UCRL-52957 Rev. 1
Distribution Category
UC- 21, 32

Master

An introduction to Praxis

James R. Greenwood
Arthur Evans, Jr.*
C. Robert Morgan*
Michael C. Zarnstorff†

Manuscript date: December 3, 1980

*Bolt, Beranek, and Newman, Inc.
Cambridge, MA

†University of Wisconsin
Physics Department
Madison, WI

LAWRENCE LIVERMORE LABORATORY
University of California • Livermore, California • 94550

Available from: National Technical Information Service • U.S. Department of Commerce
5285 Port Royal Road • Springfield, VA 22161 • $6.00 per copy • (Microfiche $3.50)

CONTENTS

An introduction to Praxis

ABSTRACT

Praxis is the practice of the programming art, science, and skill. It is a high-order language designed for the efficient programming of control and systems applications. It is a comprehensive, strongly typed, block-structured language in the tradition of Pascal, with much of the power of the Mesa and Ada languages. It supports the development of systems composed of separately compiled modules, user-defined data types, exception handling, detailed control mechanisms, and encapsulated data and routines. Direct access to machine facilities, efficient bit manipulation, and interlocked critical regions are provided within Praxis.

Keywords: Praxis, high-level control language, compilers, real time.

Section 1

INTRODUCTION

This report describes the control-system implementation language Praxis, which has been developed in the Laser Fusion Program at the Lawrence Livermore National Laboratory (LLNL) for control applications. It serves as an introduction to the language so that the reader can get a *feel* for what the language is and find out if it is applicable to the reader's needs.

Most of the report consists of graduated examples that provide an overview of the language. The definition and details of the language can be found in the *Praxis: Language Reference Manual* and in other companion reports that follow the publication of this report.

Section 2

DEVELOPMENT HISTORY

In the summer of 1978, it became apparent in the laser fusion program at LLNL that we needed a **control-oriented** language for use in programming the control system of the Nova laser system. Our experience in developing the laser control system for Shiva, consisting of 55 processors, clearly indicated that if a controls-oriented programming language were available we could save considerable time and effort with respect to Nova.

After carefully evaluating potential languages, including DOD's current development of **Ada**, we chose to implement **Praxis**. Although Ada would meet our needs, it would not be available in time for Nova (compilers had to be available before the mid-1980's to meet the needs of the Nova controls programming). In retrospect, our selection of Praxis proved correct, since a Praxis compiler now exists and is in use while the more ambitious Ada development is still ongoing.

The development of Praxis originated from an initial study by Bolt, Beranek, and Newman (BBN), Inc., funded by the Defense Communications Agency (DCA), to determine the requirements of a language for communications programming. The result of that study (BBN Report 3261) concluded that no current language fulfilled the rigorous needs of communications programming.

The DCA then funded BBN to design an appropriate programming language. This resulted in a preliminary design of the COL language described in BBN Report 3534, May 1977 (A. Evans, C. R. Morgan). Also, the DCA funded BBN to design a compiler described in BBN Report 3533, May 1977.

In January 1979 LLNL funded BBN to augment the design of COL and to implement a COL compiler for the PDP-11 series of computers from Digital Equipment Corporation. With the clarification of the Nova controls design and schedule, BBN's work has been expanded to include the development of a VAX/VMS native-mode compiler, documentation, additional language design, and a high-level input/output package. BBN is scheduled to complete their work by fall 1980, with the delivery of documented operational compilers for Praxis, on both the PDP-11/RSX-11 and VAX/VMS systems, written in Praxis.

In January 1980 we changed the name of the language from COL to the current Praxis. We felt that the language had evolved significantly from that of the original COL study and that a new name would better reflect its power.

In March 1980 the preliminary PDP-11 compiler successfully passed two critical milestones. The first milestone was that the compiler, which is written in Praxis, had to compile itself successfully on the PDP-11/RSX-11M system. This would demonstrate that the compiler was self-supporting on the PDP-11 systems, and that the bulk of compiler was correctly implemented.

The second milestone was the implementation of a Nova controls application of the language, for a ROM-based LSI-11 processor. A 2000-line assembly-language, stepper-motor control program had to be recoded in Praxis, compiled, and *burnt* into read-only memory (ROM). This would demonstrate that the language was indeed powerful enough to replace detailed, assembly language sequences and that the compiler correctly implemented the controls-oriented features.

Section 3

INTENDED APPLICATIONS

Praxis is designed for programming control and communication applications. It is also useful for system programming applications, which require many of the same language facilities found in Praxis. All these applications impose stringent requirements on programming in such areas as

- Efficiency of object code.
- Direct access to machine facilities.
- Efficient bit manipulation.
- Complex data and control structures.
- Large programs developed by a team.
- Maintenance and upgrades.

The programming of these applications requires detailed control of the compiler-produced code, the optimization, the variable allocation, and the run-time support. In these applications, it is important for the programmers to explicitly control *exactly what is going on.*

Section 4

DESIGN GOALS

The design goals of Praxis are based on the requirement of the language being a useful tool for programming control applications. Consequently, the goals may be stated as follows:

- Efficiency: first of the compiled code, then of the compiler.
- Readability: particularly more important than writability.
- Completeness; in the sense that
  - * it must be possible to program all of any one application in Praxis without recourse to assembly language.
  - * it must be possible to write the compiler for Praxis in the Praxis language.
- Portability: Praxis should be reasonably machine-independent.
- Modularity: it must be possible to program large projects within Praxis, requiring separate compilation of modules and configuration control.
- Usability: primarily used by experienced programmers, so that the ease of learning Praxis is less important than the ease of using Praxis.

The primary requirements for control applications are efficiency of the compiled code, completeness, and portability. Praxis must produce programs that make effective use of hardware resources directly controlled by the programmer. Also, the programs should be as portable as possible between machines. In general, the language features are portable but, where machine-dependent parts are necessary, they are as conspicuous as possible. For example, the programmer can override the language's type-checking mechanism, but it is easy to see when this is being done.

The requirement for efficiency has had one other impact on the language design. All proposed features and facilities have to be scrutinized for the run-time and the compile-time efficiency of their implementation. No matter how desirable a particular feature might be, it had to be rejected if a reasonably efficient implementation could not be designed.

Section 5

LANGUAGE OVERVIEW

Praxis is a modern, block-structured, fully typed, algorithmic programming language in the tradition of Pascal. Its design has been influenced by the languages Simula, BCPL, Euclid, PL/I, Jovial, CS-4, Alphard, Mesa, and Bliss languages, as well as by the DOD's language development work and the proposed Ada language. In scope and power, Praxis most closely resembles Ada and Mesa.

Since the control environment differs in important ways from application to application and machine to machine, Praxis has features to handle these differences. High-level facilities that mask machine dependencies and foster machine independence (portability) usually prevent the use of exactly the programming capability needed for real-time, control applications programming. However, Praxis is a high-level language that has controlled access to machine dependencies.

Praxis is *strongly typed.* The programmer is given a collection of predefined types and has the ability to construct new types. Every variable, constant, parameter, and expression has a type. All types can be deduced at compile-time and the compiler requires that each value be used in a way that is consistent with the rules associated with its type. For instance,

it is a compile-time error to attempt to pass an integer parameter to a routine that requires a real parameter.

The language is *blocked structured.* Blocks are a method of packaging statements and declarations so that the scope of the statements is clearly specified and controlled. Praxis has more than 10 block-structured statements, each of which is delimited by an *XXX/endXXX* pair, where *XXX* represents the particular statement name. For instance:

```
for . . . . . endfor
if . . . . . endif
procedure . . . . . endprocedure
select . . . . . endselect
```

The block structuring also enforces a particular programming style that is more readable and maintainable than that of unstructured programming.

A simple example in the language is the matrix multiply of two N by N matrices named *SpecA* and *SpecB* and storing the result in *Spectrum*:

```
for I := 1 to N do
    for J := 1 to N do
        Spectrum [I,J] := 0
        for K := 1 to N do
            Spectrum [I,J] := Spectrum [I,J] + SpecA [I,K]* SpecB [K,J]
        endfor
    endfor
endfor
```

This example only makes sense within the scope of the declarations for the variables used. All the variables, except the one for loop indices, must be declared before use. Thus, the code above would be preceded by something of the form

```
declare
    N = 32                                  // constant
    SpecA : array [1..N,1..N] of integer    // an array variable
    SpecB : array [1..N,1..N] of integer    // an array variable
    Spectrum : array [1..N,1..N] of integer // an array variable
enddeclare
```

This declaration block could be written more concisely in various forms. One method would be to use a user-defined type for the array declarations which then would ensure that the three arrays are all the same type and remain so with subsequent software maintenance. Thus, the declarations could take the form

```
declare
    N = 32                                  // a constant
    matrix is array [1..N,1..N] of integer  // a type
    SpecA : matrix                          // an array variable
    SpecB : matrix                          // an array variable
    Spectrum : matrix                       // an array variable
enddeclare
```

Note that we have used the language's comment convention "//," which designates that all text to the right on the line is treated as a comment. Here, all language-reserved words are boldface in the examples, but no distinction is made in actual programs.

Another example is a simple exchange sort in which a values array is sorted into ascending order:

```
declare
    N = 100                              // a constant integer
    data : array [1..N] of integer       // an integer array variable
    done : boolean                       // a true/false variable
enddeclare
... code to store values in data ...
repeat
    done := true                         // nothing out of order found
    for K := 2 to N do
        if data [K-1] > data [K] do
            swap (data [K-1], data [K])  // if out of order, exchange them
            done := false                // not done yet
        endif
    endfor
until done
```

The **repeat** block-structured statement is the exception to the ending syntax rule, in that the **until** is the end for the repeat block. The **repeat/until** has the semantics that the included statements are executed repeatedly until the expression after the **until** is true. Other looping constructs are available in Praxis, including the **while/endwhile**, and four forms of **for/endfor**.

A more detailed control programming application is shown below. It directly reads a hardware input/output device on a PDP-11 computer in a multi-process environment. In this example, the resource (i.e., I/O device) is protected by the interlock variable *padlock* in a critical region. Another process with similar code, using the same resource, cannot preempt the critical-region code sequence.

```
Declare
    status : location (8!176420) volatile logical   // status register
    datum  : location (8!176422) volatile char      // input register
    padlock : static interlock                      // exclusion variable
    temporary : char
enddeclare
...
Region padlock do                                   // wait for device ready
    Repeat until (status and 8#200) <> 8#0          // read the character
    temporary := datum                              // lock the interlock
endregion                                           // unlock the interlock
```

The attribute **volatile** on the variables *status* and *datum* informs the compiler that the variables must be referenced directly each time they are mentioned in the program, and no optimizations are to be performed on these variables. It allows variables to be used as I/O registers, as above, as well as to be used in shared memory.

The **location** attribute informs the compiler to place the variable in the physical address specified by the octal (8!) integer constant in the parentheses. The variable is static and always resides at that location. The **static interlock** is at a fixed location determined by the compiler.

The **logical** predefined data type may be thought of as a bit-string data type on which bit-by-bit operations may be performed. In the **until** clause, a bit in the *status* variable is tested by the bit-by-bit **and** with the octal (8#) logical constant and comparison to a logical zero.

A more complex application, which demonstrates the ability in Praxis to bypass the strong typing (when desired), is the sequence that extracts the exponent value from a real number on the PDP-11:

```
Declare
    scale : real                                    // floating point variable
    power : integer                                 // signed integer variable
    temporary : logical                             // 16-bit bit-string variable
enddeclare
... code assigning value to scale ...
    temporary := ((force logical (scale)) rsh 8) and 8#177
    power := integer (temporary) - 8!100            // make -N to N
```

The **force** explicitly overrides the type-checking mechanism and specifies that the variable *scale* is to be handled as a logical in this expression. The logical value (i.e., 16 bits) is shifted right 8 bits and masked with the logical constant. *Temporary* is assigned the resulting value that was the exponent of the real variable *scale*. The logical value is then converted to an integer and stored in the variable *power*.

Note the distinction between **force** and type conversion; **force** informs the compiler to treat a variable as a particular type regardless of its actual type; conversion causes the variable to be converted to the desired type.

Another application of type conversion is shown in the function *upper*, which converts a possible lower-case letter to an upper-case letter:

```
function Upper (inchar:char) returns outchar:char
    if inchar < $a or inchar > $z do
        outchar := inchar                           // set returned value
        return                                      // exit if not lower-case letter
    endif
    outchar := char (integer (inchar) - 8!40)       // convert to upper-case
endfunction {Upper}
```

The previous function example utilized the **return** statement for explicit exit from a routine (i.e., procedure or function). This statement is one of several such statements that eliminates the need for a GOTO in the language. An important *feature* in the language is the *lack* of the GOTO statement. The following example uses two other control flow statements, together with block labeling, to program an application that normally requires a GOTO statement.

```
primary : For index := 0 to Bound do                // labeled statement
            size := Motor_size index                // assignment
            While Motor [index] = on do             // inner loop
                if size < mid_size do               // conditional statement
                    loop primary                    // iterate for loop
                orif size < max_size do             // alternative
                    break primary                   // exit for loop
                otherwise                           // default alternative
                    Slew_motor (index)              // procedure invocation
                endif                               // end of alternatives
            endwhile                                // end of inner loop
        endfor {primary}                            // end of labeled block
```

The **loop** statement above causes the **for** loop iteration to occur; that is, it acts like a GOTO the **for**, which causes the iteration count of the loop to be incremented, the test for completion to be performed, and the **for** block to be executed if the iterations have not been completed. The **break** statement on the other hand is a block exit statement. In the above case, it exits three levels of blocks: the **if**, **while**, and **for**, and execution continues after the **endfor**. Labels can only appear on blocks (at the beginning and end) and are only used with the **break**, **loop**, and **retry** (in critical **regions**) statements.

The statement sequence above would have had to be preceded by a declaration in which the variables, types, and constants are declared. All items must be declared before their use. The declaration for the above could be

```
Declare
    min_size = 0                              // a constant
    mid_size = 25
    max_size = 50
    size : integer initially min   size       // a variable
    bound : integer initially 0
    Onoff is [on, off] initially off          // an enumeration type
    Motor_size : array [0..9] of integer      // array variable
    Motor : array [0..9] of Onoff             // array variable
enddeclare
```

Notice the use of initialization clauses on variable and type declarations, which allow for variables to be declared with initial values. For instance, the variable *bound* is declared with the initial value zero, and the variable *motor* is declared as an array of enumerated data values, initially all elements being the value *off*. The declaration form **is** declares new data types and is discussed more fully below.

The **break** and **loop** example above also introduced the multiarm **if** statement that allows the programming of a *branch-tree*. Only one arm of the statement is elaborated on each iteration of the **while** loop, depending on the boolean expressions in each arm. Any number of **orif** clauses may be present, and the **otherwise** clause is optional. Thus, the forms below are valid **if** statements:

```
if (x = 0) or (y = 15) do
    ....
endif
if x = y do
    ....
otherwise
    ....
endif
```

Another form of flow control statement in Praxis is the **select** statement, which selects a sequence to elaborate from a set of cases according to a selection expression. For example:

```
Declare
    subsystem is [power, align, beam, target]   // a type
    system : subsystem initially beam           // a variable
enddeclare
```

```
....
select system from
    case power : Print ("Power subsystem")
    case align  : Print ("Alignment")
    default     : Print ("Others")
endselect
```

Only one of the *Print* procedure invocations is executed, depending on the value of the enumerated variable *system*. Note that the default clause will be executed for any values other than *power* or *align*. The strong typing and declarations ensure that the only other enumerated values the system can take on are *beam* and *target*.

Another control application that can be run on the PDP-11 uses data structures, procedure variables, and interrupt procedures to quickly and easily program an application that normally must be done in assembly language:

```
interrupt procedure clock_service ( )
        ticks := ticks + 1
endprocedure {clock_service}

declare
    vector is structure
        routine : interrupt procedure ( ) initially clock_service
        status : logical initially 8#340
    endstructure
    clock : location (8!100) vector
    ticks : static integer initially 0
enddeclare
```

The variable *ticks* gets incremented for each interrupt from the line clock on the PDP-11.

Note that because the interrupt procedure is executed asynchronously, communication with the other code must be done through **static** variables. Only one copy exists of any static variable.

The user-defined structure data-type *vector* has two fields: the first is the *routine*, which is of type **Interrupt Procedure** and is initialized to be the *address* of the clock service routine; the second field is a logical (i.e., bit-string) variable, which is set to the value desired for the processor status word. The actual declaration and positioning of the clock vector are accomplished by the variable declaration *clock* and the location attribute.

The above sequence would most likely be used in conjunction with a read routine of the form

```
function Read_ticks ( ) returns t : integer
    t := ticks
endfunction {Read_ticks}
```

The empty parentheses (i.e., ( )) denote a routine with no parameters and would be invoked with the form

```
count := Read_ticks ( )                    // get # of ticks
```

The interrupt-procedure example utilized the **structure** data type (i.e., the user-defined *vector*) and the **procedure** data type. These data types are two of the predefined data types in the language, all of which are listed below:

Discrete types

integer	- signed
cardinal	- unsigned integer
char	- ASCII character
boolean	- true/false
enumeration	- programmer-specified values

Control types

interlock	- locked/unlocked
logical	- bit string
pointer	- pointer to a typed object

Floating types

real	- floating point
long real	- double precision real

Aggregate types

array	- array of any type, access by index
structure	- various type components, access by name
set	- set of discrete type

Routine types

procedure	- typed procedure variables
function	- typed function variables

Other types

general	- union of all types (used as formal parameter)
descriptor	- *type* descriptor

User-defined data types may be characterized in terms of the predefined types or other user-defined types. The **is** form declares a user-defined data type. The semaphore in the example below is a user-defined data type. *Sync* is a variable of type semaphore:

```
declare
    semaphore is structure                  ! type decl
        lock : interlock
        count : integer initially 0
    endstructure
    Sync : semaphore                        ! a semaphore variable
enddeclare
```

This method for synchronization was proposed by Dijkstra in 1968. The semaphore is a special variable that can be manipulated only by the primitives Wait (also called the P operator) and Signal (also called the V operator), defined as follows:

```
procedure Wait (Sem : inout ref semaphore)      ! P operator
    Region Sem.lock do                          ! protect count access
        if Sem.count = 0 do                     ! check count value
            retry                               ! loop, unlock, a relock lock.
        endif
        Sem.count *= -1                         ! decrement count
    endregion                                   ! end of critical region
endprocedure {Wait}                             ! return from procedure

procedure Signal (Sem : inout ref semaphore)    ! V operator
    Region Sem.lock                             ! enter critical region
```

```
        Sem.count *= +1                    // increment count
    endregion                              // exit critical region
endprocedure {Signal}                      // exit from Signal
```

The *Wait* procedure allows a process to delay while waiting for an event to occur. The *Signal* procedure is used to signal another process that an event has occurred. In the above example, it is assumed that the semaphore would be shared between two processes, and each process would have its own copy of the *Wait* and *Signal* procedures. The interlock is utilized to guarantee *atomic* access to the semaphore count without worrying about actual code sequences.

The form "*=" assignment statement can be read as *transformed by*. Thus, the statement

```
Sem.count *= +1
```

increments the *count* field of the semaphore passed as an argument to *Signal* and is equivalent to the statement

```
Sem.count := Sem.count +1
```

The formal parameter specification on *Wait* (and *Signal*) explicitly specifies that the actual parameter be passed by **Ref** (i.e., reference) and that the parameter will be both read (i.e., **in**) and written (i.e., **out**). Parameters may be passed by **Ref** or **Val** (i.e., value, by copy) with the default being by **Val**. The programmer would usually specify by **Ref**, for large aggregates, in the interest of efficiency. The data-passing direction can be specified as **in**, **inout**, or **out** with the default being **in**. The compiler checks at compile-time to ensure that the usage of the parameter, within the routine, is consistent with the passing direction.

The *semaphore*, *Wait*, and *Signal* definitions can be encapsulated within a **Module** for separate compilation, or for data abstraction, or for both. Thus, the definition module would be

```
Module Semaphore_package
Export semaphore, Wait, Signal
Declare
    semaphore is hidden structure
        lock  : interlock
        count : integer initially 0
    endstructure
enddeclare
Procedure Wait (Sem : inout ref semaphore)
    .....
endprocedure {Wait}
Procedure Signal (Sem : inout ref semaphore)
    .....
endprocedure {Signal}
endmodule {Semaphore_package}
```

The declarations of *semaphore*, *Wait*, and *Signal* are made available by the **Export** to other modules (i.e., if this module was within another) or to other separately compiled modules that **Import** the declarations. Note that types as well as data and routines, can be imported and exported.

The **hidden** attribute specified on this new declaration of the semaphore type implements what is referred to as an abstract data type. That is, the type name is known outside of the module, but the internal structure is unknown. Thus, an application program can import the type and declare and use variables of type semaphore without knowing the details of the structure. For instance:

```
Main Module Joe_Schmoe
Import semaphore, Wait, Signal from Semaphore_package
Declare
    Async : segment (control_area) volatile semaphore
    Bsync : segment (control_area) volatile semaphore
enddeclare
While true do                                   // infinite loop
    Wait (Async)                                // process synchronization
        ....
        ....
    Signal (Bsync)                              // process synchronization
endwhile
endmodule {Joe_Schmoe}
```

The **main module** allows the use of top-level code (i.e., code not within a routine) and is the main program or process, depending on the operating system employed. In the example, two variables, *Async* and *Bsync*, are declared, using the imported semaphore definition. These variables are then used with the *Wait* and *Signal* procedure calls to synchronize this process with other processes. Note that the language makes no assumptions about the run-time system; no tasking or multiprocess operations are built into Praxis. These facilities can be programmed in the language, or provided by existing operating environments.

The **segment** storage class on the declarations of *Async* and *Bsync* specify that the semaphores are static in a named (i.e., control area) data area. This data area can be associated with program sections or location counters (depending on the implementation) by means of the **%Segment** compiler directive. For instance, for a PDP-11/RSX-11M implementation, the directive

```
%Segment control_area = RW, D
```

creates a program section (i.e., PSECT) which can be controlled and positioned at link-time. **Segment** can be viewed as a named **location**.

The *Print* routine used in a previous example could be written as

```
Procedure Print (string : in ref array [1..?length] of char)
    For index := 1 to length do
        Put_character (string [index])
    endfor
endprocedure {Type}
```

The formal parameter specifies a flexible array of characters as the type of the parameter; this allows the arrays of characters of any length to be passed, with an implicit-size parameter *length*. A quoted string is considered an array of characters indexed 1 through N, where N is the number of characters between the quotes.

Flexible arrays can also be allocated from the free memory storage (i.e., heap) and accessed through pointers. The free memory is only utilized when the programmer explicitly

specifies it by the **allocate** and **free** operations. There is no implicit heap usage or *garbage collection* in the language, an essential requirement in real-time control applications. Data objects in the heap are referenced by pointers. For instance:

```
Declare
    node is pointer structure
        address : integer
        status : logical initially 8#201
        data : array [-3..2] of real
        next : node initially nil
    endstructure
    head : node initially nil
enddeclare
    head := allocate node (address : 8!177560)
    if head@.data [2] = 0 do
        ....
    endif
```

The *node* declaration is a pointer to a structure of the form shown. *Head* is a declaration of a pointer object, and the assignment statement creates an object within the heap and places the location of the object in the variable *head*. The field *address* will be initialized to the octal value 177560, and the field *status* will be initialized to the octal value 201 via the type initialization clause.

The object is referred to with the "@" operator. That is, since *head* is a pointer to a structure, then

```
head@                - whole structure
head@.address        - an integer field
head@.data [J]       - an element of a field
head@.next           - a field
head@.next@.address  - a field of an object pointed to by a field
```

are valid references. Note that the last reference only makes sense if the value in the *next* field points to something (i.e., not **nil**).

The node pointer structure allows a linked list to be allocated at runtime from the heap. The iterator form of the **for** loop is useful for stepping through such a list.

```
For p := head then p@.next while p <>nil do
    if p@.status and 8#200 <> 8#0 do
        ....
    endif
    ....
endfor
```

The pointer variable *p* is declared and is assigned the value from *head*; if the value is not **nil** then the body of the **for** block is elaborated. The expression between the **then** and **while** is the iteration expression that specifies the subsequent values of *p*.

Objects allocated from the heap must be explicitly returned with the *free* procedure, which has the form

```
Free (p, head)                          // release P and head
```

Free may be called with any type of pointer and any number of parameters.

An important consideration in real-time systems is the ability to handle abnormal conditions and catastrophic failures. In Praxis, this is accomplished with named exceptions and guard blocks. Both predefined and user-defined exceptions are available and can be caught with a **guard** block. Thus,

```
Guard
    .....
    X := Y/Z
    .....
catch
    case divide_zero : Print ("Whoops")
endguard
```

would catch any divide-by-zero exception in the code between the **guard** and **catch** phases, or in any nested routines invoked from within the code. When and if a named exception occurs, the first (deepest) dynamically nested catch case for the named exception is elaborated. The **catch** clause can specify various named exceptions as well as use a **default** clause (i.e., all others).

Guard blocks may be used to contain exceptions in a large program or to catch an exception from a localized section. For instance, the Praxis input/output package uses exceptions for abnormal condition handling:

```
import Open, Open_error, file, Name_error from IO_package
Declare
    myfile : file
enddeclare
Guard
    Open
        with
            name   : "DB3: [Shiva] Test.dat"
            file_id : myfile
            access : default_access
        endwith
catch
    case open_error : Print ("Bad I/O")
                      raise Bad_IO
    case name_error : Print ("Bad filename")
endguard
```

The *Open* procedure invocation is surrounded by a **guard** block; the procedure upon detecting an error will **raise** the exception *open_ err* that is declared in the *IO_ package*. Control is transferred to the **case** clause in the **catch** block for the exception named. The clause is then elaborated. In the *open_ err* case, a routine is invoked and then a user-defined exception is explicitly raised, and elaboration continues in a higher-level guard block. For the *name_ err* exception, the case clause is elaborated and elaboration continues after the **endguard**. If no exceptions are raised within the *open*, then elaboration continues after the **endguard**.

The *Open* example also introduced an alternate procedure invocation, using named formal parameters and the list (i.e., **with-endwith**) format. The named parameters allow the use of optional parameters and parameter specification in any order. The name on the left of the colon (:) is the name of the **formal** parameter, and the name on the right is the **actual** parameter of the invocation. The declaration of the *Open* procedure could be of the form

```
procedure Open
    param
        file_id : in val file
        name : in ref array [1..?N] of char
        access : in val set of access_type
        window : optional in val 8 bit integer
                    initially 0
        share : optional in val set of sharing
                    initially empty_share
        . . more optional parameters . .
    endparam
. . procedure body . .
endprocedure Open
```

Only formal parameters declared as optional may be omitted in any actual invocation. Each formal parameter specified as optional must have a default value specified by the **initially** clause.

The **Guard** example introduced two procedures from the Input/Output package. The package is implemented as a series of procedures, functions, and abstract data types written in the language. Each implementation will have slightly different I/O packages, tailored to the particular operating environment. Under the PDP-11/RSX-11M and VAX-/VMS operating systems, the I/O package supports the full RMS-11 capabilities including indexed files. The standard, I/O-related, encapsulated data types are

file	record	stream	attribute

and some of the routines are

Create	Open	Close	Extend
Display	Erase	Connect	Disconnect
Find	Delete	Flush	Release
Get	Put	Rewind	Update

In addition, a set of conversion routines for the predefined data types are supplied, which convert to/from ASCII text.

Other packages are supplied with implementations, or are supplied as interfaces to existing packages in other languages. Praxis routines can invoke other language subroutines and functions, or they may be called from other languages. For instance, a Fortran mathematics package would be defined as

```
Module Math_package
Export Sin, Cos, Log
Fortran Function Sin (X : real) returns Y : real
. . . .
endfunction {Sin}
. . . .
endmodule {Math_ package}
```

Other than the Fortran linkage, Praxis provides the linkages

Inline - Place routine code in place of invocation
Interrupt - PDP-11 Interrupt service routine

Different compiler implementations could supply additional linkages.

An important feature that is necessary in the control environment is the ability to control the actual code generated for differing applications: for instance, the ability to generate code that would reside in ROM. This control is supplied by means of both predefined and user-defined compiler variables (comp_ var), in conjunction with compiler directives. For instance:

```
%define Author, three_D
%Set Author = "J R Greenberg"          // string comp_ var
%Set Object_ROM                        // predefined comp_ var
%Set three_D = true                    // user defined

Declare
    span is 0..5
    %if three_D or All_three
        data : array [span, span, span] of real
    %otherwise
        data : array [span, span] of real
    %endif
enddeclare
```

Compiler variables can be either boolean or string types and are explicitly declared and assigned to by the %Set compiler directive. The comp_ var *Object_ ROM* specifies that the code generation should be such that the code and constant data can be *burned* into ROM. The %if-%otherwise-%endif allows conditional compilation under control of a boolean comp_ var expression. The referenced comp_ var values can be set either within program text or upon compiler invocation.

Another feature that needs mentioning is the ability to generate specific instructions or nonstandard calling sequences. This is provided by the block-structured **code** statement shown below for a PDP-11 application:

```
Procedure Trigger (X : integer)
    Declare
        timer : static integer
        index : integer
    enddeclare
    code "PDP-11" do
        MOV #33, index(SP)          // set a count
        MOV X(SP), R1               // pass parameter
    LP: INC timer                   // strobe
        TRAP                        // go to another routine
        DEC index(SP)               // count
        BNE  LP                     //
    endcode
    if timer = X do
```

```
        ....
      endif
endprocedure {Trigger}
```

The concluding example outlines a simple task processor, using arrays of procedure variables and the **set** data type:

```
Declare
        number is integer range 0..5                    // range of integers
        active is set of number                         // set of integers
        active_tasks : static active initially active () // a set variable
        task is procedure ()                            // procedure type
        task_list : static array [number] of task       // list of possible tasks
enddeclare
....
Procedure Activate (task_id:number)
        Active_tasks *= + Active (task_id)              // place in set
endprocedure {Activate}
....
For index in active_tasks do                            // scan all active tasks
        task_list [index] ()                            // invoke task
endfor                                                  //
```

The set data type in the declaration of *active* is used as an attribute associated with each *task*. The set has six possible members denumerated by the values 0 through 5. Sets can be of any discrete type and can be arbitrarily large (i.e., limited by memory size of machine). The *active* () after the **initially** clause and in the assignment statement is the set constant constructor, which allows items from the set to be included or removed. The **For** statement iterates through the set of *active_tasks* and will invoke any *active* task.

Section 6

SUMMARY

The preceding section, although introducing many features of the Praxis language, is by no means exhaustive. Some features have not been mentioned, and others have only been partially described. The full language is described in *Praxis: Language Reference Manual* and the *Programming in Praxis* manual.

The Praxis language is specifically **within** the state of the art of language design, particularly designed for control and system implementation needs. Complex language features, such as generic procedures, overloading of operators, and parallel processes, have been intentionally left out. We felt that these concepts were either not understood enough to be incorporated at this time, or that they need not be part of the language.

In conclusion, Praxis is an extremely powerful, modern programming language that goes beyond Pascal and is available today.

ACKNOWLEDGMENTS

The original language was designed by Arthur Evans, Jr., and C. Robert Morgan of BBN in 1977. Additional language design in 1979 by Evans and Morgan was augmented by James R. Greenwood (LLNL) and Michael C. Zarnstorff (University of Wisconsin). The final language design in 1980 was developed by the above individuals, with contributions from Earl Killian (BBN), Graeme Williams (BBN), and W. Nowicki (Stanford University).

The continued support of the management of the laser fusion program and the Nova laser project at LLNL, in particular J. L. Emmett, J. F. Holzrichter, R. O. Godwin, and W. W. Simmons, is gratefully acknowledged. The encouragement and support of H. Ahlstrom and L. Coleman of the fusion experiments program at LLNL is also greatly appreciated.

The tremendous effort by F. Holloway in developing the first application program in Praxis for the Nova control system is hereby acknowledged. His patience with early compiler releases, his persistence in developing the application acceptence test, and his constant enthusiasm were invaluable to the success of the project.

Additional thanks go to G. J. Suski, P. Rupert, and the controls development group at LLNL for their willingness to attempt the project and suffer through the preliminary versions of the product.

Also, the dedicated support and documentation efforts by W. Nowicki was essential. In particular, his work on the *Programming in Praxis* manual came at a critical time.

The documentation and support role of J. Walker and R. Shapiro at BBN was extremely valuable. J. Walker created the *Language Reference Manual* in a short period of time from an everchanging definition.

BIBLIOGRAPHY

Many languages are identified in the body of this report without specific references. Citations are as follows:

Ada	(Ichbiah-79A-79B)
ALGOL-60	(Naur-63)
ALPHARD	(Wulf-76)
BCPL	(Richard-69), (BBN-74)
BLISS	(Wulf-71)
CS-4	(Intermetrics-75)
EUCLID	(Lampson-77)
FORTRAN	(FORTRAN-76)
IMP	(Irons-70)
JOVIAL	(Shaw-63)
Mesa	(Mitchell-79)
Pascal	(Jensen-74)
PL/I	(IBM)
Simula	(Dahl-70)

(BBN-74)

BCPL Manual, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts (1954).

(Brinch-Hansen-72)

P. Brinch Hansen, "Structured Multiprogramming," *Comm. ACM* **15**, 7, 574-578 (1972).

(Brinch-Hansen-73)

P. Brinch-Hansen, *Operating Systems Principles*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1973).

(Dahl-70)

O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Common Base Language*, Norwegian Computing Center, Publication S-22 (1970).

(DoD-77)

"Department of Defense Requirements for High-Order Computer Programming Language—Ironman," January 14, 1977.

(Evans-76)

A. Evans, Jr., and C. R. Morgan, *Development of a Communications Oriented Language*, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 3261 (1976)

(Evans-77)

A. Evans, Jr., and C. R. Morgan, *A Communications Oriented Language (COL): Language Design*, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 3534 (1977).

(Evans-79)
A. Evans, Jr., C. R. Morgan, E. S. Roberts, and E. M. Clarke, *The Impact of Multiprocessor Technology on High-Level Language Design*, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 4188 (1979).

(Fisher-76)
D. A. Fisher, "A Common Programming Language for the Department of Defense—Background and Technical Requirements," Institute for Defense Analysis, Paper P-1191, June 1976

(FORTRAN-76)
"Draft proposed ANS FORTRAN," *ACM Sigplan Notices* **11**, 3 (1976) (entire issue).

(IBM)
"PL/I Language Specification," IBM Corporation, ANSII Standard for PL/I, Subset G, Form GY33-6003-2 (undated).

(Ichbiah-79A)
J. D. Ichbiah, J. Heiard, O. Roubine, J. Barnes, B. Krieg-Brueckner, and B. A. Wichmann, "Rationale for the Design of the Ada Programming Language," *ACM Sigplan Notices* **14**, 6 (1979).

(Ichbiah-79B)
J. D. Ichbiah, J. Heiard, O. Roubine, J. Barnes, B. Krieg-Brueckner, and B. A. Wichmann, "The Preliminary Ada Language Reference Manual," *ACM Sigplan Notices* **14**, 6 (1979).

(Intermetrics-75)
CS-4 Language Reference Manual and Operating System Interface, Intermetrics, Inc., Cambridge, Massachusetts, Report IR-130-2 (1975).

(Irons-70)
E. T. Irons, "Experience with an Extensible Language," *Comm. ACM* **13**, 1 (1970).

(Jensen-74)
K. Jensen and N. Wirth, *PASCAL User Manual and Report* (Second Edition), Springer-Verlag, Berlin (1974).

(Knuth-73)
D. E. Knuth, *A Review of Structured Programming*, Stanford University, Stanford, California, Computer Science Department, Report STAN-CS-73-371 (1973).

(Knuth-74)
D. E. Knuth, "Structured Programming with Goto Statements," *Computing Surveys* (December 1974).

(Lampson-77)
B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchel, and G. J. Popek, "Report on the Programming Language EUCLID," *ACM Sigplan Notices* **12**, 2 (1977) (entire issue).

(Mitchell-79)
J. G. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual V5*, Xerox Corporation, Palo Alto, California, Report CSL-79-3 (1979).

(Morgan-77)
C. R. Morgan and A. Evans, Jr., *Communications Oriented Language (COL): Language Implementation*, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 3533 (1977).

(Naur-63)
"Revised Report on the Algorithmic Language ALGOL 60" (P. Naur, Ed.), *Comm. ACM* **6**, 1, 1-17 (1963).

(Richards-69)
M. Richards, "BCPL—A Tool for Compiler Writing and Systems Programming," from *Spring Joint Computer Conference* (1969), pp. 557-566.

(Shaw-63)
C. J. Shaw, "A Specification of JOVIAL," *Comm. ACM* **6**, 12, 721-736 (1963).

(Wirth-76)
N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1976).

(Wulf-71)
W. A. Wulf, D. B. Russell, and A. N. Haberman, "BLISS: A Language for System Programming," *Comm. ACM* **14**, 12, 780-790 (1971).

(Wulf-76)
W. A. Wulf, R. L. London, and M. Shaw, *Abstraction and Verification in ALPHARD: Introduction to Language and Methodology*, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Department of Computer Science (June 1976).

(Zahn-74)
C. T. Zahn, "A Control Structure for Natural Top Down Structured Programming," from *Symposium on Programming Languages*, Paris, France (1974).

BJ/jvb

U. S. Government Printing Office: 1981/1—789-002/5531

Appendix

LANGUAGE SYNTAX

Backus-Naur Form (BNF)

Here, we describe the context-free syntax of the language, using a variant of the Backus-Naur Form (BNF). In particular, we adhere to the following conventions in the BNF representation:

- Lower-case words, perhaps containing underscores, denote syntactic categories, such as:

  function_list
  relation_operator
  linkage

- Boldface words denote reserved words, for example.

  select
  function
  or

- Square brackets enclose optional items. A quoted square bracket means that it is part of the syntax (i.e., array subscripts and enumerations).

  endif [[label]]
  [mode] function ....
  [access_mode] **structure** ....

  array '[' subscript,...']' **of** type
  for ID **in** '[' enumeration,...']'

- Repeated items are represented by a delimiter followed by three dots. Thus, a list of identifiers could be designated by

  identifier,...

  where the comma is the repeat delimiter. Thus, the BNF form,

  identifier_list := identifier,...

  means that the identifier list can contain one or more identifiers separated by commas. Another example is

  statement_list := statement;...

  where the semicolon is the delimiter.

- The syntax rules describing structured constructs in the language are presented in a form that is visually similar to their usage in programs. For example, the select statement is specified in the BNF as

```
select_statement := label:  select expression from
                              [case case_literal,... : sentence;... ] ...
                              [default : sentence;...]
                            endselect |label|
```

- Various syntactic items can be represented by the item prefixed by a qualifier corresponding to a category name. The prefix is intended to convey extra semantic information. For instance:

  module_identifier module_ID function_identifier

  are all equivalent to:

  identifier

- Some abbreviations used in the syntax description are

ID	identifier
expr	expression
spec	specifier
c_constant	compile-time constant
l_constant	link-time constant

- The slash (/) is used to delimit various cases of a BNF production. It can be read as "or."

  Thus:

```
declaration  ::= procedure_declaration
               / function_declaration
```

  is just shorthand for

```
declaration  ::= procedure_declaration
declaration  ::= function_declaration
```

Syntax Definition

```
module declaration   ::= [main] module module_ID segment_list
                           export ID,... [to module_ID,...] ;...
                           sentence;...
                         endmodule [|module_ID|]

module ID            ::= ID / module_ID.ID

sentence             ::= statement / declaration / empty

declaration          ::= procedure_declaration
                       / function_declaration
                       / listed_declaration
                       / import_declaration
```

```
                        / module_ declaration
                        / exception_ declaration

statement               ::= assignment_ statement
                        / invocation_ statement
                        / iterative_ statement
                        / flow_ statement
                        / special_ statement
                        / miscellaneous_ statement

import_ declaration     ::= import ID,... from module_ ID
                        / use module_ ID

segment_list            ::= segment (segment_ ID,...) [aligned (c_ const,...)]
```

Declarations

```
procedure_ declaration  ::= forward [mode] procedure procedure_ ID procedure_ spec

                        / [mode] procedure procedure_ ID procedure_ spec
                              sentence;...
                          endprocedure [{procedure_ ID}]

mode                    ::= inline / fortran / interrupt

procedure_ spec         ::- parameter_ spec [segment_ list]

parameter_ spec         ::= (parameter,...) / ()
                        / param
                              parameter;...
                          endparam

function_ declaration   ::= forward [mode] function function_ ID function_ spec

                        / [mode] function function_ ID function_ spec
                              sentence;...
                          endfunction [{function_ ID}]

function_ spec          ::= parameter_ spec returns variable_ spec {segment_ list}

variable_spec           ::= variable_ ID,... : [storage] type [initial]

storage                 ::= static
                        / location (l_ constant)
                        / register (register_ spec)
                        / segment_ spec

initial                 ::= initially expression

parameter               ::= ID,... : [call_ type] [storage] type [default] [desc_ clause]
```

segment_spec ::= **segment** (segment_ID) [**aligned** (c_constant)]

desc_clause ::= **with** descriptor_ID

call_type ::= **variadic** call_type
/ [**optional**] [**volatile**] [**in** / **out** / **inout**]
[**ref** / **val**]

default ::= **initially** expression

Type ::= [***different***] [attribute_list] base_type [constraint]
[initial] [abstract_list]

attribute ::= **hidden** / **readonly**
/ **volatile**
/ **packed** / **packed packed** / **unpacked**
/ c constant **bit**

constraint ::= **range** discrete_type

abstract_list ::= abstraction / abstraction abstraction_list

abstraction ::= **starting** [mode] **procedure** procedure_spec
/ **finishing** [mode] **procedure** procedure_spec
/ **in** zone_ID

base_type ::= basic_type
/ discrete_type
/ aggregate_type
/ special_type

listed_declaration ::= **declare** (decl)
/ **declare**
decl;...
endeclare

decl ::= variable_spec
/ constant_ID,... = l_constant
/ type_ID,... **is** [**different**] type [initial]
/ zone_declaration

basic_type ::= **integer** / **real** / **logical** / **char** / **long_real** /
/ **interlock** / **cardinal** / **boolean**

discrete_type ::= limit..limit
/ '[' enumeration_ID,...']'
/ base_type

limit ::= expression / ?ID

zone_declaration ::= zone_ID : storage **zone** (parameter,...)

special_type ::= **pointer** type
/ **descriptor**
/ **general**
/ [mode] **procedure** procedure_spec
/ [mode] **function** function_spec

aggregate_type ::= **array** '[' discrete_type,...']' **of** type
/ **structure**
field;...
endstructure
/ **set of** type

field ::= **fill** (c_constant **bit**)
/ field_id,... : type
/ **select** tag_ID **from**
[**case** case_label,... : field;...] ...
endselect

case_label ::= c_constant .. c_constant / c_constant

exception_declaration ::= **exception** exception_ID,...
/ **arm** comp_var_ID,...
/ **disarm** comp_var_ID,...

Statements

assignment_statement ::= expression := expression
/ expression *= infix_op expression

invocation_statement ::= procedure_ID (expression,...) / procedure_ID ( )
/ procedure_expression
/ procedure_ID (parameter_ID: expression,...)
/ procedure_ID
with
parameter_ID: expression;...
endwith

procedure_expression ::= expr_10

iterative_statement ::= [loop_label:] **while** boolean_expression **do**
sentence;...
endwhile [[end_label]]
/ [loop_label:] **repeat**
sentence;...
until boolean_expression [[end_label]]
/ [loop_label:] **for** for_element **do**
sentence;...
endfor [[end_label]]

```
for_ element        ::= for_ ID := expression downto expression
                     / for_ ID := expression to expression
                     / for_ ID := expr then expr while boolean_ expr
                     / for_ ID in discrete_ type
                     / for_ ID in set_ type

flow_ statement     ::=  break label
                     /    loop [loop_ label]
                     /    return
                     / [begin_ label:] if boolean_ expr do
                                          sentence;...
                                       orif boolean_ expression do
                                          sentence;...
                                       otherwise sentence;...
                                       endif [{end_ label}]

flow_ statement     ::= [begin_ label:] select expression from
                                          case case_ label,... :
                                            sentence;... ...
                                          default : sentence;...
                                       endselect[{end_ label}]
                     / [begin_ label:] upon viaduct_ ID,... leave
                                          sentence;...
                                       through
                                          case viaduct_ ID : sentence;... ...
                                       endupon [{end_ label}]
                     / via viaduct_ ID

special_ statement  ::= [begin_ label:] region interlock_ expression do
                                          sentence;...
                                       otherwise
                                          sentence;...
                                       endregion [{end_ label}
                     / retry
                     / [begin_ label:] guard
                                          sentence;...
                                       catch
                                          case exception_ ID,... : sentence;...
                                          default : sentence;...
                                       endguard [{end_ label}]
                     / raise exception_ id [finishing ID,...]
                     / reraise [finishing ID,...]
                     / [begin_ label:] block
                                          sentence;...
                                       endblock [{end_ label}]

special_statement   ::= [begin_ label:] code "machine_ designator" do
                                          instruction;...
                                       endcode [{end_ label}]
```

instruction ::= assembler_ instruction

misc._ statement ::= **free** (pointer_ type_ expression:...)
/ ***swap*** *(expression, expression)*
/ **assert** boolean_ expression

Expressions

The numeric values on the "expr" identifiers below represent the operator precedence levels.

expression ::= expr_ 0
/ **when** boolean expr **then** expr **else** expr

expr_0 ::= [expr_ 0 **eqv**] expr_ 1
/ expr_ 0 **xor** expr_ 1

expr_1 ::= expr_ 2 [***or*** expr_ 2]

expr_2 ::= expr_ 3 [**and** expr_ 3]

expr_3 ::= [**not**] expr_ 4

expr_4 ::= expr_ 5 [relational_ operator expr_ 5]

expr_5 ::= expr_ 6 [shift_ operator expr_ 6]

expr_6 ::= [expr_ 6 addition_ operation] expr_ 7

expr_7 ::= expr_ 8 [multiplication_ operator expr_ 8]

expr_8 ::= [unary_ sign] expr_ 9

expr_9 ::= expr_ 10
/ **allocate** expr_ 10
/ **force** expr_ 10

expr_10 ::= ID / constant / expr_ 10 (expression,...)
/ expr_ 10 (field_ value,...)
/ (expression)
/ expr_ 10 '[' expression,... ']'
/ expr_ 10 . field_ ID
/ expr_ 10 @
/ expr_ 10 **with** parameter_ ID : expression;... **endwith**

field_ value ::= field_ ID : expression /
'[' case_ element ']' : expression

Operators

Infix_ operator	::= **eqv** / **xor** / **or** / **and** / relational_ operator / shift_ operator / addition_ operator / multiplication_ operator
relational operator	::= = /<>/<=/</>=/>
shift_ operator	::= **lsh** / **rsh**
multiplication_ operator	::= * / '/' / **mod**

Predefined Functions

max	- maximum
min	- minimum
succ	- successor
pred	- predeccessor
abs	- absolute value
round	- real to integer rounded
floor	- largest integer not greater than real
ceiling	- smallest integer not less than real
low	- lower limit of discrete type
high	- upper limit of discrete type
size_ of	- size in bits of data object
descriptor_ of	- descriptor of a type

USGPO-1980: 789-002/5531