Multiprocessing CODA: Applying PLITS Formalism to a Quasi-parallel Model of SCHEME*

by

Joseph R. Ginder

Indiana University

TECHNICAL REPORT NO. 92

MULTIPROCESSING CODA: APPLYING PLITS FORMALISM
TO A QUASI-PARALLEL MODEL OF SCHEME*

JOSEPH R. GINDER

INDIANA UNIVERSITY

MAY, 1980

* Submitted to the Department of Computer Science in partial fulfillment of the requirements for the Bachelor of Arts degree in Computer Science with Honors.

Multiprocessing CODA: Applying PLITS Formalism to a Quasi-parallel Model of SCHEME

Abstract:

Coda is an iterative interpreter for SCHEME, a dialect of LISP. SCHEME is an applicative-order, lexically-scoped, full-funarg lambda calculus system. The CODA interpreter evaluates tail-recursive functions without net growth of the LISP interpreter stack. When a stack is needed, it is kept explicitly in the form of continuation-links, collections of registers to be used for evaluation of an expression. Multiprocessing is simulated by the CODA interpreter, allowing processes to execute "in parallel". Multiprocessing is available with either shared environments or distributed processing. Monitors can easily be defined as processes with a unique environment; communication is facilitated by a PLITS-like message passing system. Primitive operations are provided for message manipulation, process synchronization, and new process creation. The entire system is implemented in LISP.

Multiprocessing CODA: Applying PLITS Formalism to a Quasi-parallel Model of SCHEME

1. Introduction

CODA is an iterative interpreter capable of interpreting either LISP or SCHEME [12] expressions. LISP expressions are evaluated in the usual manner, lambda-expressions being closed in the execution environment. SCHEME is an applicative-order, lexically-scoped, full-funarg dialect of LISP; lambda-expressions are closed in the definition environment [12]. In both cases, the interpreter stack is kept explicitly in the form of continuation links (c-links), which are collections of the interpreter register values at the time when that link was pushed onto the stack.

The major part of this project was to implement a version of this interpreter in which multiprocessing was simulated ("quasiparallel" [9]). This was accomplished without confining the multiprocessing to either a distributed processing or shared environment model; rather both are available. Processes are usually defined as SCHEME expressions using the "LABELS" syntax (see [12] for discussion of LABELS). Particular instances of process definitions are specified at the initiation of multiprocessing or through use of the "create-process" primitive. Each process is then evaluated in the specified environment until it returns a value (or indefinitely if no final value is reached). It should be emphasized that no fairness is guaranteed or implied for process evaluation -- the processes are evaluated one at a time in a completely random order, for one cycle of the interpreter and then suspended. A cycle is defined as one evaluation of the #pc# register. In practice, this usually means evaluating one top level element of the current expression register. This may entail recovering a value from the environment or determining that the current element is a list and pushing a c-link onto the c-link stack for this list to have its elements evaluated. The stack is popped after the last element of the current expression is evaluated. Values are carried from c-link to c-link in the accumulator register. A cycle may last for several times longer than usual if the form being evaluated has the property "INDIVISIBLE". In this case, the entire form is completely evaluated before suspension of the process.

Shared environments are facilitated by simply passing the same environment pointer to several processes in the interpreter call. Note that partially shared environments would result from passing processes pointers to different parts of the same environment. A process modifies its environment using either of the assignment primitives "aset" or "cset". These assignment operations are indivisible operations. The primitive "aset" modifies the environment by changing an old binding if one is found, or adding a new pair (identifier, value) to the END of the current environment (NCONC [10,13]). This is an assignment that is visible to all processes sharing the environment in which the assignment was made. "Cset", alternatively, is a primitive that always adds a new pair onto the front of the environment. This means that the scope of this binding is as narrow as possible, and is no greater than that of any argument bindings of the expression being evaluated; as soon as the c-link is popped, it will disappear.

Distributed processing is accomplished by specifying either a null environment at process initiation, or specifying a unique environment to which no other process has a pointer. The assignment primitives work exactly the same as with shared environments.

The implementation of these assignment primitives and several other multiprocessing primitives introduced a new property name for functions into the interpreter. These functions are defined with property-name "INDIVISIBLE". When the interpreter "GET"s an indivisible property of an identifier, any processing guided by that definition takes place immediately, without interruption, but otherwise exactly as any other processing.

The remainder of this paper is organized into several sections and appendices. The next section contains a discussion of new primitives defined for use in this system. The third section is made up of a discussion of various examples. A conclusion follows these sections. The code for the system is included in several appendices along with several notes on the implementation of the system.

2. Primitives

Process creation and manipulation is accomplished through the use of the following primitives. Several are based in part on various procedures defined by Hansen for the RC 4000 operating system [5] and by Holt, et. al. [8]. The implementation of primitives was undertaken with Hoare's axiomatic approach [7] in mind.

```
(create-process <process-definition>
                <environment>
                <process-name>)
```

Here, <process-definition> is a form to be multiprocessed by the interpreter; <environment> is a list of pairs or NIL; and <process-name> is an identifier to be used by other processes to refer to the process being created. "Create-process" sets up a suspended process closure which will, when activated, evaluate the process-definition in the specified environment. Of course, many instances of the same process-definition may be active in parallel.

```
(activate <process-name>)
```

This primitive activates the suspended process referred to by <process-name> by placing it in the pool of processes being evaluated.

(deactivate)

The primitive "deactivate" makes a suspended process closure of the calling process and removes it from the pool of processes being evaluated.

(aset <name> <value>)

"Aset" is an indivisible operation that binds the value specified by <value> to the identifier specified by <name> (ie. "SET", not "SETQ" as in [10,13]). This operation either updates an already present name-value pair or adds a new pair at the end of the environment.

(cset <name> <value>)

"Cset is analagous to "aset" except that "cset" always adds a new pair, name-value, to the front of the environment.

(termination-value <process-name>)

This primitive returns the value to which the process referred to by <process-name> evaluated upon termination. This value is the contents of the #accumulator# register at process termination (see Appendix B).

(suicide)

The primitive "suicide" immediately terminates execution of the calling process, prints a suicide message, and leaves a suicide note as the value of the process. This primitive executes uninterruptedly.

(extantp <process-name>)

This predicate returns T if the process referred to by <process-name> is in the active process pool.

Communication between processes in a distributed system can be defined within any of several paradigms for message passing. Synchronization between processes is accomplished using message passing in a unique way within each paradigm. In Feldman´s PLITS system [2,3], a process sends a message and then proceeds with execution; synchronization is accomplished by causing a process to wait until an appropriate message is available upon issue of a "receive" command. In Hoare´s CSP system [6], the reverse option is in effect; process waiting occurs when a "send" is issued until the destination process issues a "receive". In Brinch Hansen´s DP system [4], communication must be exchanged in both directions before either process can proceed. In any of these paradigms, the other paradigms can be simulated. This system employs a paradigm not quite like any of the above, but very close to PLITS. PLITS-like communication takes place when the "send" and "receive-wait" primitives are used. Messages are lists of pairs, or environments; and messages are given a source pair and transaction pair -- both as in PLITS. A description of communication primitives follows.

(close-communications)

The primitive "close-communications" closes the calling process´ queue so that no messages can be delivered. This is an indivisible operation.

(open-communications)

This primitive opens the calling process´ queue so that messages can be delivered. This is also an indivisible operation.

(send <destination> <message>)

"Send" delivers the message <message> to the process referred to by the identifier <destination>. <message> is a list of pairs (a PLITS message) created by the primitive "create-message". If the process being sent to has closed itself to communication, "send" returns NIL and no action is taken.

(receive <source> <transaction>)

This primitive attempts to retrieve a message from the queue of the calling process that is from the process referred to by <source> and about the subject identified by <transaction>. It should be noted that <source> and <transaction> can both be specified as null, thereby enabling a message from any process or about any subject to be retrieved. If no appropriate message is found, NIL is returned and no action taken; if more than one appropriate message is present in the queue, the oldest is taken.

```
(send-wait <destination> <message> <continuation>)
```

"Send-wait" attempts to deliver a message to the queue of the process referred to by <destinations> until it is successful. This means that if the queue is closed, the calling process will wait until it is opened. This does not mean that the destination process must "receive" the message; it only means that it must be queued. <continuation> is a form that will become the new expression to be evaluated by the interpreter after the message is successfully sent. In essence, this primitive sends the message after waiting however long is necessary, and then <continuation> acts as a GOTO to specify further processing.

```
(receive-wait <source>
              <transaction>
              <identifier>
              <continuation>)
```

The primitive "receive-wait" causes the calling process to attempt to receive a message from the process specified by <source> about the subject identified by <transaction>. Either or both of these descriptions of an appropriate message can be specified as null so that any message will match that description. If no message is present, then the process must wait until one becomes available; if a message is available, it is retrieved and bound to an identifier in the calling process' environment, <identifier> (using aset). This is necessary because receive-wait (like send-wait) uses the

<continuation> expression to specify further processing after a reception is made, so no value is returned from receive-wait -- but the message must be made accessible to the calling process in some way.

(clearq)

"Clearq" is a primitive that deletes all messages from the queue of the calling process. It is an indivisible operation, but usually should be called while a process has closed itself to communication.

(openp <process-name>)

This predicate returns T if the process referred to by <process-name> is open for communication, NIL otherwise. It is an indivisible operation.

(closedp <process-name>)

This predicate returns the opposite of the value returned by "openp" under the same conditions. It is also indivisible.

(create-message <slot-list> <transaction>)

"Create-message" returns a PLITS-like message of the slots (pairs) specified by <slot-list> with slots for "FROM" and "ABOUT" added in. All of these slots are accessible and modifiable using the provided primitives.

(slot-assign <slotname> <value> <message>)

This primitive assigns <value> as the value of the slot with name <slotname> in the message <message>. If no slot with the specified slot name exists, then one is created and assigned to.

(slot-value <slotname> <message>)

The primitive "slot-value" returns the value of the slot with slotname <slotname> in <message>. If the requested slot does not exist, the message "SLOT-NOT-FOUND" is returned to avoid confusion with NIL slot-values.

(presentp <slotname> <message>)

This predicate returns T if a slot with slotname <slotname> is found in <message>, NIL otherwise.

Communication between processes is intended to be much like that in a PLITS system [2,3]. Messages are pairs or slots of identifiers and values, much like a LISP environment. (One difference between a message and a LISP environment is that in a message an identifier can only appear once.) Each message has a "FROM" slot and an "ABOUT" slot added by the "create-message" primitive. If sending of messages is confined to the use of the "send" primitive, and receiving is confined to the use of the "receive-wait" primitive, synchronization is like that of a PLITS system.

Other types of synchronization are also possible. For instance, waiting upon issue of a "send" can come into the model when processes can close themselves to communication and "send-wait" is used. Note, however, that this wait is only until the destination process declares its queue open, <u>not</u> until the destination process <u>receives</u> the message. The "receive" primitive allows a process to attempt a message reception without risking the wait possibility inherent to "receive-wait". Brinch Hansen´s DP [4] synchronization can be attained by having the sending process "send", then immediately "receive-wait"; while the receiving process "receive-waits" and then immediately "sends". Hoare´s CSP [6] synchronization can be attained by having the sending process "send" and immediately "receive-wait"; while the receiving process would "receive" and immediately release the waiting sender by sending a reply.

Most of the usual SCHEME primitives [12] are implemented. Explanations of these primitives are in "The Revised Report on SCHEME, a Dialect of LISP". Specifically, the primitives "labels", "catch", "IF", "define", "evaluate", and "QUOTE" are available. Standard ILISP [1] primitives are available also, but primitives of this sort called directly from an expression evaluated by the CODA interpreter must be of type SUBR, LSUBR, or EXPR (ie. FSUBRS are unrecognized!).

3. Examples

The examples below serve to illustrate the use of the primitives described in section two and to show the various ways in which multiprocessing can be used in this system. Note that examples of both shared environment multiprocessing and distributed processing are given.

Example 1 -- Use of "labels"

The following example illustrates the use of the CODA interpreter for evaluating common SCHEME "labels" expressions [12]. No communication takes place; standard SCHEME is multiprocessed.

```
(DEFPROP LAB01
 (NIL labels
      ((PLUSS
        (LAMBDA(A B)
         (IF (ZEROP A) B (PLUSS (SUB1 A) (ADD1 B))))))
      (PRINT (PLUSS 5 7)))
VALUE)

(DEFPROP LAB02
 (NIL labels
      ((MEMBER2
        (LAMBDA(A L)
         (IF (NULL L)
             NIL
             (IF (F1 A (CAR L)) T (MEMBER2 A (CDR L))))))
       (F1 (LAMBDA (E1 E2) (EQ E1 E2))))
      (MEMBER2 1 (QUOTE (2 3 1 4 5))))
VALUE)

(DEFPROP LAB
 (NIL labels
      ((PLUX
        (LAMBDA(A B)
         (IF (ZEROP A) B (ADD1 (PLUX (SUB1 A) B))))))
      (PLUX 3 7))
VALUE)
```

```
(DEFPROP LAB2
 (NIL labels
      ((UNION
        (LAMBDA(A B)
         (IF (NULL A)
             B
             (IF (IN (CAR A) B)
                 (UNION (CDR A) B)
                 (CONS (CAR A) (UNION (CDR A) B))))))
       (IN
        (LAMBDA(A B)
         (IF (NULL B)
             NIL
             (IF (EQ (CAR B) A) T (IN A (CDR B)))))))
      (UNION (QUOTE (A B C D)) (QUOTE (A C E G))))
VALUE)

(DEFPROP LAB3
 (NIL labels
      ((FIB
        (LAMBDA(N)
         (IF (ZEROP N)
             1
             (IF (ZEROP (SUB1 N))
                 1
                 (PLUX (FIB (SUB1 N))
                       (FIB (SUB1 (SUB1 N))))))))
       (PLUX
        (LAMBDA(N M)
         (IF (ZEROP N) M (ADD1 (PLUX (SUB1 N) M))))))
      (FIB 6))
VALUE)

(DEFPROP CA1
 (NIL catch
      SURPRIZE
      (labels
       ((REMB
         (LAMBDA(L)
          (IF (EQ (CAR L) (QUOTE DOG))
              (REMB (CDR L))
              (IF (EQ (CAR L) (QUOTE HELP))
                  (SURPRIZE (QUOTE WOW))
                  (CONS (CAR L) (REMB (CDR L))))))))
       (REMB (QUOTE (WHEN DOG I DOG HELP ME)))))
VALUE)
```

```
(DEFPROP CA2
 (NIL catch
      SURPRIZE
      (labels
       ((REMB (LAMBDA (A L) (REMB2 A L)))
        (REMB2
         (LAMBDA(B Q)
          (IF (NULL Q)
              NIL
              (IF (EQ (CAR Q) B)
                  (REMB2 B (CDR Q))
                  (IF (EQ (CAR Q) (QUOTE HELP))
                      (SURPRIZE Q)
                      (CONS (CAR Q) (REMB2 B (CDR Q))))))))))

       (REMB (QUOTE DOG) (QUOTE (WHEN DOG HELP ME)))))
VALUE)

(DEFPROP CALL2
 (NIL EVALU8
      (LIST LAB01 NIL (QUOTE P01))
      (LIST LAB02 NIL (QUOTE P02))
      (LIST LAB NIL (QUOTE P1))
      (LIST LAB2 NIL (QUOTE P2))
      (LIST LAB3 NIL (QUOTE P3))
      (LIST CA1 NIL (QUOTE P4))
      (LIST CA2 NIL (QUOTE P5)))
VALUE)
```

```
*(EVAL CALL2)

PROCESS:
P1
terminated
PROCESS:
P02
terminated
12
PROCESS:
P01
terminated
PROCESS:
P5
terminated
PROCESS:
P4
terminated
PROCESS:
P2
terminated
PROCESS:
P3
terminated
((value-record P1 10)
 (value-record P02 T)
 (value-record P01 12)
 (value-record P5 (HELP ME))
 (value-record P4 WOW)
 (value-record P2 (B D A C E G))
 (value-record P3 13))
```

Example 2 -- Simple Communication

The following examples illustrate the use of the basic communication primitives. Process creation and activation is also shown; along with message manipulation. Two executions are shown to exhibit the equivalence of explicit process specification in the "EVALU8" expression and process creation and activation by another process. "PROCESS4" creates a message, updates it, and then sends it to "PROCESS3". "PROCESS3" waits until message reception is possible, then retrieves a value from the message and prints it. "PROCESS5" is used as a start-up process for "PROCESS3" and "PROCESS4".

```
(DEFPROP PROCESS3
 (NIL labels
      ((RW
        (LAMBDA NIL
         (receive-wait (QUOTE PROCESS4)
                       (QUOTE TRAN2)
                       (QUOTE RWM)
                       (QUOTE (CONT)))))
       (CONT
        (LAMBDA NIL
         ((LAMBDA (X) (PRINT SV))
          (aset (QUOTE SV) (slot-value (QUOTE A) RWM))))))

      (RW))
VALUE)
```

```
(DEFPROP PROCESS4
 (NIL labels
      ((S (LAMBDA NIL (send (QUOTE PROCESS3) MS)))
       (A
        (LAMBDA        NIL
         ((LAMBDA (X) (S))
          (slot-assign (QUOTE SV)
                       (ADD1 (slot-value (QUOTE SV) MS))
                       MS)))))
      ((LAMBDA (X) (A))
       (aset (QUOTE MS)
             (create-message (QUOTE ((A 1) (SV 0)))
                             (QUOTE TRAN2))))))
VALUE)

(DEFPROP PROCESS5
 (NIL (LAMBDA(X)
       ((LAMBDA(Y)
         ((LAMBDA (Z) (activate (QUOTE PROCESS4)))
          (activate (QUOTE PROCESS3))))
        (create-process PROCESS4 NIL (QUOTE PROCESS4))))
      (create-process PROCESS3 NIL (QUOTE PROCESS3)))
VALUE)
```

```
*(EVALU8 (LIST PROCESS3 NIL @PROCESS3)
         (LIST PROCESS4 NIL @PROCESS4))

PROCESS:
PROCESS4
terminated
1
1
PROCESS:
PROCESS3
terminated
((value-record PROCESS4 (queue))
 (value-record PROCESS3 1))


*(EVALU8 (LIST PROCESS5 NIL @PROCESS5))

PROCESS:
PROCESS5
terminated
PROCESS:
PROCESS4
terminated
1
1
PROCESS:
PROCESS3
terminated
((value-record PROCESS5 PROCESS4)
 (value-record PROCESS4 (queue))
 (value-record PROCESS3 1))
```

Example 3 -- Process Manipulation

The following examples demonstrate the use of the "receive-wait" and "aset" primitives, along with several process manipulation primitives ("deactivate" and "activate") and predicates. Notice that activating the processes in different orders affects the internal path of evaluation of the two processes (1 and 2). "PROCESS2A" is a start-up process for "PROCESS2". When it is used to create "PROCESS2", "PROCESS1" deactivates itself until "PROCESS2" is created and can reactivate it. The different values for the message-slot "A" show the differing paths of execution between the samples.

```
(DEFPROP PROCESS1
 (NIL labels
      ((CONT
        (LAMBDA NIL
         (IF (presentp (QUOTE A) RWM)
             (slot-value (QUOTE A) RWM)
             (suicide))))
       (CLOOP
        (LAMBDA NIL
         (IF (extantp (QUOTE PROCESS2))
             (receive-wait (QUOTE PROCESS2)
                           (QUOTE TRAN1)
                           (QUOTE RWM)
                           (QUOTE (CONT)))
             (deactivate (QUOTE (CLOOP)))))))
      (CLOOP))
VALUE)

(DEFPROP PROCESS2A
 (NIL (LAMBDA (X) (activate (QUOTE PROCESS2)))
      (create-process PROCESS2 NIL (QUOTE PROCESS2)))
VALUE)
```

```
(DEFPROP PROCESS2
 (NIL IF
      (extantp (QUOTE PROCESS1))
      (IF (openp (QUOTE PROCESS1))
          (send (QUOTE PROCESS1)
                (create-message (QUOTE ((B 7) (A 8) (C 9)))

                               (QUOTE TRAN1)))
          (suicide))
      ((LAMBDA(X)
        (send (QUOTE PROCESS1)
              (create-message (QUOTE ((B 1) (A 2) (C 3)))
                              (QUOTE TRAN1))))
       (activate (QUOTE PROCESS1))))
VALUE)
```

```
*(EVALU8 (LIST PROCESS1 NIL @PROCESS1)
         (LIST PROCESS2A NIL @PROCESS2A))

PROCESS:
PROCESS2A
terminated
PROCESS:
PROCESS2
terminated
PROCESS:
PROCESS1
terminated
((value-record PROCESS2A PROCESS2)
 (value-record PROCESS2 (queue))
 (value-record PROCESS1 2))


*(EVALU8 (LIST PROCESS1 NIL @PROCESS1)
         (LIST PROCESS2 NIL @PROCESS2))

PROCESS:
PROCESS2
terminated
PROCESS:
PROCESS1
terminated
((value-record PROCESS2 (queue))
 (value-record PROCESS1 8))


*(EVALU8 (LIST PROCESS2 NIL @PROCESS2)
         (LIST PROCESS1 NIL @PROCESS1))

PROCESS:
PROCESS2
terminated
PROCESS:
PROCESS1
terminated
((value-record PROCESS2 (queue))
 (value-record PROCESS1 8))
```

Example 4 -- Synchronization

These sample executions serve to further illustrate the use of communication primitives for synchronization. Note that in these two examples, the order of specification influences the value to which "PROCESS6" converges. "PROCESS7" floods the queue of "PROCESS6" with useless messages until a certain counter value is reached. At this point, "PROCESS7" sends a meaningful message to "PROCESS6". Once alerted, "PROCESS6" clears its queue of the useless messages and prepares to receive another meaningful message. The value of message-slot "B" indicates that the correct message was, in fact, recognized and used.

```
(DEFPROP PROCESS6
 (NIL labels
      ((RWG
        (LAMBDA NIL
         (receive-wait (QUOTE PROCESS7)
                       (QUOTE TRANG)
                       (QUOTE RWM)
                       (QUOTE (CONTG)))))
       (CONTG
        (LAMBDA NIL
         ((LAMBDA(X)
           ((LAMBDA (Y) (RWH))
            (send (QUOTE PROCESS7)
                  (create-message NIL (QUOTE TR)))))
          (clearq))))
       (RWH
        (LAMBDA NIL
         (receive-wait (QUOTE PROCESS7)
                       (QUOTE TRANH)
                       (QUOTE RWM)
                       (QUOTE (CONTH)))))
       (CONTH (LAMBDA NIL (slot-value (QUOTE B) RWM))))
      (RWG))
VALUE)
```

```
(DEFPROP PROCESS7
 (NIL labels
      ((S1 (LAMBDA NIL (send (QUOTE PROCESS6) MG)))
       (S2 (LAMBDA NIL (send (QUOTE PROCESS6) MH)))
       (MW
        (LAMBDA NIL
         (receive-wait (QUOTE PROCESS6)
                       NIL
                       (QUOTE RWM)
                       (QUOTE (S2)))))
       (INCR
        (LAMBDA NIL
         ((LAMBDA (X) (aset (QUOTE LC) (ADD1 LC)))
          (slot-assign (QUOTE B)
                       (ADD1 (slot-value (QUOTE B) MH))
                       MH))))
       (INI
        (LAMBDA NIL
         ((LAMBDA(X)
           ((LAMBDA(Y)
             (aset (QUOTE MG)
                   (create-message NIL (QUOTE TRANG))))
            (aset (QUOTE LC) 1)))
          (aset
           (QUOTE MH)
           (create-message (QUOTE ((A T) (B 0)))
                           (QUOTE TRANH))))))
       (LOOP
        (LAMBDA NIL
         (IF (EQ LC 5)
             ((LAMBDA (X) (MW)) (S1))
             ((LAMBDA (X) ((LAMBDA (Y) (LOOP)) (INCR)))
              (S2))))))
      ((LAMBDA (X) (LOOP)) (INI)))
VALUE)
```

```
*(EVALU8 (LIST PROCESS6 NIL @PROCESS6)
         (LIST PROCESS7 NIL @PROCESS7))

PROCESS:
PROCESS7
terminated
PROCESS:
PROCESS6
terminated
((value-record PROCESS7 (queue))
 (value-record PROCESS6 4))


*(EVALU8 (LIST PROCESS7 NIL @PROCESS7)
         (LIST PROCESS6 NIL @PROCESS6))

PROCESS:
PROCESS7
terminated
PROCESS:
PROCESS6
terminated
((value-record PROCESS7 (queue))
 (value-record PROCESS6 8))
```

Example 5 -- Shared Environment

In this example, several instances of the same processes are specified in the call to "EVALU8". They are all given a pointer to the same environment in order that the effects of several processes' accessing the same environment can be demonstrated. This competition in environment accessing is signalled by the appearance of 1's in the output. According to the internal tests of these processes, a "1" should never be printed, but the updating of "A" by other processes between the test and print expressions allow a "1" to be output.

```
(DEFPROP MLAB1
 (NIL labels
      ((F1
        (LAMBDA NIL
         ((LAMBDA (X) ((LAMBDA (Y) (F11 7)) (PRINT A)))
          (aset (QUOTE A) 0))))
       (F11
        (LAMBDA(N)
         (IF (ZEROP N)
             (QUOTE DONE)
             ((LAMBDA (X) (F11 (SUB1 N)))
              (aset (QUOTE A) 1))))))
      (F1))
VALUE)
```

```
(DEFPROP MLAB2
 (MLAB2
  labels
  ((F1
    (LAMBDA NIL
     ((LAMBDA (X) ((LAMBDA (Y) (F11 7)) (PRINT A)))
      (aset (QUOTE A) 0))))
   (F11
    (LAMBDA(N)
     (IF (ZEROP N)
      (F1)
      ((LAMBDA (X) (F11 (SUB1 N)))
       (aset (QUOTE A) 1))))))
  (F1))
VALUE)

(DEFPROP $E
 (NIL (A 7))
VALUE)

(DEFPROP CALL1
 (NIL EVALU8
      (LIST MLAB2 $E (QUOTE P1))
      (LIST MLAB1 $E (QUOTE P2))
      (LIST MLAB2 $E (QUOTE P3))
      (LIST MLAB2 $E (QUOTE P4))
      (LIST MLAB1 $E (QUOTE P5)))
VALUE)
```

```
*(EVAL CALL1)

0
0
0
0
0
PROCESS:
P5
terminated
PROCESS:
P2
terminated
0
0
1
0
0
0
0
0
1
0
0
0
0
1
0
0
1
1
0
1
0
^C
^C
```

Example 6 -- Dining Philosophers

A solution to the dining philosophers problem is given on the following page along with sample executions. The first process definition is that of the process that creates and activates the fork-monitor and philosophers. It was included in order to illustrate the use of the "create-process" and "activate" primitives. Sequential execution of expressions is accomplished through the use of embedded lambda-expressions.

The fork-monitor keeps a list that tells of the availability of each fork. For example, philosopher three requires forks two and three to eat, so if he were the only philosopher eating, the list would look like this: (T NIL NIL T T). Obviously, the key is to allow only the fork-monitor to update this list in order to avoid timing errors. It should be noted that while this solution prevents a second philosopher from doing anything but putting forks down or waiting while a first is waiting for one or both of his requested forks, it also prevents that second philosopher from using a fork that the first philosopher might not be using until he can find one more, as in [9].

The fork-monitor process begins by executing initialization code and issuing a receive-wait for any message. After a message is received, the continuation expression is specified to be the function that decodes the message to determine whether the philosopher that sent the message is trying to pick-up or put-down forks. Once a message requesting forks has been received, only messages from philosophers desiring to put forks down are received until the request can be granted. Once this type of "eat" request is satisfied, any message can be accepted. (Note that the list of pairs, PN-PAIRS is used simply to match a philosopher process with positions on the fork-condition list; PHILS is to identify a process-name with a position number on the fork-condition list.)

The philosophers are five instances of the philosopher process definition. Each philosopher attempts to eat, then think, then eat, ... etc. until the system is stopped. The order of the philosophers' eating and thinking in the following sample executions is determined by a combination of the order of activation and the random selection of processes to be multiprocessed by the interpreter.

```
(DEFPROP start-dinner2
 (NIL
  labels
  ((C1
    (LAMBDA NIL
     (create-process fork-monitor NIL (QUOTE FM))))
   (C2
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL1))))
   (C3
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL2))))
   (C4
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL3))))
   (C5
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL4))))
   (C6
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL5))))
   (A1 (LAMBDA NIL (activate (QUOTE FM))))
   (A2 (LAMBDA NIL (activate (QUOTE PHIL1))))
   (A3 (LAMBDA NIL (activate (QUOTE PHIL2))))
   (A4 (LAMBDA NIL (activate (QUOTE PHIL3))))
   (A5 (LAMBDA NIL (activate (QUOTE PHIL4))))
   (A6 (LAMBDA NIL (activate (QUOTE PHIL5)))))
  ((LAMBDA(A)
    ((LAMBDA(B)
      ((LAMBDA(C)
        ((LAMBDA(D)
          ((LAMBDA(E)
            ((LAMBDA(F)
              ((LAMBDA(G)
                ((LAMBDA(H)
                  ((LAMBDA(I)
                    ((LAMBDA(J)
                      ((LAMBDA(K)
                        ((LAMBDA (L) (suicide)) (A6)))
                       (A4)))
                     (A2)))
                   (A5)))
                 (A3)))
               (A1)))
             (C6)))
           (C5)))
         (C4)))
       (C3)))
     (C2)))
   (C1)))
VALUE)
```

```
(DEFPROP start-dinnerl
 (NIL
  labels
  ((C1
    (LAMBDA NIL
     (create-process fork-monitor NIL (QUOTE FM))))
   (C2
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL1))))
   (C3
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL2))))
   (C4
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL3))))
   (C5
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL4))))
   (C6
    (LAMBDA NIL
     (create-process philosopher NIL (QUOTE PHIL5))))
   (A1 (LAMBDA NIL (activate (QUOTE FM))))
   (A2 (LAMBDA NIL (activate (QUOTE PHIL1))))
   (A3 (LAMBDA NIL (activate (QUOTE PHIL2))))
   (A4 (LAMBDA NIL (activate (QUOTE PHIL3))))
   (A5 (LAMBDA NIL (activate (QUOTE PHIL4))))
   (A6 (LAMBDA NIL (activate (QUOTE PHIL5)))))
  ((LAMBDA(A)
    ((LAMBDA(B)
      ((LAMBDA(C)
        ((LAMBDA(D)
          ((LAMBDA(E)
            ((LAMBDA(F)
              ((LAMBDA(G)
                ((LAMBDA(H)
                  ((LAMBDA(I)
                    ((LAMBDA(J)
                      ((LAMBDA(K)
                        ((LAMBDA (L) (suicide)) (A6)))
                       (A5)))
                     (A4)))
                   (A3)))
                 (A2)))
               (A1)))
             (C6)))
           (C5)))
         (C4)))
       (C3)))
     (C2)))
   (C1)))
VALUE)
```

```
(DEFPROP fork-monitor
 (NIL
  labels
  ((pickup
    (LAMBDA(I)
     (IF
      (ANDD (fork-cond (MOD5 (SUB1 I))) (fork-cond I))
      (give-forks I)
      (ready-wait I))))
   (ready-wait
    (LAMBDA(I)
     (IF
      (fork-cond (MOD5 (SUB1 I)))
      (receive-wait
       (phil (MOD5 (ADD1 I)))
       (QUOTE THINK)
       (QUOTE RWM)
       (QUOTE
        ((LAMBDA(X)
          (pickup
           (MOD5
            (SUB1 (VALUE (slot-value (QUOTE FROM) RWM))))))
         (putdown (VALUE (slot-value (QUOTE FROM) RWM))))))
      (receive-wait
       (phil (MOD5 (SUB1 I)))
       (QUOTE THINK)
       (QUOTE RWM)
       (QUOTE
        ((LAMBDA(X)
          (pickup
           (MOD5
            (ADD1 (VALUE (slot-value (QUOTE FROM) RWM))))))
         (putdown
          (VALUE (slot-value (QUOTE FROM) RWM)))))))))
   (give-forks
    (LAMBDA(I)
     ((LAMBDA(X)
       ((LAMBDA(Y)
         ((LAMBDA(Z)
           (receive-wait NIL NIL (QUOTE RWM) (QUOTE (decode))))
          (send (phil I) (create-message NIL (QUOTE EAT)))))
        (set-fork-cond I NIL)))
      (set-fork-cond (MOD5 (SUB1 I)) NIL))))
   (putdown
    (LAMBDA(I)
     ((LAMBDA(X)
       (send (phil I) (create-message NIL (QUOTE THINK))))
      ((LAMBDA (Z) (set-fork-cond I T))
       (set-fork-cond (MOD5 (SUB1 I)) T)))))
    (decode
     (LAMBDA NIL
      (IF
       (EQ (QUOTE EAT) (slot-value (QUOTE ABOUT) RWM))
       (pickup (VALUE (slot-value (QUOTE FROM) RWM)))
       ((LAMBDA (X) (receive-wait NIL
                                  NIL
                                  (QUOTE RWM)
                                  (QUOTE (decode))))
        (putdown (VALUE (slot-value (QUOTE FROM) RWM)))))))
    (MOD5
     (LAMBDA (N) (IF (ZEROP N) 5 (IF (EQ N 6) 1 N))))
    (ANDD (LAMBDA (E1 E2) (IF E1 E2 NIL)))
    (phil (LAMBDA (N) (nth PHILS N)))
    (fork-cond (LAMBDA (N) (nth FORKS N)))
    (set-fork-cond (LAMBDA (N V)
                    (RPLACA (NTH FORKS N) V)))
    (VALUE (LAMBDA (PN) (CADR (ASSOC PN PN-PAIRS))))
    (INI
     (LAMBDA NIL
      ((LAMBDA(X)
        ((LAMBDA(Y)
          (aset (QUOTE FORKS) (QUOTE (T T T T T))))
         (aset (QUOTE PHILS)
               (QUOTE (PHIL1 PHIL2 PHIL3 PHIL4 PHIL5)))))
       (aset
        (QUOTE PN-PAIRS)
        (QUOTE
         ((PHIL1 1) (PHIL2 2)
                    (PHIL3 3)
                    (PHIL4 4)
                    (PHIL5 5))))))))
   ((LAMBDA (X) (receive-wait NIL NIL (QUOTE RWM) (QUOTE (decode))))
    (INI)))
VALUE)
```

```
(DEFPROP philosopher
 (NIL labels
      ((seek-eat
        (LAMBDA NIL
         ((LAMBDA(X)
           (receive-wait
            (QUOTE FM)
            (QUOTE EAT)
            (QUOTE RWM)
            (QUOTE ((LAMBDA (Z) (seek-think)) (eat)))))
          (send (QUOTE FM)
                (create-message NIL (QUOTE EAT))))))
       (eat
        (LAMBDA NIL
         (print3 (QUOTE PHILOSOPHER:)
                 #pn#
                 (QUOTE :::EATING!!))))
       (print3
        (LAMBDA(A1 A2 A3)
         ((LAMBDA (X) ((LAMBDA (Y) (PRINC A3)) (PRINC A2)))

          (PRINT A1))))
       (seek-think
        (LAMBDA NIL
         ((LAMBDA(X)
           (receive-wait
            (QUOTE FM)
            (QUOTE THINK)
            (QUOTE RWM)
            (QUOTE ((LAMBDA (Z) (seek-eat)) (think)))))
          (send (QUOTE FM)
                (create-message NIL (QUOTE THINK))))))
       (think
        (LAMBDA NIL
         (print3 (QUOTE PHILOSOPHER:)
                 #pn#
                 (QUOTE :::THINKING:::)))))
      (seek-eat))
VALUE)
```

```
*(EVALU8 (LIST start-dinner1 NIL @SD))

PROCESS:
SD
SUICIDE!!!
PROCESS:
SD
terminated
PHILOSOPHER: PHIL1:::EATING!!
PHILOSOPHER: PHIL1:::THINKING:::
PHILOSOPHER: PHIL2:::EATING!!
PHILOSOPHER: PHIL2:::THINKING:::
PHILOSOPHER: PHIL3:::EATING!!
PHILOSOPHER: PHIL3:::THINKING:::
PHILOSOPHER: PHIL4:::EATING!!
PHILOSOPHER: PHIL4:::THINKING:::
PHILOSOPHER: PHIL5:::EATING!!
PHILOSOPHER: PHIL5:::THINKING:::
PHILOSOPHER: PHIL1:::EATING!!
PHILOSOPHER: PHIL1:::THINKING:::
PHILOSOPHER: PHIL2:::EATING!!
^C
^C


*(EVALU8 (LIST start-dinner2 NIL @SD))

PROCESS:
SD
SUICIDE!!!
PROCESS:
SD
terminated
PHILOSOPHER: PHIL2:::EATING!!
PHILOSOPHER: PHIL4:::EATING!!
PHILOSOPHER: PHIL2:::THINKING:::
PHILOSOPHER: PHIL1:::EATING!!
PHILOSOPHER: PHIL4:::THINKING:::
PHILOSOPHER: PHIL3:::EATING!!
PHILOSOPHER: PHIL1:::THINKING:::
PHILOSOPHER: PHIL5:::EATING!!
PHILOSOPHER: PHIL3:::THINKING:::
PHILOSOPHER: PHIL2:::EATING!!
PHILOSOPHER: PHIL5:::THINKING:::
PHILOSOPHER: PHIL4:::EATING!!
PHILOSOPHER: PHIL2:::THINKING:::
PHILOSOPHER: PHIL1:::EATING!!
PHILOSOPHER: PHIL4:::THINKING:::
PHILOSOPHER: PHIL3:::EATING!!
^C
^C
```

4. Conclusion

This multiprocessing system provides an elegant means of defining processes in either a shared environment or distributed processing model. Communication and synchronization can be modeled after any of several distributed processing paradigms. Various primitives are provided for process creation, activation, deactivation, and internal assignment; along with the standard SCHEME primitives. The interpreter stack is handled explicitly in the form of continuation links (c-links), and various new function properties are available.

Acknowledgement

I would like to acknowledge the aid given me on this project by Daniel Friedman of the Computer Science Department of Indiana University, under whose guidance this project was undertaken. His advice and direction in developing this system have proved invaluable. I would also like to acknowledge the aid of my wife Susan; without her patient proofreading of several drafts of this thesis, the present version would be much less readable.

Bibliography

1. Robert J. Bobrow, Richard R. Burton, Jeffrey M. Jacobs, Daryle Lewis. UCI LISP Manual. PDP-10 documentation.

2. Jerome A. Feldman. A Programming Methodology for Distributed Computing (among other things). Technical Report 9, Department of Computer Science, University of Rochester.

3. Jerome A. Feldman. High level programming for distributed computing. CACM 22, 6 (June, 1979), 353-367.

4. Per Brinch Hansen. Distributed Processes: a concurrent programming concept. CACM 21, 11 (November 1978), 934-941.

5. Per Brinch Hansen. RC 4000 Software Multiprogramming System. A/S Regnecentralen. (1969).

6. C. A. R. Hoare. Communicating sequential processes. CACM 21, 8 (August, 1978), 666-677.

7. C. A. R. Hoare. Parallel Programming: An Axiomatic Approach. Computer Languages, vol. 1. Pergamon Press, Belfast, 1975.

8. R. C. Holt, G. S. Graham, E. D. Lazowska, and M. A. Scott. Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley Publishing Company, Reading, 1978.

9. W. H. Kaubisch, R. H. Perrott, C. A. R. Hoare, Quasiparallel Programming. Software Practice and Experience, vol. 6, pp 341-356.

10. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin. LISP 1.5 Programmer's Manual, The M.I.T. Press, Cambridge, 1962.

11. Stuart C. Shapiro. Techniques of Artificial Intelligence, 142-143. D. Van Nostrand Company, New York, 1979.

12. Guy Steele and Gerald Sussman. The revised report on SCHEME, a dialect of LISP. AI Memo 452, (January, 1978), MIT.

13. Lynn H. Quam and Whitfield Diffie. LISP 1.6. Stanford Artificial Intelligence Project, Stanford, 1973.

Appendix A -- Implementation Notes

Message queues

Each process' queue is a linear list of all messages sent to that process since its queue was last cleared or since its creation and activation. Message reception entails retrieving the oldest satisfactory message in the queue.

Deactivated processes, newly created processes

These processes are stored as suspended processes in an alternate pool of processes awaiting activation. No process is guaranteed to be activated from out of this pool.

DATA structures

Several data structures are defined using DATA and DATADEF. These functions are used to define a data structure and automatically write functions for changing or retrieving the fields of the structure. [11]

INDIVISIBLE

This property name specifies that the function with this property is to be evaluated uninterruptedly. The interpreter, upon retrieving a function of this type, sets the value of a particular register #ind# to T. As long as this register is T, the function is evaluated. once the c-link is popped, #ind# resumes its normal value, NIL.

EVALU8

This initial call to the interpreter returns a value which is a sequence of the process name and termination value pairs for each process that was terminated during evaluation.

RANDOM

The interpreter requires that the additional package of arithmetic primitives be loaded into the LISP system so that RANDOM will be defined.

LISP

The interpreter is implemented in UCI LISP, and may not run in other versions of LISP. However, any changes that might be needed would be quite minor.

Appendix B -- CODA Interpreter Code

The top level call to the interpreter is through the function "EVALU8"; the driver is in the function "repeateval". The actual evaluating code is contained in the two functions "dlcode" and "dscode". Various registers are used throughout the interpreter code and the communication primitives code:

#exp#	the expression being evaluated
#env#	the environment used for evaluation
#evl#	that part of #exp# already processed
#unl#	that part of #unl# still to be processed
#pc#	the current interpreter code being used
#clink#	the explicit CODA stack
#ind#	indicates whether exp is INDIVISIBLE
#pn#	the name of the active process
#accumulator#	this register collects the result of evaluating the #exp# of the current c-link. It serves as a means of carrying information over from one c-link of the stack to the rest. When an expression is completely evaluated, this register contains the value.

```
(DEFPROP INTERPRETER
 (INTERPRETER ceval
              neval
              returnto
              return
              doeval
              dlcode
              doevlis
              dscode
              evalu8
              Xevalu8
              nevaleach
              nevalhelp
              repeateval
              restore-registers
              registers
              register-download
              register-upload
              terminate
              suspend
              queue-position
              QUOTE
              define
              LAMBDA
              IF
              evaluate
              catch
              labels
              aset
              aseth
              setrl
              cset)
VALUE)

(DEFPROP ceval
 (LAMBDA (X A) (:= #clink# (register-download)) (doeval X))
EXPR)

(DEFPROP neval
 (LAMBDA (X N) (:= #env# N) (doeval X))
EXPR)

(DEFPROP returnto
 (LAMBDA (V C) (:= #clink# C) (return V))
EXPR)
```

```
(DEFPROP return
 (LAMBDA(V)
  (:= #accumulator# V)
  (if #clink#
      (then (register-upload))
      (else
       (error (QUOTE PROCESS-RAN-OUT)
              #exp#
              (QUOTE FAIL-ACT)))))
EXPR)

(DEFPROP doeval
 (LAMBDA (X) (:= #exp# X) (:= #pc# (dlcode)))
EXPR)

(DEFPROP dlcode
 (LAMBDA NIL
  (QUOTE
   (COND ((atomp #exp#)
          (return
           (COND ((OR (constantp #exp#) (primop #exp#))
                  #exp#)
                 ((assoc #exp# #env#) (two assoc))
                 (T (symeval #exp#)))))
         ((get (one #exp#) (QUOTE QUICK))
          (return (EVAL get)))
         ((get (one #exp#) (QUOTE SLOW))
          (APPLY (QUOTE ceval) (EVAL get)))
         ((get (one #exp#) (QUOTE MOVING))
          (APPLY (QUOTE neval) (EVAL get)))
         ((get (one #exp#) (QUOTE MACRO))
          (:= #exp# (APPLY get (LIST #exp#))))
         ((get (one #exp#) (QUOTE INDIVISIBLE))
          (:= #ind# T)
          (APPLY (QUOTE doevlis)
                 (LIST (LIST get) (CDR #exp#))))
         (T
          (APPLY
           (QUOTE doevlis)
           (if (AND (NOT (atomp (CAR #exp#)))
                    (same (one (CAR #exp#)) (QUOTE LAMBDA)))
               (then (LIST (LIST (CAR #exp#)) (CDR #exp#)))
               (else (LIST (LIST) #exp#))))))))
EXPR)

(DEFPROP doevlis
 (LAMBDA (E U) (:= #evl# E) (:= #unl# U) (:= #pc# (dscode)))
EXPR)
```

```
(DEFPROP dscode
 (LAMBDA NIL
  (QUOTE
   (COND (#unl#
          (ceval
           (CAR #unl#)
           (QUOTE
            (doevlis (snoc #evl# #accumulator#)
                     (CDR #unl#)))))
         ((atomp (CAR #evl#))
          (return (APPLY# (CAR #evl#) (CDR #evl#))))
         ((same (one (CAR #evl#)) (QUOTE LAMBDA))
          (neval
           (three (CAR #evl#))
           (bindup (two (CAR #evl#)) (CDR #evl#) #env#)))
         ((same (one (CAR #evl#)) (QUOTE BETA))
          (neval
           (three (two (CAR #evl#)))
           (bindup (two (two (CAR #evl#)))
                   (CDR #evl#)
                   (one (three (CAR #evl#))))))
         ((same (one (CAR #evl#)) (QUOTE DELTA))
          (returnto (two #evl#) (two (CAR #evl#))))
         (T
          (error (QUOTE BAD-FUNCTION-EVARGLIST)
                 #exp#
                 (QUOTE FAIL-ACT))))))
EXPR)

(DEFPROP evalu8
 (LAMBDA (PL) (Xevalu8 (MAPCAR (QUOTE EVAL) PL)))
FEXPR)

(DEFPROP Xevalu8
 (LAMBDA(LOP)
  (if (NULL LOP)
      (then NIL)
      (else (:= #process-queue-seq#
                (CONS NIL (makepqs (threes LOP))))
            (:= #deactivated-pn-seq# (LIST NIL))
            (:= #running-pn-seq# (CONS NIL (threes LOP)))
            (:= #closed-seq# (LIST NIL))
            (:= #not-activated-process-seq# (LIST NIL))
            (:= #numprocs# (LENGTH LOP))
            (repeateval (nevaleach LOP)))))
EXPR)

(DEFPROP nevaleach
 (LAMBDA (LOP) (MAPCAR (QUOTE nevalhelp) LOP))
EXPR)
```

```
(DEFPROP nevalhelp
 (LAMBDA(LP)
  (neval (one LP) (two LP))
  (:= #pn# (three LP))
  (suspended-process (LIST)
                     NIL
                     #exp#
                     #env#
                     (LIST)
                     (LIST)
                     #pc#
                     (anchor)
                     #pn#))
EXPR)

(DEFPROP repeateval
 (LAMBDA(pp)
  (PROG (valueseq processpool)
        (:= processpool (CONS NIL pp))
        (:= valueseq (LIST NIL))
   looplabel:
        (if (drainedp processpool)
            (then (RETURN (CDR valueseq))))
        (:= #running-process# (random-integer #numprocs#))
        (restore-registers #running-process#)
   indivisiblelabel:
        (execute #pc#)
        (if #ind# (then (GO indivisiblelabel:)))
        (if (finished)
            (then (terminate #running-process#))
            (else (suspend #running-process#)))
        (GO looplabel:)))
EXPR)

(DEFPROP restore-registers
 (LAMBDA(RN)
  (PROG (suspension)
        (:= suspension (nth (CDR processpool) RN))
        (:= #accumulator# (&accumulator suspension))
        (:= #ind# (&ind suspension))
        (:= #exp# (&exp suspension))
        (:= #env# (&env suspension))
        (:= #evl# (&evl suspension))
        (:= #unl# (&unl suspension))
        (:= #pc# (&pc suspension))
        (:= #clink# (&clink suspension))
        (:= #pn# (&pn suspension))))
EXPR)
```

```
(DEFPROP registers
 (LAMBDA NIL
  (suspended-process #accumulator#
                     #ind#
                     #exp#
                     #env#
                     #evl#
                     #unl#
                     #pc#
                     #clink#
                     #pn#))
EXPR)

(DEFPROP register-download
 (LAMBDA NIL
  (clink #ind# #exp# #env# #evl# #unl# A #clink# #pn#))
EXPR)

(DEFPROP register-upload
 (LAMBDA NIL
  (:= #ind# ($ind #clink#))
  (:= #exp# ($exp #clink#))
  (:= #env# ($env #clink#))
  (:= #evl# ($evl #clink#))
  (:= #unl# ($unl #clink#))
  (:= #pc# ($pc #clink#))
  (:= #pn# ($pn #clink#))
  (:= #clink# ($clink #clink#)))
EXPR)

(DEFPROP terminate
 (LAMBDA(N)
  (if (NOT (deactivatedp #pn#))
      (then (addvalue valueseq
                      (value-record #pn# #accumulator#))
            (print (QUOTE PROCESS:)
                   #pn#
                   (QUOTE terminated))))
  (remove-nth N processpool)
  (remove-nth (queue-position #pn#
                              (CDR #process-queue-seq#))
              #process-queue-seq#)
  (DREMOVE #pn# #running-pn-seq#)
  (:= #numprocs# (SUB1 #numprocs#)))
EXPR)

(DEFPROP suspend
 (LAMBDA(N)
  (replace (nth (CDR processpool) N) (registers)))
EXPR)
```

```
(DEFPROP queue-position
 (LAMBDA(PN PQS)
  (PROG (POSN TPQS)
        (:= POSN 1)
        (:= TPQS PQS)
   qplabel:
        (if (NULL TPQS) (RETURN 0))
        (if (same PN (process-name (CAR TPQS)))
            (then (RETURN POSN))
            (else (:= POSN (ADD1 POSN))
                  (:= TPQS (CDR TPQS))
                  (GO qplabel:)))))
EXPR)

(DEFPROP QUOTE
 (two #exp#)
QUICK)

(DEFPROP define
 (setr1 (two #exp#)
        (LIST (QUOTE BETA) (three #exp#) (LIST)))
QUICK)

(DEFPROP LAMBDA
 (LIST (QUOTE BETA) #exp# #env#)
QUICK)

(DEFPROP IF
 (LIST (two #exp#)
       (QUOTE
        (doeval
         (COND (#accumulator# (three #exp#))
               (T (four #exp#))))))
SLOW)

(DEFPROP evaluate
 (LIST (two #exp#) (QUOTE (doeval #accumulator#)))
SLOW)

(DEFPROP catch
 (LIST (three #exp#)
       (CONS (LIST (two #exp#) (LIST (QUOTE DELTA) #clink#))
             #env#))
MOVING)
```

```
(DEFPROP labels
 (LIST
  (three #exp#)
  ((LAMBDA(Y)
    ((LAMBDA(Z)
      ((LAMBDA (B) (NCONC (replace Y B) #env#))
       (MAPCAR
        (QUOTE
         (LAMBDA(D)
          (pair (one D) (LIST (QUOTE BETA) (two D) Z))))
        (two #exp#))))
     (LIST Y)))
   (LIST NIL)))
MOVING)

(DEFPROP aset
 (LAMBDA (TVN TVAL) (aseth TVN TVAL #env#))
INDIVISIBLE)

(DEFPROP aseth
 (LAMBDA(VN VAL ENV)
  ((LAMBDA(PAIR)
    ((LAMBDA (Z) VAL)
     (COND (PAIR (RPLACA (CDR PAIR) VAL))
           (ENV (NCONC ENV (LIST (LIST VN VAL))))
           (T (:= #env# (LIST (LIST VN VAL)))))))
   (ASSOC VN ENV)))
EXPR)

(DEFPROP setrl
 (LAMBDA (VAR VAL) (aset VAR VAL) VAR)
INDIVISIBLE)

(DEFPROP cset
 (LAMBDA (VN VAL) (:= #env# (CONS (LIST VN VAL) #env#)))
INDIVISIBLE)
```

Appendix C -- Communication Primitives Code

```
(DEFPROP COMMLIST
 (NIL create-message
      slot-assign
      slot-value
      NO-SLOT-MES
      presentp
      &process-queue-record
      NO-PQR-MES
      extantp
      openp
      closedp
      closedpexpr
      close-communications
      open-communications
      suicide
      suicide-EXPR-layer
      clearq
      descriptor
      receive-wait
      simple-dowmload
      wcsetup
      rwait-closure
      receive
      receiveh
      dmatchp
      send
      send-wait
      swait-closure
      create-process
      anchor2
      activate
      activateh
      deactivate
      dcsetup
      termination-value
      value)
VALUE)

(DEFPROP create-message
 (LAMBDA(SLOT-LIST TRAN)
  (NCONC (LIST (LIST (QUOTE FROM) #pn#)
               (LIST (QUOTE ABOUT) TRAN))
         SLOT-LIST))
EXPR)

(DEFPROP slot-assign
 (LAMBDA (SN VAL MES) (aseth SN VAL MES))
INDIVISIBLE)
```

```
(DEFPROP slot-value
 (LAMBDA (SN MES) (two (SASSOC SN MES (QUOTE NO-SLOT-MES))))
INDIVISIBLE)

(DEFPROP NO-SLOT-MES
 (LAMBDA NIL (QUOTE slot-not-found))
EXPR)

(DEFPROP presentp
 (LAMBDA (SN MES) (ASSOC SN MES))
INDIVISIBLE)

(DEFPROP &process-queue-record
 (LAMBDA (PN PQS) (SASSOC PN PQS (QUOTE NO-PQR-MES)))
EXPR)

(DEFPROP NO-PQR-MES
 (LAMBDA NIL (QUOTE no-process-queue-record-found))
EXPR)

(DEFPROP extantp
 (LAMBDA (PN) (MEMBER PN #running-pn-seq#))
INDIVISIBLE)

(DEFPROP openp
 (LAMBDA (PN) (NOT (MEMBER PN #closed-seq#)))
INDIVISIBLE)

(DEFPROP closedp
 (LAMBDA (PN) (MEMBER PN #closed-seq#))
INDIVISIBLE)

(DEFPROP closedpexpr
 (LAMBDA (PN) (MEMBER PN #closed-seq#))
EXPR)

(DEFPROP close-communications
 (LAMBDA NIL (NCONC #closed-seq# (LIST #pn#)))
INDIVISIBLE)

(DEFPROP open-communications
 (LAMBDA NIL (DREMOVE #pn# #closed-seq#))
INDIVISIBLE)

(DEFPROP suicide
 (LAMBDA NIL (suicide-EXPR-layer))
INDIVISIBLE)

(DEFPROP suicide-EXPR-layer
 (LAMBDA NIL
  ($ind #clink# NIL)
  ($pc #clink# NIL)
  (print (QUOTE PROCESS:) #pn# (QUOTE SUICIDE!!!))
  (LIST (QUOTE SUICIDE) #accumulator#))
EXPR)
```

```
(DEFPROP clearq
 (LAMBDA NIL
  (aseth #pn# (QUOTE (queue)) (CDR #process-queue-seq#)))
INDIVISIBLE)

(DEFPROP descriptor
 (LAMBDA (QR) (message-description (about QR) (from QR)))
EXPR)

(DEFPROP receive-wait
 (LAMBDA(F A MID continuation)
  ((LAMBDA(R)
    (if R
        (then (aseth MID R #env#) (wcsetup continuation))
        (else (:= #pc# (rwait-closure))
              (:= #clink# (simple-download)))))
   (receive F A)))
EXPR)

(DEFPROP wcsetup
 (LAMBDA(C)
  (neval C #env#)
  (:= #ind# NIL)
  (:= #evl# (LIST))
  (:= #unl# (LIST))
  (:= #clink# (simple-download))
  (:= #accumulator# (LIST)))
EXPR)

(DEFPROP rwait-closure
 (LAMBDA NIL
  (LIST (QUOTE
         (LAMBDA (FR AB MI CO) (receive-wait FR AB MI CO)))
        (LIST (QUOTE QUOTE) F)
        (LIST (QUOTE QUOTE) A)
        (LIST (QUOTE QUOTE) MID)
        (LIST (QUOTE QUOTE) continuation)))
EXPR)

(DEFPROP receive
 (LAMBDA(F A)
  (receiveh F
            A
            (queue
             (&process-queue-record
              #pn#
              #process-queue-seq#))))
EXPR)
```

```
(DEFPROP receiveh
 (LAMBDA(F A Q)
  (COND ((NULL (CDR Q)) NIL)
        ((dmatchp (message-description F A)
                  (description (CADR Q)))
         ((LAMBDA (MSG) (RPLACD Q (CDDR Q)) MSG)
          (message (CADR Q))))
        (T (receiveh F A (CDR Q)))))
EXPR)

(DEFPROP dmatchp
 (LAMBDA(REQD ACTD)
  (OR (AND (wildp (source REQD)) (wildp (transaction REQD)))
      (AND (wildp (source REQD))
           (same (transaction REQD) (transaction ACTD)))
      (AND (same (source REQD) (source ACTD))
           (wildp (transaction REQD)))
      (AND (same (source REQD) (source ACTD))
           (same (transaction REQD) (transaction ACTD)))))
EXPR)

(DEFPROP send
 (LAMBDA(DEST MES)
  (if (NOT (closedpexpr DEST))
      (then
       (NCONC (queue
               (&process-queue-record
                DEST
                #process-queue-seq#))
              (LIST
               (message-record
                (message-description (from MES) (about MES))
                MES))))
      (else NIL)))
EXPR)

(DEFPROP send-wait
 (LAMBDA(DEST MSG continuation)
  (if (send DEST MSG)
      (then (wcsetup continuation))
      (else (:= #pc# (swait-closure))
            (:= #clink# (simple-download)))))
EXPR)

(DEFPROP swait-closure
 (LAMBDA NIL
  (LIST (QUOTE (LAMBDA (DS MES CON) (send-wait DS MES CON)))
        (LIST (QUOTE QUOTE) DEST)
        (LIST (QUOTE QUOTE) MSG)
        (LIST (QUOTE QUOTE) continuation)))
EXPR)
```

```
(DEFPROP create-process
 (LAMBDA(DEF ENV PN)
  (NCONC #not-activated-process-seq#
         (LIST
          (suspended-process (LIST)
                             NIL
                             DEF
                             ENV
                             (LIST)
                             (LIST)
                             (dlcode)
                             (anchor2)
                             PN))))
EXPR)

(DEFPROP anchor2
 (LAMBDA NIL
  (clink NIL (LIST) (LIST) (LIST) (LIST) (LIST) (LIST) PN))
EXPR)

(DEFPROP activate
 (LAMBDA (PN) (activateh PN #not-activated-process-seq#))
EXPR)

(DEFPROP activateh
 (LAMBDA(PN NAPS)
  (COND ((NULL (CDR NAPS)) NIL)
        ((EQ PN (&pn (CADR NAPS)))
         (NCONC processpool (LIST (CADR NAPS)))
         (NCONC #process-queue-seq#
                (LIST (LIST PN (LIST (QUOTE queue)))))
         (:= #numprocs# (ADD1 #numprocs#))
         (DREMOVE PN #deactivated-pn-seq#)
         (NCONC #running-pn-seq# (LIST PN))
         (RPLACD NAPS (CDDR NAPS))
         PN)
        (T (activateh PN (CDR NAPS)))))
EXPR)

(DEFPROP deactivate
 (LAMBDA(continuation)
  (dcsetup continuation)
  (NCONC #not-activated-process-seq# (LIST (registers)))
  (NCONC #deactivated-pn-seq# (LIST #pn#))
  #pn#)
EXPR)
```

```
(DEFPROP dcsetup
 (LAMBDA(C)
  (neval C #env#)
  (:= #ind# NIL)
  (:= #evl# (LIST))
  (:= #unl# (LIST))
  (:= #clink# (anchor))
  (:= #accumulator# (LIST)))
EXPR)

(DEFPROP termination-value
 (LAMBDA(PN)
  (if (childprocessp PN)
      (then (value PN #child-value-seq#))
      (else (value PN valueseq))))
EXPR)

(DEFPROP value
 (LAMBDA(PN VS)
  (COND ((NULL VS) (QUOTE no-value))
        ((same PN (%pn (CAR VS))) (%value (CAR VS)))
        (T (value PN (CDR VS)))))
EXPR)
```

Appendix D -- Help Functions

```
(DEFPROP HELPLIST
 (HELPLIST if
           nth
           print
           replace
           finished
           expand
           position
           randigit
           constantp
           primop
           bindup
           pairlis
           anchor
           DATA
           DATADEF
           assoc
           get
           makepqs
           remove-nth
           addvalue
           deactivatedp
           childprocessp
           random-integer)
VALUE)

(DEFPROP if
 (LAMBDA(IFEXP)
  (COND ((NULL (cdr3 IFEXP))
         (RPLACA
          (RPLACD
           IFEXP
           (LIST
            (RPLACA (consequent IFEXP) (predicate IFEXP))))
          (QUOTE COND)))
        (T
         (RPLACA
          (RPLACD
           IFEXP
           (LIST (RPLACA (consequent IFEXP)
                         (predicate IFEXP))
                 (RPLACA (alternative IFEXP) (QUOTE T))))
          (QUOTE COND)))))
MACRO)

(DEFPROP nth
 (LAMBDA(L N) (CAR (NTH L N)))
EXPR)
```

```
(DEFPROP print
 (LAMBDA(PL)
  (LIST (QUOTE MAPC)
        (QUOTE (QUOTE PRINT))
        (LIST (QUOTE MAPCAR)
              (QUOTE (QUOTE EVAL))
              (LIST (QUOTE QUOTE) (CDR PL)))))
MACRO)

(DEFPROP replace
 (LAMBDA (O N) (RPLACA (RPLACD O (CDR N)) (CAR N)))
EXPR)

(DEFPROP finished
 (LAMBDA (Fl) (replace Fl (QUOTE (NULL #pc#))))
MACRO)

(DEFPROP expand
 (LAMBDA(L FN)
  (if (NULL (CDR L))
      (then (CAR L))
      (else (LIST FN (CAR L) (expand (CDR L) FN)))))
EXPR)

(DEFPROP position
 (LAMBDA(A L)
  (PROG (POSN TL)
        (:= POSN 1)
        (:= TL L)
   poslabel:
        (if (NULL TL) (RETURN 0))
        (if (same A (CAR TL))
            (then (RETURN POSN))
            (else (:= TL (CDR TL))
                  (:= POSN (ADD1 POSN))
                  (GO poslabel:)))))
EXPR)

(DEFPROP randigit
 (LAMBDA NIL (FIX (TIMES 10 (RANDOM))))
EXPR)

(DEFPROP constantp
 (LAMBDA(A)
  (OR (NUMBERP A) (NOT A) (same A (QUOTE T))))
EXPR)

(DEFPROP primop
 (LAMBDA (A)
  (GETL A (QUOTE (SUBR EXPR LSUBR))))
EXPR)
```

```
(DEFPROP bindup
 (LAMBDA(VARL VALL AL)
  (if (AND (atomp VARL) (NOT (NULL VARL)))
      (then (CONS (LIST VARL VALL) AL))
      (else (pairlis VARL VALL AL))))
EXPR)

(DEFPROP pairlis
 (LAMBDA(NL VL NV)
  (if (NULL NL)
      (then NV)
      (else
       (CONS (LIST (CAR NL) (CAR VL))
             (pairlis (CDR NL) (CDR VL) NV)))))
EXPR)

(DEFPROP anchor
 (LAMBDA NIL
  (clink NIL (LIST) (LIST) (LIST) (LIST) (LIST) (LIST) #pn#))

EXPR)

(DEFPROP DATA
 (LAMBDA(TYFLDS)
  (CONS (PUTPROP
         (CAR TYFLDS)
         (LIST (QUOTE LAMBDA)
               (QUOTE (FLDS))
               (SUBST
                (CAR TYFLDS)
                (QUOTE TP)
                (QUOTE
                 (CONS (QUOTE TP)
                       (MAPCAR (QUOTE EVAL) FLDS)))))
         (QUOTE FEXPR))
        (DATADEF (CADR TYFLDS)
                 (QUOTE (CDR (EVAL (CAR TNEW)))))))
FEXPR)
```

```
(DEFPROP DATADEF
 (LAMBDA(FIELDS CDRS)
  (COND
   ((NULL FIELDS) NIL)
   (T
    (CONS
     (PUTPROP
      (CAR FIELDS)
      (LIST
       (QUOTE LAMBDA)
       (QUOTE (TNEW))
       (SUBST
        CDRS
        (QUOTE CCS)
        (QUOTE
         (COND
          ((NULL (CDR TNEW)) (CAR CCS))
          ((RPLACA CCS (EVAL (CADR TNEW)))
           (EVAL (CAR TNEW)))))))
      (QUOTE FEXPR))
     (DATADEF (CDR FIELDS) (LIST (QUOTE CDR) CDRS))))))
EXPR)

(DEFPROP assoc
 (LAMBDA (EX NV) (:= assoc (ASSOC EX NV)))
EXPR)

(DEFPROP get
 (LAMBDA (A ID) (:= get (GET A ID)))
EXPR)

(DEFPROP makepqs
 (LAMBDA(PNs)
  (MAPCAR (QUOTE
           (LAMBDA (PN)
            (LIST PN (LIST (QUOTE queue)))))
          PNs))
EXPR)

(DEFPROP remove-nth
 (LAMBDA(N P)
  (APPLY (QUOTE (LAMBDA (CL) (RPLACD CL (CDDR CL))))
         (LIST (NTH P N)))
  P)
EXPR)

(DEFPROP addvalue
 (LAMBDA (VS NV) (NCONC VS (LIST NV)))
EXPR)

(DEFPROP deactivatedp
 (LAMBDA (PN) (MEMBER PN #deactivated-pn-seq#))
EXPR)
```

```
(DEFPROP childprocessp
 (LAMBDA (PN) (MEMBER PN #child-pn-seq#))
EXPR)

(DEFPROP random-integer
 (LAMBDA (N) (ADD1 (FIX (TIMES N (RANDOM)))))
EXPR)

(DEFPROP MACDEFLIST
 (NIL MACRO-DEFINE MACRO-IZE MACLIST LCLIST)
VALUE)

(DEFPROP MACRO-DEFINE
 (LAMBDA(PAIRS)
  (MAPC (QUOTE
         (LAMBDA (PAIR) (MACRO-IZE (CAR PAIR) (CADR PAIR))))
        PAIRS))
EXPR)

(DEFPROP MACRO-IZE
 (LAMBDA(NICENAME REALNAME)
  (PUTPROP NICENAME
           (LIST (QUOTE LAMBDA)
                 (QUOTE (EX))
                 (LIST (QUOTE RPLACA)
                       (QUOTE EX)
                       (LIST (QUOTE QUOTE) REALNAME)))
           (QUOTE MACRO)))
EXPR)

(DEFPROP MACLIST
 (NIL (about (LAMBDA (E) (two (two E))))
      (from (LAMBDA (E) (two (one E))))
      (process-name CAR)
      (queue CADR)
      (execute EVAL)
      (error print)
      (pair LIST)
      (wildp NULL)
      (drainedp (LAMBDA (E) (NULL (CDR E))))
      (rac (LAMBDA (E) (CAR (LAST E))))
      (rdc (LAMBDA (E) (REVERSE (CDR (REVERSE E)))))
      (snoc (LAMBDA (L A) (APPEND L (LIST A))))
      (onep (LAMBDA (E) (ZEROP (SUB1 E))))
      (one CAR)
      (two CADR)
      (three CADDR)
      (four CADDDR)
      (five (LAMBDA (E) (CADR (CDDDR E))))
      (six (LAMBDA (E) (CADDR (CDDDR E))))
      (seven (LAMBDA (E) (CADDDR (CDDDR E))))
      (cdr2 CDDR)
      (cdr3 CDDDR)
      (cdr4 (LAMBDA (E) (CDR (CDDDR E))))
      (cdr5 (LAMBDA (E) (CDDR (CDDDR E))))
      (cdr6 (LAMBDA (E) (CDDDR (CDDDR E))))
      (ones (LAMBDA (E) (MAPCAR (QUOTE CAR) E)))
      (twos (LAMBDA (E) (MAPCAR (QUOTE CADR) E)))
      (threes (LAMBDA (E) (MAPCAR (QUOTE CADDR) E)))
      (fours (LAMBDA (E) (MAPCAR (QUOTE CADDDR) E)))
      (fives (LAMBDA (E) (MAPCAR
                           (QUOTE (LAMBDA (L)
                                   (CADR (CDDDR L))))
                          E)))
      (predicate CADR)
      (consequent CADDR)
      (alternative CADDDR)
      (block PROGN)
      (same EQ)
      (rember REMOVE)
      (ncdr NTH)
      (:= SETQ)
      (atomp ATOM)
      (symeval EVAL)
      (change replace))
VALUE)

(DEFPROP LCLIST
 (NIL (trace TRACE)
      (untrace UNTRACE)
      (grindef GRINDEF)
      (grinl GRINL)
      (break BREAK)
      (unbreak UNBREAK)
      (xevalu8 Xevalu8)
      (car CAR)
      (cdr CDR)
      (cadr CADR)
      (editf EDITF)
      (defprop DEFPROP)
      (setq SETQ)
      (dskin DSKIN)
      (dskout DSKOUT)
      (rplaca RPLACA)
      (rplacd RPLACD)
      (list LIST)
      (eval EVAL)
      (null NULL)
      (cond COND)
      (prog PROG)
      (or OR)
      (and AND)
      (RDC rdc)
      (RAC rac)
      (SNOC snoc)
      (ONEP onep)
      (ONE one)
      (TWO two)
      (THREE three)
      (EVALU8 evalu8)
      (XEVALU8 Xevalu8))
VALUE)
```

Appendix E -- System DATA Definitions

The following call to the function "DATA" (see Appendix D) must be made to initialize the system. In the section 2 examples, these calls are made when the file "CODA" is loaded by ILISP (see Appendix F). "DATA" is discussed by Shapiro in _Techniques of Artificial Intelligence_ [11].

```
(DATA suspended-process (&accumulator
                         &ind
                         &exp
                         &env
                         &evl
                         &unl
                         &pc
                         &clink
                         &pn))

(DATA clink ($ind $exp $env $evl $unl $pc $clink $pn))

(DATA value-record (%pn %value))

(DATA message-record (description message))

(DATA message-description (source transaction))
```

Appendix F -- System Startup

The following LISP expressions are used to load all needed files into ILISP. The first call to "DSKIN" loads the help functions, the MACROs, the interpreter proper, and the communication primitives. The second call to "DSKIN" executes the DATA definition expressions (see Appendix E) and loads three files of test process definitions. The calls to "MACRO-DEFINE" define a large group of MACROs for system and user use (see Appendix D). The last expression is the first of three needed to load the ILISP arithmetical function package. The other two expressions must be typed by the user at the terminal and are explained in the ILISP manual [1]. These expressions are contained in the file "CODA" that is loaded by specifying an input file during ILISP initialization.

```
(DSKIN (HELPF.LSP)(MACDEF.LSP)(INTERP.CDA)(COMMUN.CDA))
(DSKIN (SETUP.CDA)(TEST.DPF)(TEST.COM)(TEST.DIN))
(MACRO-DEFINE MACLIST)
(MACRO-DEFINE LCLIST)
(INC (INPUT SYS: (ARITH.LSP)))
```