begin
           perform necessary housekeeping;
           perform next interpreter action;
         end;
```

which can then be called by any program that is linked together with the Prolog module. The obvious question is: what is the "next interpreter action"; more particularly, is it what we want done in order to do AI in an embedded mode? As it stands, the answer is "no", since, as indicated above, the next interpreter action is to "process next language element". In Prolog this amounts to prompting the user for the next query, deducing an answer from the rulebase, and printing this answer out for the user.

This interactive mode is inappropriate for embedded applications, where the AI facilities must communicate not with a human user in interactive mode, but with the calling program. It is at this point that the second factor mentioned above comes into play. As it happens, the "next interpreter action" performed in the loop is defined not by a body of Pascal code, but by _Prolog_ statements that are read in by the interpreter upon initialization. These Prolog statements define (are the body of) the Prolog procedure $top; "perform next interpreter action" then amounts merely to causing the invocation of $top. This simplicity of function allows us to reproduce procedure _test_ verbatim:

```
[global]procedure test;
var x:term; e:env;
begin
  choicepoint := 0;           { housekeeping code      }
  NewEnv(e, nil, 0, nil, 0); { more housekeeping code }

  { the following statement invokes Prolog procedure $top: }

  if topA^.proc <> nil then  Goal(MakeFunc(topA, 0, nil, 0);

  KillStacks(0);  { yet more housekeeping }
end;
```

The significance of the fact that the basic interpreter action is defined in terms of Prolog code which is read in at interpreter initialization time is that if we do not like what the interpreter does, we need not reprogram long sections of obscure Pascal code; changing the Prolog statements defining $top is all that is required. This, however, is quite easy to do, since Prolog is a high-level language. To transmit a feel for what is involved, we present part of the original definition of $top.

```
'$top' :- write('?- '), read(X), nonvar(X), '$exec'(X).


'$exec'(end) :- !, end, nl.
'$exec'((?- end)) :- !, end, nl.
'$exec'(G) :- '$gnd'(G), !, G.
'$exec'(G) :- G, write('==> '), write(G), write(' ? '), '$ask'.
```

As can be seen, $top writes out the prompt '?-', reads the user's input, makes sure that this input is not solely a variable, and executes it by invoking $exec on it. The definition of $exec, in turn, follow immediately. The first two clauses simply cause termination if the user types "end" or "?-end". The last two clauses of the definition deal with the cases where G does, respectively does not, contain variables; in either case, G is invoked. When no additional answers for G exist, $exec completes, causing $top to complete as well and return to the interpreter loop.

For the purposes of implementing embedded Prolog it was necessary to change the above definition of $top so that it accepted data from the calling program rather than the user, processed it as desired, and passed the results back to the calling program, rather than printing them out at the terminal by means of a write(G). Here is the modified version of $top:

```
'$top' :-  import(X), '$process'(X).
```

The initial $ sign, inciaentally, is a naming convention designating the procedure name as part of the interpreter loop definition; adherence is optional. The ' marks surrounding such names are needed to let Prolog accept "strange" characters such as $ without complaint.

As will be seen in the course of the subsequent discussion, the procedures import(X) and export(X) transfer data from, respectively to, outside programs written in algorithmic languages. The data in question is bound to variable X; procedure $process(X) processes it.

The elegance and simplicity of this method of defining the interpreter loop is apparent. What is even more impressive is the flexibility this approach yields: the code defining the action of the interpreter is available to the Prolog programmer for modification. The power of this particular modification which we have undertaken becomes apparent when it is noted that the definition of $process is to be supplied by the <u>user</u>, and may do anything at all that the user desires. As a simple test case, the following rule definition was used:

```
'$process'(X) :- write(' imported/exported '),
                 write(X), export(X).
```

The data imported into Prolog is written on the terminal, whereupon <u>export</u> returns it unchanged to the calling program.

The import and export Predicates

The escape predicate described above transfers information to a non-Prolog program, which acts on it, whereupon the results are transferred back into the Prolog program. For the purposes of this work it has proved useful to break out the primitive components of the transfers involved. As indicated, import(X) and export(X) are new evaluable (built-in) predicates that have been added to Prolog to achieve the goals of this project. import is used to make data created externally (say by an algorithmic program) available to Prolog; export passes data back to the "outside". In both cases the data involved is bound to the parameter of the predicate. Since they are central to the results that have been achieved, we will describe the structure and use of these predicates in detail.

The communications interface between Prolog and the "outside world" that was devised to implement these predicates is a buffer structure that is shared by the programs that need to exchange information. In the (typical) case of the STRUTEX system a FORTRAN program is communicating with the (Pascal-based) Prolog interpreter; we will give the buffer declarations on both sides of the interface. The Pascal declarations are:

```
   arg_i: [COMMON(FPCOMI)] array[1..maxargs] of integer;
   arg_r: [COMMON(FPCOMR)] array[1..maxargs] of real;
   arg_s: [COMMON(FPCOMS)] array[1..maxargs] of alpha;

   arg_type: [COMMON(FPCOM2)] array[1..maxargs] of char;
```

As can be seen, the buffer structure consists of four parallel arrays. Array arg_type[i] contains a one-character flag indicating whether the i'th data element is of type integer (flagged by 'i'), real ('r'), or string ('s'), i.e. packed array[1..alphasize] of char. If the element is an integer, it is contained in arg_i[i]; if real, in arg_r[i], and if string, in arg_s[i]. In the Prolog interface reals are actually passed in arg_s as strings, due to quirks of this particular Prolog implementation. Array arg_r is thus not used in STRUTEX, but has been retained for the sake of generality.

This storage scheme optimizes simplicity and portability at the expense of space: to add an unforeseen data type, we need simply add the declaration

```
  arg_u: [COMMON(FPCOMU)] array[1..maxargs] of unforeseen_type;
```

and decide on a character flag to denote it. Since the number of data elements to be passed will generally be moderate (maxargs is currently set to 10), allocating unused space is well worth the savings in complexity that result over a scheme using data overlays produced by EQUIVALENCEing. The phrases [COMMON(FPCOM*)] in the above declarations indicate to the compiler that the storage to be allocated to these data structures is to be a COMMON area that will be shared by other programs; FPCOM* names the COMMON area in which this data structure is to be placed. The FORTRAN side of the interface looks like this for integer data:

```
    INTEGER intval(maxargs)
    CHARACTER*1 argtype(maxargs)
    COMMON /FPCOM2/ argtype
    COMMON /FPCOMI/ intval
```

and analogously for the real and string buffers.

Information Transfer

We will now describe how information flows into and out of these buffers on both sides of the interface. The interface operates as follows:
when a FORTRAN program wishes to invoke embedded Prolog, it places the information to be passed to Prolog in the buffer(s) of the corresponding type, with the appropriate flag in the flag buffer. Subroutines to perform this placement in a uniform and modular manner are provided, and will be discussed below. Once the data to be transferred has been placed, the subroutine call

```
CALL TEST
```

invokes the (global) procedure test within the Prolog interpreter, thus invoking $top, as discussed above. On the Prolog side, a call to import will retrieve the data stored in the shared buffer structure, bind it to the parameter of import, and make it available to the Prolog rules. If there is data to be passed back, procedure export places it in the buffer structure on the Prolog side.

Here is a listing of subroutine pushstr, which is used by the FORTRAN programmer to place string data in the buffer structure for transmittal to Prolog:

```
SUBROUTINE pushstr(sarg)
implicit none
integer alfalength, maxargs
PARAMETER (alfalength = 8, maxargs = 25)

character*(*) sarg
character*(alfalength) strng
INTEGER no_of_args
character*1 argtype(maxargs)
common /fpcom2/ argtype
common no_of_args
character*(alfalength) strval(maxargs)
common /fpcoms/ strval

strng = sarg
no_of_args = no_of_args + 1
strval(no_of_args) = strng
argtype(no_of_args) = 's'
RETURN
END
```

As can be seen, this routine places its argument in the appropriate buffer array, sets the type flag to 's', and updates no_of_args, the number of arguments inserted so far. To transmit the string 'Hello', for example, the programmer would write

```
CALL PUSHSTR('Hello')
```

The routines for inserting integer and real arguments into the buffer structure are analogous. Here is a complete sequence corresponding to a typical parameter setup:

```
    NO_OF_ARGS = 0
    CALL PUSHSTR('color')
    CALL PUSHSTR('red')
    CALL PUSHSTR('volume')
    CALL PUSHREAL('16.47')
    CALL PUSHSTR('amount')
    CALL PUSHSTR(100)
    CALL TEST
```

What happens to these parameters on the Prolog side depends on the particular rules which the user has provided as definition of $process.

As can be seen, the interface is rather straightforward on the FORTRAN side, the perhaps most unaesthetic element being the requirement to initialize NO_OF_ARGS to 0. Means of obviating this requirement exist and were considered, but the cure proved worse than the disease in every case.

The Prolog Side of the Interface

From the programmer's point of view, the Prolog side of the interface is irreducibly simple. Suppose the above sequence of calls has been made; the call to TEST then causes $top to be activated, which in turn causes $process to execute, which does whatever the (Prolog) programmer has programmed. If a Prolog rule needs access to the parameters, an invocation of import(X) does it: after completing, the parameter X will be bound to the list

[color, red, volume, '16.47', amount, 100] which can then be used by the Prolog program as needed.

The implementation of import and export is easily described. Two procedures, Doimport and Doexport, were written to act as handlers for these constructs. As indicated above, Doimport collects the data from the buffer structure (and counts the elements transmitted), converts them into Prolog atoms, collects these atoms into a Prolog list, and finally binds this list to the argument of import. Doexport does the inverse: its argument must be bound to a list of Prolog atoms. These atoms are pulled off the list one by one. Their data type is determined, they are converted to the corresponding buffer structure type (integer, real or string), and inserted in the buffer structure.

Calling Program Control of Embedded Prolog

We have described how information can be passed from FORTRAN to embedded Prolog and accessed by the invoked Prolog rules. The nature of Prolog, however, makes it easy for the calling program to exert considerable control over the processing performed on the Prolog side. If the Prolog rules are set up correctly, any desired Prolog procedure to be invoked can be specified from the FORTRAN side. In fact, since Prolog can interpret the passed data, a virtual interface of any desired design can easily be created. The one we have designed is simple and powerful, but we emphasize that it is only one of an infinite number of possibilities.

Our interface design is based on the observation that there are two basic operations that can be performed in Prolog: invocation of a Prolog procedure, and updates of the Prolog database. It can be maintained that the database updates are themselves merely procedure calls to the <u>assert</u> and <u>retract</u> procedures. This is correct, but updates are conceptually sufficiently distinct to deserve their own classification. Our $process procedure therefore expects the data being passed to it to be in one of two possible list formats:

[assert, <predicate>, <arguments>]

and

[call, <function>,<arguments>]

Thus, suppose the list passed from FORTRAN to Prolog is

[assert, p, a, b, c]

Then the Prolog procedure call

assert(p(a,b,c))

is executed. Similarly, passing the list

[call, f, x, y, z] causes call(f(a,b,c)) to be executed,

invoking f(a,b,c) as Prolog procedure.

Here are the Prolog statements that create this interface:

```
'$process'(X) :- X = [assert | Y],!, F =.. Y, assert(F).
/* e.g. if X = [assert, f, a, b, c],
   an assert(f(a,b,c)) is executed */

'$process'(X) :- X = [call | Y],!, F =.. Y, call(F).
/* e.g. if X = [call, f, a], a call(f(a))
   is executed */

'$process'(X) :- write(' imported/exported '),
                 write(X), nl, export(X), nl.
/* this last definition can be expanded
   to do whatever is desired with X */
```

A Case Study: STRUTEX

The embedded AI facilities we have developed are being tested and applied in STRUTEX, a prototype knowledge-based system for the conceptual design of structures to support point loads in two dimensions.

As presently constituted, STRUTEX combines a database, a knowledge base, and a graphics display into a prototype knowledge-based system. The program simulates an engineer, beginning work on a new project with a blank piece of paper, and a discussion with his manager. The graphics screen plays the part of the blank piece of paper, with a text area for dialogue between the manager and engineer.

The user inputs data about the load, such as number of loads, type of load (e.g. gravity load), the load magnitude, and similar information. A mouse is used to position the load on the screen. The user then inputs data about the support surface, such as position with respect to load, whether or not it is a point surface, and the area of a non-point surface. The mouse is again used to display the midpoint of the support surface, and the program calculates the length of the surface and the distance from the surface to the load point(s). Finally the user specifies whether or not the support must be lightweight. All of this data is stored in the database (RIM).

The knowledge base is then executed to determine the type of support (e.g. beam or truss) that is required. This determination is based on knowledge about the relationship between the support surface and the load and data in the database. Here is a Prolog rule typical of those called in embedded mode by the FORTRAN-based STRUTEX program:

```
beam :- surflc(below),surfa(large), not(suppwt(light)).

/*
   a beam support is appropriate if the support surface
   location is below the load, the surface area is large,
   and the support is not known to be lightweight
*/
```

The program computes the coordinates of the members of the support, which are also entered into the database. If there is a single load point and the support type is a truss, then a determination is made of whether or not bracing is needed by checking the ratios of the member lengths against the loading conditions. If there are multiple load points and the support type is a truss, then the user designs an initial truss guided by recommendations from the knowledge base. Features of the design are checked against the knowledge base and recommendations for improvements are made. These iterations continue until the user is satisfied with the design. Each new support is displayed on the graphics screen.

The FORTRAN/Prolog Interface

We will now examine the interface used to call the embedded rule base from FORTRAN. The FORTRAN main program component of STRUTEX is structured so that requirements for services such as graphics support, RIM database accesses, or calls to embedded AI facilities, are satisfied by CALLs to handler subroutines. These handlers have the logical structure of <u>case</u> statements (although FORTRAN must, of course, simulate this effect by means of IFs or computed GOTOs); thus invocations of these handlers have as parameters a numeric code indicating the particular service required, plus the specific information required to perform that service. The name of the handler for the embedded knowledge base is KBXEC; a listing of KBXEC may be found in Appendix 1.

The following FORTRAN statements define the interface among STRUTEX , the graphics handler, and the RIM database handler:

ORIGINAL PAGE IS
OF POOR QUALITY

```
      IMPLICIT REAL*8 (A-H, O-Z)
      CHARACTER*8 PLOADT,SURFLC,SUPTYP,SUPPWT
      CHARACTER*8 SURFT,CHOICE,BRCTYP,CHBRAC,SIDES
      CHARACTER*10 TEMP
      CHARACTER*80 STRING
      COMMON/LOADC/PLOADN,PLOADT,PLOADX,PLOADY,HLOAD,VLOAD,DIST
      COMMON/SURFC/SURFLC,SURFXS,SURFYS,SURFXE,SURFYE,SURFA,
     1            SURFXM,SURFYM
      COMMON/SUPPC/SUPPNO,SUPTYP,SUPPWT,SUPPXS,SUPPYS,SUPPXE,
     1            SUPPYE,SUPDIS
      COMMON/SHRCOM/NPTS,NTOTSP,PIXPER,XSECT,YSECT,SURFT,
     1    RLOAD,RSRFAC,RSUPRT,RATIO,CHBRAC,BRCTYP,SIDES,SIDDIF
      COMMON/MEMXY/SMEMNO(100),XS(100),XE(100),YS(100),YE(100)
      DIMENSION ARLOAD(7),ARSURF(8),ARSUPP(8)
      EQUIVALENCE (ARLOAD(1),PLOADN),(ARSURF(1),SURFLC),
     1           (ARSUPP(1),SUPPNO)
```

The subsequent statements:

```
       integer alfalength, maxargs
       PARAMETER (alfalength = 8, maxargs = 10)

       CHARACTER*(alfalength) strval(maxargs)
       character*1 argtype(maxargs)

       integer no_of_args   ! for sharing with the
       common no_of_args    ! stacking routines only

       common /fpcoms/ strval
       COMMON /fpcom2/ argtype
```

define the FORTRAN/Prolog communications interface, which has been described previously. We will describe the action of KBXEC for a typical invocation of the handler:

```
C USE KNOWLEDGE BASE TO DETERMINE HOW DIAGONALS
C ARE TO BE DRAWN BETWEEN MEMBERS OF A TRUSS
C BY CHECKING LENGTH OF TWO ADJACENT SIDE MEMBERS

         CALL KBXEC(2,HDIST,TDIST,ALPHA)
```

The section of KBXEC code executed as a result of this call is:

ORIGINAL PAGE IS
OF POOR QUALITY

```
C
C   DETERMINE HOW DIAGONALS ARE TO BE DRAWN
C   BETWEEN MEMBERS OF A TRUSS
C
      IF(IOPT.EQ.2) THEN
       no_of_args = 0
       call pushstr('assert')
       call pushstr('dist1')
       call pushreal(tdist)
       call test

       no_of_args = 0
       call pushstr('assert')
       call pushstr('dist2')
       call pushreal(hdist)
       call test

       no_of_args = 0
       call pushstr('call')
       call pushstr('cmpsides') ! activate compare_sides rule in Prolog
       call test

       call cc('u',strval(1),SIDES)
       read(strval(2),'(F8.2)')SIDDIF
      ENDIF
```

The code segment

```
       call pushstr('assert')
       call pushstr('dist1')
       call pushreal(tdist)
```

causes the character strings "assert" and "dist1", as well as the real number tdist, to be inserted into the interface buffer. The subsequent line:

call test

invokes the Prolog routine <u>test</u>, which, as indicated earlier, simply activates the Prolog interpreter on the goal (Prolog predicate call) $top. Recall that $top is defined as

'$top' :- import(X), '$process'(X).

Suppose, for example, that the value of tdist (which was passed to KBXEC as floating-point parameter) was 3.5. The import predicate assembles the arguments passed in the interface buffer into a Prolog list:

```
[assert, dist1, '3.5']
```

and binds it to X. (Note that the real number 3.5 has been automatically converted to a Prolog string. The reason for this will be set forth in the subsequent discussion of real arithmetic operations in Prolog.) Finally, $process is activated with this value of X as argument.

As discussed above, the action of $process when encountering a list beginning with the atom "assert" is to invoke the call

```
assert(dist1('3.5'))
```

which inserts the predicate dist1('3.5') into the Prolog database.

The subsequent code sequence similarly causes

```
dist2(<value of hdist>)
```

to be inserted. Finally, the sequence

```
 call   pushstr('call')
 call  pushstr('cmpsides')  !   activate  compare_sides rule in Prolog
 call test
```

causes execution of the Prolog procedure call(cmpsides), defined as follows:

```
 /* Rule COMPARE_SIDES; IOPT = 2 */
cmpsides :- dist1(D1), dist2(D2),!,
            retract(dist1(D1)), retract(dist2(D2)),
            fminus(D1, D2, Siddif), fabs(Siddif, Diffa),
            fdiv(Diffa, D1, Pcdif1), fdiv(Diffa, D2, Pcdif2),
            csstuff(Pcdif1, Pcdif2).
```

As is evident, this rule looks up the values of dist1 and dist2 in the Prolog database, binds the results to D1 respectively D2, and deletes the current dist1 and dist2 entries from the database. The procedure _csstuff_ is then called with arguments ¦D1 - D2¦/D1 and ¦D1 - D2¦/D2. Note that since this particular Prolog implementation lacks floating-point arithmetic, such operations must be performed by calls to procedures such as fminus, which are defined in terms of the escape predicate, which in turn invokes FORTRAN code. We thus have FORTRAN invoking embedded AI rules, which in turn can invoke FORTRAN code; such invocations can chain indefinitely.

The _csstuff_ procedure is defined as

```
csstuff(X, Y) :- (fgt(X, '0.1') ; fgt(Y, '0.1')),
                 export([notequal,Siddif]).

csstuff(X, Y) :- export([equal,Siddif]).
```

The first rule for _csstuff_ stipulates that if X > 0.1 or Y > 0.1, then the character string 'notequal' and the numeric value of Siddif are to be inserted into the interface buffer; otherwise, the string 'equal' and Siddif are inserted.

With completion of procedure _csstuff_, procedures cmpsides, $process, and $top complete as well. With the completion of $top, control is returned to the FORTRAN calling program. In this case, the code executed immediately after returning is

```
call cc('u',strval(1),SIDES)
read(strval(2),'(F8.2)')SIDDIF
```

Recall that the array strval is the one of the three parallel interface buffer arrays in which <u>string</u> values are returned from Prolog. The FORTRAN procedure <u>cc</u> converts from upper to lower case letters or back; in this case the string in strval(1) (which was 'equal' or 'notequal') is converted to capitals and the result stored in FORTRAN variable SIDES. cc is needed because names with initial capitals designate variables in Prolog; names beginning with lower-case letters denote constants. Similarly, the real number value (returned in <u>string</u> form) of Siddif is converted to floating point representation via an internal read, and the result stored in FORTRAN variable SIDDIF. This completes processing of option 2 on part of KBXEC, and control returns to the caller.

Implementation of Floating Point Operations

Since the University of York Prolog interpreter [5] emphasizes simplicity, floating-point operations are not implemented. The STRUTEX operation, however, requires such operations at every turn. The ease with which floating-point operations were added to Prolog is indicative of the flexibility and simplicity of the interface that has been constructed.

Here are the Prolog rules defining floating-point operations:

```
flt(F1,F2)   :- escape([1,F1,F2],[lt]).
fle(F1,F2)   :- escape([1,F1,F2],[le]).
feq(F1,F2)   :- escape([1,F1,F2],[eq]).
fge(F1,F2)   :- escape([1,F1,F2],[ge]).
fgt(F1,F2)   :- escape([1,F1,F2],[gt]).
fplus(F1, F2, R) :-  escape([2,F1,F2],[R]).
fminus(F1, F2,R) :- escape([3,F1,F2],[R]).
ftimes(F1, F2,R) :- escape([4,F1,F2],[R]).
fdiv(F1,F2 ,R) :-   escape([5,F1,F2],[R]).
fabs(F,R)  :-   escape([6,F],[R]).
```

As is evident, each of these operations invokes the <u>escape</u> predicate. Appendix 3 reproduces the subroutine IFACE, which implements the <u>case</u> statement which is invoked by escape. To illustrate its operation, we will consider the will consider the rule for floating less-than:

```
flt(F1,F2)  :- escape([1,F1,F2],[lt]).
```

A typical call to the procedure appears thus:

```
flt('3.29', '-2.6')
```

Recall that floating-point numbers are represented in string format. This call invokes

```
escape([1,'3.29', '-2.6'],[lt])
```

which causes the arguments 1, '3.29', and '-2.6' to be placed in the interface buffer as usual. As is generally the case, the first argument (the "1") is a command code; the following line of IFACE <u>cases</u> on this code:

```
goto (100,200,300,400,500,600),intval(1)
```

Recall that intval is the part of the interface buffer that holds integer arguments. Since intval(1) contains the 1 that was transmitted, control is transferred to statement 100 in IFACE. The statements

```
100       read(strval(2),'(F8.2)')r1
          read(strval(3),'(F8.2)')r2
```

transform the real values, which are in the string representation required by Prolog, to floating-point representation, and store them in variables r1 and r2. The subsequent statements test the relationship between these values:

```
      IF (r1 .gt. r2) THEN
        strval(1) = 'gt'
      ELSE IF (r1 .eq. r2) THEN
        strval(1) = 'eq'
      ELSE IF (r1 .lt. r2) THEN
         strval(1) = 'lt'
      ELSE IF (r1 .le. r2) THEN
         strval(1) = 'le'
      ELSE IF (r1 .ge. r2) THEN
        strval(1) = 'ge'
      else
        print *, ' *** COMMAND CODE 2: WEIRD ARGS, NOT ORDERED'
      END IF
      no_of_args = 1
      argtype(1) = 's'
      goto 3000
```

Since r1 = 3.29 and r2 = -2.6, it is evident that 'gt' will be stored in strval(1). This string is returned to Prolog and made into a list, [gt], which becomes the second (output) argument of escape. Since, however, this invocation of escape had [lt] as second argument, and [lt] does not match [gt], the invocation fails. This is, of course, the desired result, since 3.29 is not less than -2.6.

An obvious question that might arise on examination of the floating-point comparisons is why all of them were assigned the same action code, i.e. 1. The answer is that this was not a compelled choice; choosing a separate action code for each comparison is a feasible alternative. Design of the appropriate IFACE FORTRAN code is left as an exercise for the interested reader; it is our opinion that the given design results in somewhat cleaner code.

Operations such as flt(F1,F2) are predicates that operate by testing their operands and succeeding or failing, depending on the outcome. Operations such as fplus (floating-point plus), however, must produce results. The natural way to implement such operations is as functions. Prolog syntax, however, does not allow for functions: all procedures are subroutines. Values must therefore be returned bound to an output parameter rather than to the function name. Thus, to add 1.0 and 1.0, and print out the result, we would write

```
fplus('1.0','1.0', X), write(X).
```

causing a '2.0' to be written out. The principle of operation of the definition of fplus in terms of an escape predicate is similar to that of flt; Appendix 3 provides details.

We have presented a complete dissection of a typical invocation of embedded AI rules from a FORTRAN program, and demonstrated how these rules could invoke FORTRAN code in turn. Processing for the other options is analogous. As can be seen, the calling and return sequences are stereotyped and rather straightforward; programming with embedded AI rules expressed in Prolog thus becomes sufficiently straightforward to serve as a standard programming technique for algorithmic applications.

Power of Embedded Prolog

The STRUTEX rules reproduced in Appendix 2 correspond in their effects to the CLIPS [3] rules used by the STRUTEX version described in [4]. It is natural to pose questions regarding the relative and absolute power of Prolog rules.

Strictly speaking, CLIPS and Prolog are equivalent, since both systems can implement a Turing machine. From the programmer's point of view, however, it is fair to say that Prolog is significantly more powerful than CLIPS. Most of the features of CLIPS, such as the built-in rule base, are present, or at least can be easily simulated, in Prolog. In addition, Prolog has a powerful deductive capability based on resolution. This capability is central to the capabilities of Prolog, and is not matched by any feature of CLIPS.

Prolog is, of course, an extremely powerful stand-alone programming language in its own right. Its capabilities are sufficiently impressive to have caused it to be chosen as the language of Japan's fifth-generation project, as well as being the dominant AI language in Europe. It suffers, however, from severe deficiencies in the area of control structures, since all control flow in Prolog is based on backtracking rule application. While this is natural for certain applications, it can become an extremely unnatural way to program in situations requiring more traditional control structures such as <u>while</u> and <u>do</u> loops.

One of the most significant results of the present research is that it imposes the control structures provided by the traditional calling language on Prolog. As is clear from the calls to embedded rules we have examined, such invocations can be enclosed within loops, if statements, or whatever other construct the calling language offers. Programming in Prolog is thus brought, perhaps for the first time, into the realm of general-purpose algorithmic programming.

CONCLUSION

A method for embedding Artificial Intelligence capabilities based on Prolog rules has been reported. The techniques developed were applied to the STRUTEX program, a prototype system for the conceptual design of structures to support point loads in two dimensions. The Prolog-based rules proved to be more expressive and powerful than the original CLIPS version; moreover, needed features such as real arithmetic were easily supplied by means developed in the initial phase of this project. The approach developed should be applicable to a wide variety of algorithmic languages, since our implementation presupposes only the existence of a straightforward separate compilation capability, as supplied by the algorithmic language processing systems of most modern machines.

At least as significant a result is the imposition of control structures provided by the algorithmic calling language on Prolog. This superposition eliminates much of the difficulty which Prolog programming poses, thus making this powerful AI tool available to the algorithmic programmer.

REFERENCES

1. BCS RIM Version 6 User Guide, Boeing Commercial Aircraft Company: Central Scientific Computing Complex Document Z-3, NASA/Langley Research Center, May 1985

2. Feyock, S., Implementation of Artificial Intelligence Rules in a Data Base Management System, NASA Contractor Report 178048, VAIR, INC., February 1986.

3. Riley, G., C. Culbert and R. Savely, "CLIPS: an Expert System Tool for Delivery and Training", Proceedings of the Third Conference on AI for Space Applications, November 1987.

4. Rogers, J., S. Feyock and J. Sobieski, STRUTEX: A Prototype Knowledge-Based System for Initially Configuring a Structure to Support Point Loads in Two Dimensions, submitted to AIEE 3, Los Angeles.

5. Spivey, J., Portable Prolog User's Guide, Dept. of Computer Science, University of York, Heslington, York, England, October 1983.

6. "Programming in VAX FORTRAN", Document AA-D034D-TE, Software Version V4.0, Digital Equipment Corporation, Maynard, MA, September 1984.

Appendix 1

STRUTEX Rules

```
'$process'(X) :- X = [assert | Y],!, F =.. Y, assert(F).
/* e.g. if X = [assert, f, a, b, c],
   an assert(f(a,b,c)) is executed */

'$process'(X) :- X = [call | Y],!, F =.. Y, call(F).
/* e.g. if X = [call,  f, a, b, c],
   a call(f(a,b,c)) is executed    */

'$process'(X) :- write(' imported/exported '),
                 write(X), nl, export(X), nl.
/* this last definition can be expanded
   to do whatever is desired with X */

flt(F1,F2) :- escape([1,F1,F2],[lt]).

fle(F1,F2) :- escape([1,F1,F2],[le]).

feq(F1,F2) :- escape([1,F1,F2],[eq]).

fge(F1,F2) :- escape([1,F1,F2],[ge]).

fgt(F1,F2) :- escape([1,F1,F2],[gt]).

fplus(F1,F2,R) :-  escape([2,F1,F2],[R]).

fminus(F1,F2,R) :- escape([3,F1,F2],[R]).

ftimes(F1,F2,R) :- escape([4,F1,F2],[R]).

fdiv(F1,F2,R) :-   escape([5,F1,F2],[R]).

fabs(F,R) :-   escape([6,F],[R]).


/***********************************/
/* application program starts here */
/***********************************/


/*rule BEAM; IOPT = 1 */

support :- beam,!, assert(support(beam)),  export([beam]).

support :- truss,!, assert(support(truss)),  export([truss]).
support :- string,!, assert(support(string)),  export([string]).

beam :- surflc(side), surfa(point).

beam :- surflc(side), surfa(large), not(suppwt(light)).

beam :- surflc(below), surfa(point).

beam :- surflc(below), surfa(large), not(suppwt(light)).

beam :- surflc(above), surfa(point), not(ploadt(gl)),
        not(suppwt(light)).


/* Rule TRUSS */

truss :- (surflc(side) ; surflc(below)),
          surfa(large), suppwt(light).

truss :- surflc(above), surfa(large),
          not(ploadt(gl)), suppwt(light).


/* Rule STRING */

string :- surflc(above), ploadt(gl).


/* Rule BRACE_TYPE; IOPT = 4 */

brcetype :- alpha(Alphaval),!,dobracetype(Alphaval).

dobracetype(Alphaval) :- flt(Alphaval,'40.0'),!,
                         assert(typeofbrace(v)), export([v]).

dobracetype(Alphaval) :- assert(typeofbrace(z)), export([z]).


/* Rule COMPARE_SIDES; IOPT = 2 */

cmpsides :- dist1(D1), dist2(D2),!,
            retract(dist1(D1)), retract(dist2(D2)),
            fminus(D1, D2, Siddif), fabs(Siddif, Diffa),
            fdiv(Diffa, D1, Pcdif1), fdiv(Diffa, D2, Pcdif2),
            csstuff(Pcdif1, Pcdif2, Siddif).

csstuff(X, Y, Siddif) :- (fgt(X, '0.1') ; fgt(Y, '0.1')),
                         export([notequal,Siddif]).

csstuff(X, Y, Siddif) :- export([equal,Siddif]).
```

```
/* Rule BRACE_CORRECT for triangles; IOPT = 33 */

brcorrtr :- triok(Alpha),!, retract(triok(Alpha)),
            triokstuff(Alpha).

triokstuff(A) :- flt(A, '15.0'), export([small,'0.0']).

triokstuff(A) :- fgt(A, '120.0'), export([large,'0.0']).

triokstuff(A) :- export([good,'0.0']).


/* Rule BRACE_CORRECT; IOPT = 3 */

brcorrqd :- quadok(Alpha),!, retract(quadok(Alpha)),
            qokstuff(Alpha).

qokstuff(A) :- flt(A, '15.0'), export([small,'0.0']).

qokstuff(A) :- fgt(A, '75.0'), export([large,'0.0']).

qokstuff(A) :- export([good,'0.0']).


/* Rule BRACING; IOPT = 5 */

bracing :- xn1(N1), dist(D), toleranc(Tol),!,
           fdiv(N1,D,Temp), fdiv(Temp,Tol,R), fabs(R,Ratio),
           (fgt(Ratio, '1.0') -> Brace = yes ; Brace = no),
           assert(ratio(Ratio)), assert(brace(Brace)),
           export([Brace, Ratio]).


/* Rule EXPLANATION; IOPT = 8 */

explain :- support(Supp),nl,
           write(' ########################################'),
           nl,nl, write(' A '), write(Supp),
           write(' is the choice for a support.'), nl, nl,
           write(' ########################################'),
           nl,nl, write(' Reasons: '), nl,!, reasons, fail.

reasons :- surflc(side),
 write(' The support surface is to the side of the loads.'),nl.

reasons :- surflc(below),
 write(' The support surface is below the loads.'),nl.

reasons :- surflc(above),
 write(' The support surface is above the loads.'),nl.
```

```
reasons :- surfa(large),
 write(' The support surface is not a point.'),nl.

reasons :- suppwt(X),
 (X = light ->
    write(' The support surface must be lightweight.')
    ; write(' The support can be heavy.')), nl.

reasons :- ploadt(vl),
 write(' There are only vertical loads.'),nl.

reasons :- ploadt(gl),
 write(' There are only gravity loads.'),nl.

reasons :- ploadt(sl),
 write(' There are only sideways loads.'),nl.

reasons :- ploadt(gs),
 write(' There is a combination of gravity and sideways loads.'),
 nl.

reasons :- ploadt(vs),
 write(' There is a combination of vertical and sideways loads.'),
 nl.
```

Appendix 2

Embedded AI Calling Routine

```
      SUBROUTINE KBXEC(IOPT,HDIST,TDIST,ALPHA)
C
C  THIS SUBROUTINE INTERFACES WITH THE KNOWLEDGE BASE
C  STRINGS ARE ASSERTED AND CLIPS IS EXECUTED
C
      IMPLICIT REAL*8 (A-H, O-Z)
      CHARACTER*8 PLOADT,SURFLC,SUPTYP,SUPPWT
      CHARACTER*8 SURFT,CHOICE,BRCTYP,CHBRAC,SIDES
      CHARACTER*10 TEMP
      CHARACTER*80 STRING
      COMMON/LOADC/PLOADN,PLOADT,PLOADX,PLOADY,HLOAD,VLOAD,DIST
      COMMON/SURFC/SURFLC,SURFXS,SURFYS,SURFXE,SURFYE,SURFA,
     1             SURFXM,SURFYM
      COMMON/SUPPC/SUPPNO,SUPTYP,SUPPWT,SUPPXS,SUPPYS,SUPPXE,
     1             SUPPYE,SUPDIS
      COMMON/SHRCOM/NPTS,NTOTSP,PIXPER,XSECT,YSECT,SURFT,
     1    RLOAD,RSRFAC,RSUPRT,RATIO,CHBRAC,BRCTYP,SIDES,SIDDIF
      COMMON/MEMXY/SMEMNO(100),XS(100),XE(100),YS(100),YE(100)
      DIMENSION ARLOAD(7),ARSURF(8),ARSUPP(8)
      EQUIVALENCE (ARLOAD(1),PLOADN),(ARSURF(1),SURFLC),
     1            (ARSUPP(1),SUPPNO)

       integer alfalength, maxargs
       PARAMETER (alfalength = 8, maxargs = 10)

       CHARACTER*(alfalength) strval(maxargs)
       character*1 argtype(maxargs)

       integer no_of_args  ! for sharing with the
       common no_of_args   ! stacking routines only

       common /fpcoms/ strval
       COMMON /fpcom2/ argtype

C
C  INITIALIZE THE KNOWLEDGE BASE AND LOAD THE RULES
C
      IF(IOPT.EQ.0) THEN
        no_of_args = 0
       do i = 1, maxargs
        argtype(i) = ' '
       end do
      END IF
```

```
C
C   DETERMINE THE TYPE OF SUPPORT THAT IS REQUIRED
C
       IF(IOPT.EQ.1) THEN
        no_of_args = 0
        call pushstr('assert')
        call pushstr('ploadt')
        call pushstr(ploadt)
        call test

        no_of_args = 0
        call pushstr('assert')
        call pushstr('surflc')
        call pushstr(surflc)
        call test

        no_of_args = 0
        call pushstr('assert')
        call pushstr('suppwt')
        call pushstr(suppwt)
        call test

        no_of_args = 0
        call pushstr('assert')
        call pushstr('surfa')
        call pushstr(surft)
        call test

        no_of_args = 0
        call pushstr('call')
        call pushstr('support')
        call test

        call cc('u', strval(1), suptyp)
C      TRANSFER RESULT TO suptyp(1), CAPITALIZING THE LETTERS
       ENDIF
C
C   DETERMINE HOW DIAGONALS ARE TO BE DRAWN
C   BETWEEN MEMBERS OF A TRUSS
C
       IF(IOPT.EQ.2) THEN
        no_of_args = 0
        call pushstr('assert')
        call pushstr('dist1')
        call pushreal(tdist)
        call test

        no_of_args = 0
        call pushstr('assert')
        call pushstr('dist2')
        call pushreal(hdist)
        call test
```

```
           no_of_args = 0
           call pushstr('call')
           call pushstr('cmpsides')
C   ACTIVATE COMPARE_SIDES RULE IN PROLOG
           call test

           call cc('u',strval(1),SIDES)
           read(strval(2),'(F8.2)')SIDDIF
          ENDIF
C
C   DETERMINE IF BRACING CORRECT FOR QUADRILATERALS
C      IF ALPHA < 15 THEN BRACING IS NOT CORRECT
C      IF ALPHA > 75 THEN BRACING IS NOT CORRECT
C
          IF(IOPT.EQ.3) THEN
           no_of_args = 0
           call pushstr('assert')
           call pushstr('quadok')
           call pushreal(alpha)
           call test

           no_of_args = 0
           call pushstr('call')
           call pushstr('brcorrqd')
C  ACTIVATE BRACE_CORRECT RULE IN PROLOG
           call test

           call cc('u', strval(1),CHBRAC)
           read(strval(2),'(F8.2)')RATIO

          ENDIF
C
C   DETERMINE IF BRACING CORRECT FOR TRIANGLES
C      IF ALPHA < 15 THEN BRACING IS NOT CORRECT
C      IF ALPHA > 125 THEN BRACING IS NOT CORRECT
C
          IF(IOPT.EQ.33) THEN
           no_of_args = 0
           call pushstr('assert')
           call pushstr('triok')
           call pushreal(alpha)
           call test

           no_of_args = 0
           call pushstr('call')
           call pushstr('brcorrtr')
C  ACTIVATE BRACE_CORRECT RULE IN PROLOG
           call test

           call cc('u', strval(1),CHBRAC)
           read(strval(2),'(F8.2)')RATIO
```

```
      ENDIF
C
C  DETERMINE TYPE OF BRACING
C       IF ALPHA GE 40 DEGREES THEN Z TYPE IS CHOICE
C       IF ALPHA LT 40 DEGREES THEN V TYPE IS CHOICE
C
      IF(IOPT.EQ.4) THEN
       no_of_args = 0
       call pushstr('assert')
       call pushstr('alpha')
       call pushreal(alpha)
       call test

       no_of_args = 0
       call pushstr('call')
       call pushstr('brcetype')
C  ACTIVATE BRACE_TYPE RULE IN PROLOG
       call test

       call cc('u', strval(1),BRCTYP)

      ENDIF
C
C  DETERMINE IF BRACING IS NEEDED
C
      IF(IOPT.EQ.5) THEN
       no_of_args = 0
       call pushstr('assert')
       call pushstr('toleranc')
       tol = 100.0
       call pushreal(tol)
       call test

       no_of_args = 0
       call pushstr('assert')
       call pushstr('xn1')
       call pushreal(hdist)
       call test

       no_of_args = 0
       call pushstr('assert')
       call pushstr('dist')
       call pushreal(tdist)
       call test

       no_of_args = 0
       call pushstr('call')
       call pushstr('bracing')
C  ACTIVATE BRACING RULE IN PROLOG
       call test
       call cc('u', strval(1),CHBRAC)
       read(strval(2),'(F8.2)')RATIO
      ENDIF
C
C  DETERMINE NODES IN A TRIANGLE
C
      IF(IOPT.EQ.6) THEN
       do i = 1, ntotsp
       no_of_args = 0
         call pushstr('assert')
        call pushstr('elemntno')
        call pushreal(smemno(i))
        call pushreal(xs(i))
        call pushreal(ys(i))
        call pushreal(xe(i))
        call pushreal(ye(i))
        call test
       end do
       no_of_args = 0
       call pushstr('call')
       call pushstr('findtri')
C  ACTIVATE FIND_TRIANGLE RULE IN PROLOG
       call test

      ENDIF
C
C  WRITE EXPLANATION OF CHOICES
C
      IF(IOPT.EQ.8) THEN
       no_of_args = 0
       call pushstr('call')
       call pushstr('explain')
C  ACTIVATE EXPLANATION RULE IN PROLOG
       call test
      ENDIF
      RETURN
      END

       subroutine cc(code, fromstr, tostr)
       character*1 code
       character*(*) fromstr, tostr
        integer tolen, i, acode, zcode, bigacode, bigzcode

       acode = ichar('a')
       zcode = ichar('z')
       bigacode = ichar('A')
       bigzcode = ichar('Z')
       tolen = len(tostr)
       do i = 1, tolen
        tostr(i:i) = ' '
       end do
```

```
        if (code .eq. 'u' .or. code .eq. 'U') then
        do i = 1, len(fromstr)
         if (i .gt. tolen) goto 1000
         if (ichar(fromstr(i:i)) .ge. acode
     &      .and. (ichar(fromstr(i:i)) .le. zcode))then
          tostr(i:i) = char(ichar(fromstr(i:i)) - 32)
         else
          tostr(i:i) = fromstr(i:i)
         end if
        end do
        end if

        if (code .eq. 'l' .or. code .eq. 'L') then
        do i = 1, len(fromstr)
         if (i .gt. tolen) goto 1000
         if (ichar(fromstr(i:i)) .ge. bigacode
     &      .and. ichar(fromstr(i:i)) .le. bigzcode)then
          tostr(i:i) = char(ichar(fromstr(i:i)) + 32)
         else
          tostr(i:i) = fromstr(i:i)
         end if
        end do
        end if
1000        return
        end

        SUBROUTINE pushint(iarg)
        implicit none
        integer alfalength, maxargs
        PARAMETER (alfalength = 8, maxargs = 25)

        INTEGER intval(maxargs), iarg, no_of_args
        character*1 argtype(maxargs)
        common /fpcom2/ argtype
        common no_of_args
        common /fpcomi/ intval

        no_of_args = no_of_args + 1
        intval(no_of_args) = iarg
        argtype(no_of_args) = 'i'
        RETURN
        END

        SUBROUTINE pushreal(rarg)
c        implicit none
        PARAMETER (alfalength = 8, maxargs = 25)

        REAL rarg, realval(maxargs)
        INTEGER  no_of_args
        character*1 argtype(maxargs)
        character*(alfalength) strval(maxargs)
        common /fpcoms/ strval
common /fpcom2/ argtype
common no_of_args
 common /fpcomr/ realval


no_of_args = no_of_args + 1
realval(no_of_args) = rarg
read(strval(no_of_args),'(F8.2)')rarg
argtype(no_of_args) = 's'  ! reals get passed as strings
RETURN
END

SUBROUTINE pushstr(sarg)
implicit none
integer alfalength, maxargs
PARAMETER (alfalength = 8, maxargs = 25)

 character*(*) sarg
 character*(alfalength) strng
INTEGER no_of_args
character*1 argtype(maxargs)
common /fpcom2/ argtype
common no_of_args
character*(alfalength) strval(maxargs)
common /fpcoms/ strval

strng = sarg
no_of_args = no_of_args + 1
strval(no_of_args) = strng
argtype(no_of_args) = 's'
RETURN
END
```

Appendix 3

Implementation of Real Arithmetic

```
          SUBROUTINE IFACE

          implicit none
          integer alfalength, maxargs
          PARAMETER (alfalength = 8, maxargs = 10)

          CHARACTER*(alfalength) strval(maxargs)
          INTEGER intval(maxargs)
          REAL realval(maxargs)
          character*1 argtype(maxargs)

          integer no_of_args  ! for sharing with the
          common no_of_args   ! stacking routines only
          integer i
          real r1, r2

          common /fpcomi/ intval
          common /fpcomr/ realval
          common /fpcoms/ strval
          COMMON /fpcom2/ argtype

          no_of_args = maxargs
          DO i=1,maxargs
              IF (argtype(i) .eq. ' ') THEN
                 no_of_args = i - 1
                 goto 102

              END IF
          END DO
102         continue  ! loop exit target
C          PRINT *, 'iface: no_of_args =',no_of_args
C          We expect the first arg to be a command code
          goto (100,200,300,400,500,600),intval(1)
100         read(strval(2),'(F8.2)')r1
          read(strval(3),'(F8.2)')r2
C          print *, ' r1 = ',r1, ' r2 = ', r2
          IF (r1 .gt. r2) THEN
            strval(1) = 'gt'
          ELSE IF (r1 .eq. r2) THEN
            strval(1) = 'eq'
          ELSE IF (r1 .lt. r2) THEN
            strval(1) = 'lt'
          ELSE IF (r1 .le. r2) THEN
            strval(1) = 'le'
```

```
        ELSE IF (r1 .ge. r2) THEN
          strval(1) = 'ge'
        else
          print *, ' *** COMMAND CODE 2: ARGS NOT ORDERED'
        END IF
        no_of_args = 1
        argtype(1) = 's'
        goto 3000

200        read(strval(2),'(F8.2)')r1
        read(strval(3),'(F8.2)')r2
C        print *, ' r1 = ',r1, ' r2 = ', r2, ' sum = ', r1+r2
        write(strval(1),'(F8.2)')r1+r2
        no_of_args = 1
        argtype(1) = 's'
C        print *,' '   ! skip a line
        goto 3000

300        read(strval(2),'(F8.2)')r1
        read(strval(3),'(F8.2)')r2
        write(strval(1),'(F8.2)')r1-r2
        no_of_args = 1
        argtype(1) = 's'
C        print *,' '   ! skip a line
        goto 3000

400        read(strval(2),'(F8.2)')r1
        read(strval(3),'(F8.2)')r2
        write(strval(1),'(F8.2)')r1*r2
        no_of_args = 1
        argtype(1) = 's'
C        print *,' '   ! skip a line
        goto 3000

500        read(strval(2),'(F8.2)')r1
        read(strval(3),'(F8.2)')r2
        write(strval(1),'(F8.2)')r1/r2
        no_of_args = 1
        argtype(1) = 's'
C        print *,' '   ! skip a line
        goto 3000

600        read(strval(2),'(F8.2)')r1
        write(strval(1),'(F8.2)')abs(r1)
        no_of_args = 1
        argtype(1) = 's'
C        print *,' '   ! skip a line
        goto 3000
```

```
3000    do i= no_of_args+1, maxargs
          argtype(i) = ' '
        end do

        end
```

ORIGINAL PAGE IS
OF POOR QUALITY

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-178393	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Implementing Embedded Artificial Intelligence Rules Within Algorithmic Programming Languages		5. Report Date March 1988
		6. Performing Organization Code
7. Author(s) Stefan Feyock		8. Performing Organization Report No.
		10. Work Unit No. 506-43-41-01
9. Performing Organization Name and Address VAIR, Inc. 126 Kingsport Drive Williamsburg, VA 23185		11. Contract or Grant No. NAS1-18002
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: James L. Rogers

16. Abstract Most integrations of Artificial Intelligence (AI) capabilities with non-AI (usually FORTRAN-based) application programs require the latter to execute separately to run as a subprogram or, at best, as a coroutine, of the AI system. In many cases, this organization is unacceptable; instead, the requirement is for an AI facility that runs in embedded mode; i.e., is called as subprogram by the application program. This paper describes the design and implementation of a Prolog-based AI capability that can be invoked in embedded mode. The significance of this system is twofold: (1) Provision of Prolog-based symbol-manipulation and deduction facilities makes a powerful symbolic reasoning mechanism available to applications programs written in non-AI languages. (2) The power of the deductive and non-procedural descriptive capabilities of Prolog, which allow the user to describe the problem to be solved, rather than the solution, is to a large extent vitiated by the absence of the standard control structures provided by other languages. Embedding invocations of Prolog rule bases in programs written in non-AI languages makes it possible to put Prolog calls inside DO loops and similar useful control constructs. The resulting merger of non-AI and AI languages thus results in a symbiotic system in which the advantages of both programming systems are retained, and their deficiencies largely remedied.

17. Key Words (Suggested by Author(s)) Knowledge base, Prolog, Algorithmic language, Artificial Intelligence	18. Distribution Statement Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	50	A03