SPLITS:
An Experiment in SCHEME-based Distributed Processing

by
Gregory P. Wolynes
Joseph R. Ginder
Barry Roitblat
Vicki Roland

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT No. 97
SPLITS:
AN EXPERIMENT IN SCHEME-BASED DISTRIBUTED PROCESSING
GREGORY P. WOLYNES
JOSEPH R. GINDER
BARRY ROITBLAT
VICKI ROLAND
AUGUST, 1980
SPLITS:

An Experiment in SCHEME-based Distributed Processing

by

Gregory P. Wolynes
Joseph R. Ginder
Barry Roitblat
Vicki Roland

INDIANA UNIVERSITY

Abstract:

SPLITS was developed as an experiment in using a lambda calculus language (SCHEME [15, 16]) as a base for implementing a distributed processing paradigm (PLITS [5]). The system has been implemented in SIMULA [1,2,6,7,14].

The distributed processing paradigm used (PLITS) has been extended to include added communication and synchronization primitive operations. An "incommunicado" primitive allows a process to isolate itself from other processes attempts at communication, with synchronization implications; and a lexical scanner for a SCHEME-like syntax has been implemented. PLITS "constant" modules have been eliminated; only templates (or prototypes [5] are used for creation of module-type instances. This considerably decreases the complexity of handling interprocess communication. These templates are defined through the use of a variation of the SCHEME "labels" expression, a defining mechanism for mutually recursive procedures.

The goal of this experiment was to show that a distributed processing paradigm could be elegantly added to the CODA interpreter for SCHEME [8] as implemented in SIMULA. The CODA interpreter defines SCHEME in a manner such that new primitives cannot be defined by the user (unlike AINTs in [15]). This alleviates the "levels problem" [15] incurred by defining outside of LABELS expressions.
SPLITS
An Experiment in SCHEME-based Distributed Processing

Introduction:

SPLITS is a distributed processing system (PLITS [4,5]) based in SCHEME [15]. SCHEME is an applicative order, lexically scoped, full-funarg lambda calculus language. Our implementation makes use of the CODA interpreter [8] described in the implementation section of this paper. Tail recursive functions execute without net growth of the interpreter stack. When a stack is required, it is kept in the form of continuation-links, (or c-links), which are registers specifying further evaluation.

Quasi-parallel distributed processing is modelled using the PLITS paradigm. Other paradigms were considered [3, 9, 10], but PLITS was chosen because of the philosophical implications of its synchronization mechanisms and the apparent elegance and flexibility of messages being environment-like. SPLITS is a system in which processes, called modules, are instances of a particular module definition, or template. Constant modules, or those in which the template itself is used as the process such that only the one instance of that definition may exist, are NOT included in our implementation. This considerably simplifies our handling of module creation; and we find that if the behavior of a constant module is desired, one need only define a module as usual and specify one instance
of that module. Module instances can be created by other
modules at any time, and also are created upon system
initiation. Communication between modules is in the form of
messages, which can be thought of as environments (or
a-lists [13]). Each message is a set of slots; each slot
being a name-value pair. Any module can create messages,
assign values in a message, and add slots to any message in
its possession; messages can be re-transmitted by any
module. Each message is sent to a particular specified
module and is identified as pertaining to a particular
subject (given a transaction). Transactions can be
specified as null if no particular transaction is desired.
The source of a message and its transaction code are
available for inspection in the slots named FROM and ABOUT,
respectively.

Synchronization enters the PLITS paradigm when a module
attempts to RECEIVE a message. When a message is SENT it
is queued up for later reception by the destination module.
No guarantee of reception is implied; and the sending
module continues execution. However, when a module attempts
to retrieve a message from its queue, or community, by
receiving it, that module is forced to wait (discontinue
execution) until the reception is possible and accomplished.
Appropriate predicates are provided to test for the presence
of satisfactory messages in a module's community. We have
added an additional synchronization condition. In our
extension to PLITS, a module may declare itself
INCOMUNICADO, or closed to communication. No message can be queued for a module once that module has declared itself closed in this manner. (However, message reception may continue as usual.) A module attempting to SEND a message to an incommunicado module is forced to wait until the destination module declares that it is no longer incommunicado and open for message queuing. Again, appropriate predicates are provided to test whether a module is open. Additional extensions include a CLEARQ primitive which allows a module to eliminate all messages from its community. This may at first consideration appear undesirable; however, no guarantee is assumed for a message reception, only for message delivery. A SUICIDE primitive has been included which terminates a modules execution permanently. In keeping with SCHEME, explicit data typing as in Feldman [5] has been excluded from SPLITS.

Modules are defined using a variation on the SCHEME "labels" expression, called GENESIS. This expression globally defines the module types contained therein, and may include some startup code. Instances of module definitions are created using the NEW primitive.

In our quasi-parallel SPLITS implementation, no fairness in evaluation of modules is implied or guaranteed. Modules are evaluated for a random number of cycles of the interpreter at a time, in random order. One cycle of the interpreter is defined as one execution of the contents of
the "pc" register. In practice, this is roughly equivalent to the evaluation of one top level element of the current expression being executed - either the recovery of a binding in the module's environment or recognizing a list element and pushing a continuation onto a stack (actually the continuations are stored in a TREE that degenerates into linear (stack-like) TREE when the SCHEME primitive CATCH isn't employed) in order to later continue with evaluation of the list elements.

The remainder of this paper consists of several sections and appendices. The next section is a discussion of the primitives available in SPLITS, the third contains several examples, and the last a discussion of the SIMULA implementation. A Backus-Naur form description for SPLITS is in Appendix A, the code in Appendix B.
Primitives:

Several LISP and SCHEME primitives have been implemented for SPLITs. For description and examples of AND, OR, ATOMP, CAR, CDR, CONS, EQ, GREATERP, MINUS, PLUS, NOT, NULL, PRINT, QUOTE, SAME (like EQUAL), and SET, see the LISP 1.5 Programmer's Manual [13]. For description and examples of CATCH and LABELS see "The Revised Report on SCHEME, a dialect of LISP" [15]. Although many of the above primitives are LISP primitives, the overall semantics of the interpreter makes it a version of SCHEME -- for example lexical scoping, rather than dynamic scoping as in LISP.

The following are module communication and synchronization primitives. Some have been copied from those in PLITS, others are partially changed or are extensions for SPLITs.

CLEARQ ()

CLEARQ deletes all messages from the calling module's community.

COMMUNICATE ()

COMMUNICATE opens the calling module for communication, making it possible for messages to be queued in its community. All modules waiting because the calling module had previously been incommunicado are released; this primitive is the "opposite" of INCOMMUNICADO.
EXTANT (<module-id>)

EXTANT is a predicate that determines whether the specified module exists (i.e. has been created and has not committed suicide).

EXTRACT (<slot-name> <message>)

EXTRACT returns the value associated with a slot-name in a message.

GENESIS (<module-definition-list> <start-up-code>)

GENESIS is a module definition and instance creation primitive which loosely corresponds to the SCHEME "LABELS" expression. The <module-definition-list> is a list of pairs, each pair being the name of a module template and a lambda expression containing that template. The template is embedded in the lambda expression as a normal LABELS expression; the only difference between this template and other LABELS expressions is in its use - not in its form. The formal parameters of the lambda expression are used to pass an initial environment to a module instance. The <start-up-code> is an expression to be evaluated, creating one or more module instances when used for system startup. The general form of a GENESIS expression follows:
(GENESIS
  ((module1 (LAMBDA ((formal-parameters))
      (LABELS ((labels-list) (expression)))))
  (module2 (LAMBDA ((formal-parameters))
      (LABELS ((labels-list) (expression)))))
  ...
  (moduleN (LAMBDA ((formal-parameters))
      (LABELS ((labels-list) (expression)))))
  (expression))

INCOMMUNICADO ()

INCOMMUNICADO closes the community of the calling
process to communication - no additional messages may
be queued. Modules sending messages to incommunucado
modules are forced to wait until the destination module
has re-opened its community by using the primitive
COMMUNICATE.

INCOMMUNICADOP (<module-id>)

This predicate determines if the specified module is
incommunucado.

MESSAGE (<name-list> <valuel> ... <valuen>)

MESSAGE creates and returns a message consisting of a
slot for every identifier in <name-list>, each paired
with a value specified from values <valuel> to
<valuen>. For instance, the second name in <name-list>
would be paired with <value2>. If <name-list> is null,
no values should be specified.
NEW (template-id <arg1> ... <argN>)

NEW creates and returns a pointer to a new module as an instance of template-id with the initial environment defined by matching arguments <arg1> to <argN> with the formal parameters of the lambda expression in which the specified template was embedded (in the template-defining GENESIS). No arguments (arg1...argN) need be given if no initial environment is desired. All modules are created with only their lexically enclosing environments, and empty communities. Any added environment must be built by the module itself, either from its definition or by using information sent to it in message form (after all, messages are forms of environments). Note that modules are NOT created with the same environment as their parent module.

PENDING (source <transaction>)

PENDING is a predicate which determines if a message from <source> and about <transaction> is in the calling module's community. Either or both of the arguments to PENDING may be NIL, in which case any source or transaction will match the null specification.

PRESENTP (slotname <message-id>)

PRESENTP checks for the existence of a particular slot name in a message.
RECEIVE (<source> <transaction>)

RECEIVE attempts to retrieve a message from the calling module's community which matches the <source> - <transaction> specification. If a matching message is not found, then the calling process is forced to wait until one is available to be retrieved. Again, NIL can be used in place of either or both specifications if a particular source or transaction is not required.

SEND (<destination> <transaction> <message>)

SEND transmits a message with the specified transaction to the destination module's community if the destination module has not declared itself incommunicado. If the destination module is incommunicado, then the calling module must wait until the destination module declares itself open for communication by using the primitive COMMUNICATE, at which time the message will be queued (NOT received!). Transactions may be specified as NIL.

SLOT (<slotname> <value> <message-id>)

SLOT creates a new slot in the specified message with the supplied name and value.

SUICIDE ()

SUICIDE terminates all execution by the calling module permanently. A terminated module is not extant.

TRANSACTION ()

TRANSACTION returns a new, unique transaction.
Example -- Factorial and Fibonacci

The following example demonstrates the mechanisms for module definition and creation.

.RUN SPLITS
SPLITS V2.5 RELEASE 1.0

%PLEASE SPECIFY ADDITION INFORMATION FOR LOGICAL FILE: INF
*FACTFIB.SPL
(GENESIS
  ((FACT
    (LAMBDA(N)
      (LABELS
        ((FACTORIAL
          (LAMBDA(N)
            (IF (GREATERP 2 N)
              1
              (TIMES N (FACTORIAL (MINUS N 1)))))))))
    (TIMES
      (LAMBDA(X Y)
        (IF (GREATERP 2 Y)
          X
          (PLUS X (TIMES X (MINUS Y 1)))))))))
  (FACTORIAL N))
)
(FIB
  (LAMBDA(N)
    (LABELS
      ((FIBONACCI
        (LAMBDA(X)
          (IF (GREATERP 3 X)
            1
            (PLUS (FIBONACCI (MINUS X 1))
              (FIBONACCI (MINUS X 2)))))))))
    ((LAMBDA (X) ((LAMBDA (Y) NIL) (NEW FACT 5)))
     (NEW FIB 10))))

PROCESS 0 EVALUATED TO NIL

PROCESS 2 EVALUATED TO 120

PROCESS 1 EVALUATED TO 55
Example: Alphonse and Gaston[5]

In this example a module called GASTON acts as a GUARDIAN over the variables A and B. GASTON accepts any of four commands - LOCK, UNLOCK, FETCH, UPDATE. LOCK signals GASTON to protect the variables from unauthorized access. UNLOCK restores free access by all processes. FETCH retrieves the values of A and B and transmits those values in a message to the module issuing the FETCH. UPDATE assigns new values to A and B.

ALPHONSE is an example of a module using the guardian GASTON. ALPHONSE LOCK's the variables so that only it's commands to GASTON are processed. Then it fetches the values of A and B, performs an UPDATE that exchanges A and B and negates their signs, if GASTON signals a successful operation then ALPHONSE signals GASTON to UNLOCK.
.RUN SPLITS
SPLITS V2.5 RELEASE 1.0

$PLEASE SPECIFY ADDITION INFORMATION FOR LOGICAL FILE: INF
*ALPGAS.SPL
(GENESIS
((GASTON
  (LAMBDA (A B LOCK MYKEY)
  (LABELS
   ((LOOP (LAMBDA (X) (LOOP (TEST (RECKAY))))
    (RECKAY
     (LAMBDA NIL
      (IF LOCK
       (RECEIVE NIL MYKEY)
       ((LAMBDA (MESS)
         ((LAMBDA (X) MESS)
          (SET (QUOTE MYKEY)
           (EXTRACT (QUOTE ABOUT) MESS))
           (RECEIVE NIL NIL))))
        (TEST
         (LAMBDA (MESSGE)
          (IF (SAME (EXTRACT (QUOTE ACTION) MESSGE) (QUOTE LOCK))
           (SET (QUOTE LOCK) TRUE)
           (IF (SAME (EXTRACT (QUOTE ACTION) MESSGE) (QUOTE UNLOCK))
             (SET (QUOTE LOCK) NIL)
             (IF (SAME (EXTRACT (QUOTE FROM) MESSGE) (QUOTE FETCH))
              (SEND (EXTRACT (QUOTE FROM) MESSGE) MYKEY
               (MESSAGE (QUOTE (A B ACTION))
                A
                B
                (QUOTE BLAH1)))
              (IF (SAME (EXTRACT (QUOTE ACTION) MESSGE) (QUOTE UPDATE))
               ((LAMBDA (W X Y Z)
                 (SEND (EXTRACT (QUOTE FROM) MESSGE) MYKEY
                  (MESSAGE
                   (QUOTE (ACTION))
                   (QUOTE BLAH2)))))
               (SET (QUOTE A)
                (EXTRACT (QUOTE A) MESSGE))
               (SET (QUOTE B)
                (EXTRACT (QUOTE B) MESSGE))
               (PRINT (CONS (QUOTE A) (CONS A NIL)))
               (PRINT (CONS (QUOTE B) (CONS B NIL)))
               (PRINT (QUOTE BAD-ACTION)))
        )))
])))
)
(ALPHONSE
  (LAMBDA (GASTON NUM)
    (LABELS
      ((GO
        (LAMBDA (TRANS)
          ((LAMBDA (A B C D MESSGE)
            ((LAMBDA (X MESS)
              (IF (SAME (EXTRACT (QUOTE ACTION) MESS)
                (QUOTE REJECT)
                 (PRINT (QUOTE REJECTED))
                 ((LAMBDA (A)
                   (SEND GASTON
                     TRANS
                     (MESSAGE (QUOTE (ACTION A B))
                     (QUOTE UNLOCK)
                     Ø
                     Ø))))
                 (PRINT (CONS (QUOTE ALPHONSE)
                               (CONS NUM
                                  (CONS (QUOTE UNLOCK) NIL)))))))))
               (SEND GASTON
                 TRANS
                 (MESSAGE (QUOTE (ACTION A B))
                 (QUOTE UPDATE)
                 (MINUS Ø (EXTRACT (QUOTE B) MESSGE))
                 (MINUS Ø
                 (EXTRACT (QUOTE A) MESSGE))))
               (RECEIVE GASTON TRANS) )))
            (PRINT (CONS (QUOTE ALPHONSE)
                         (CONS NUM (CONS (QUOTE LOCK) NIL)))))
          (SEND GASTON
           TRANS
           (MESSAGE (QUOTE (ACTION A B))
           (QUOTE LOCK)
           Ø
           Ø)))
        (PRINT (CONS (QUOTE ALPHONSE)
                     (CONS NUM (CONS (QUOTE FETCH) NIL)))
        (SEND GASTON
         TRANS
         (MESSAGE (QUOTE (ACTION A B))
         (QUOTE FETCH)
         Ø
         Ø))
        (RECEIVE GASTON TRANS))))
      (GO ( TRANSACTION))))))))
    (LAMBDA (X)
      ((LAMBDA (A B C) NIL) (NEW ALPHONSE X 1)
       (NEW ALPHONSE X 2)
       (NEW ALPHONSE X 3))
      (NEW GASTON 5 3 NIL Ø))
    )
(ALPHONSE 2 LOCK)
PROCESS 0 EVALUATED TO NIL
(ALPHONSE 3 LOCK)
(A 5)
(B 3)
(Alphonse 1 UNLOCK)
PROCESS 2 EVALUATED TO TRUE
(A -3)
(B -5)
(ALPHONSE 3 UNLOCK)
PROCESS 4 EVALUATED TO TRUE
Example -- Dining Philosophers

A solution to the dining philosophers problem is given on the following pages along with a sample execution. 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 Kaubisch [11].

The fork-monitor module begins by executing initialization code and receiving a message which supplies it with the names of the philosopher modules. Once this is accomplished, the fork-monitor issues a receive for any available message. After a message is received, it is DECODEd 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, additional messages can be received. PHILS is a list used to associate 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.
.RUN SPLITS
SPLITS V2.5 RELEASE 1.0

*PLEASE SPECIFY ADDITION INFORMATION FOR LOGICAL FILE: INF
*DINING.SPL

(GENESIS
  ((FORK-MONITOR
    (LAMBDA(EAT-TR THINK-TR PHILS-TR)
      (LABELS
        ((PICKUP
          (LAMBDA(I)
            (IF (AND (FORK-COND (MOD5 (MINUS I 1))) (FORK-COND I))
              (GIVE-FORKS I)
              (READY-WAIT I))))))
          (READY-WAIT
           (LAMBDA(I))
            (IF (FORK-COND (MOD5 (MINUS I 1)))
             ((LAMBDA (Y Z) (PICKUP I))
              (RECEIVE (PHIL (MOD5 (PLUS I 1))) THINK-TR)
              (PUTDOWN (MOD5 (PLUS I 1)))))
             ((LAMBDA (Y Z) (PICKUP I))
              (RECEIVE (PHIL (MOD5 (MINUS I 1))) THINK-TR)
              (PUTDOWN (MOD5 (MINUS I 1)))))))))
          (GIVE-FORKS
           (LAMBDA(I))
            ((LAMBDA (X Y Z) (DECODE (RECEIVE NIL NIL)))
             (SET-FORK-COND (MOD5 (MINUS I 1)) NIL)
             (SET-FORK-COND I NIL)
             (SEND (PHIL I) EAT-TR (MESSAGE NIL)))))
           (PUTDOWN
            (LAMBDA(I))
             ((LAMBDA (Y Z) (SEND (PHIL I) THINK-TR (MESSAGE NIL)))
              (SET-FORK-COND (MOD5 (MINUS I 1)) TRUE)
              (SET-FORK-COND I TRUE))))))
          (DECODE
           (LAMBDA(MSG)
            (IF (EQ EAT-TR (EXTRACT (QUOTE ABOUT) MSG))
             (PICKUP (EXTRACT (QUOTE PHILNUM) MSG))
             ((LAMBDA (Z) (DECODE (RECEIVE NIL NIL)))
              (PUTDOWN (EXTRACT (QUOTE PHILNUM) MSG))))))
            ((MOD5 (LAMBDA (N) (IF (EQ 0 N) 5 (IF (EQ 6 N) 1 N)))
             (PHIL (LAMBDA (I) (NTH PHILS I)))
             (FORK-COND (LAMBDA (I) (NTH FORKS I)))
             (SET-FORK-COND
              (LAMBDA (N V) (SET (QUOTE FORKS) (SFC-HELP FORKS N V))))
              (LST5
               (LAMBDA(E1 E2 E3 E4 E5)
                (CONS E1 (CONS E2 (CONS E3 (CONS E4 (CONS E5 NIL)))))))))
          )
        )
      )
    )
  )
)
(SFC-HELP
(LAMBDA(L N V)
  (IF (EQ N 1)
    (CONS V (CDR L))
    (CONS (CAR L) (SFC-HELP (CDR L) (MINUS N 1) V))))
(NTH
(LAMBDA(L N)
  (IF (EQ N 1) (CAR L) (NTH (CDR L) (MINUS N 1))))))
(INI
(LAMBDA NIL
  ((LAMBDA(Z)
    (SET (QUOTE FORKS) (LIST5 TRUE TRUE TRUE TRUE TRUE TRUE))
    ((LAMBDA(MSG)
      (SET
        (QUOTE PHILS)
        (LIST5
          (EXTRACT (QUOTE P1) MSG)
          (EXTRACT (QUOTE P2) MSG)
          (EXTRACT (QUOTE P3) MSG)
          (EXTRACT (QUOTE P4) MSG)
          (EXTRACT (QUOTE P5) MSG))))
        (RECEIVE NIL PHILS-TR))))))
  ((LAMBDA (Z) (DECODE (RECEIVE NIL NIL))) (INI))))
(PHILOSOPHER
(LAMBDA(FM OWN-ID EAT-TR THINK-TR)
  (LABELS
    ((SEEK-EAT
      (LAMBDA NIL
        ((LAMBDA (X Y Z) (SEEK-THINK))
        (SEND FM EAT-TR (MESSAGE (QUOTE (PHILNUM)) OWN-ID))
        (RECEIVE FM EAT-TR)
        (EAT)))))
      (EAT
        (LAMBDA NIL
          (PRINT3 (QUOTE PHILOSOPHER:) OWN-ID (QUOTE EATING!!))))
        (SEEK-THINK
          (LAMBDA NIL
            ((LAMBDA (X Y Z) (SEEK-EAT))
            (SEND FM THINK-TR (MESSAGE (QUOTE (PHILNUM)) OWN-ID))
            (RECEIVE FM THINK-TR)
            (THINK)))))
          (THINK
            (LAMBDA NIL
              (PRINT3 (QUOTE PHILOSOPHER:) OWN-ID (QUOTE THINKING!!))))
              (PRINT3
                (LAMBDA(A1 A2 A3)
                  ((LAMBDA (Y Z) (PRINT A3)) (PRINT A1) (PRINT A2)))))
                (SEEK-EAT)))))))
(((LAMBDA(EAT-TR THINK-TR PHILS-TR))
  ((LAMBDA(FM))
    ((LAMBDA(PH1 PH2 PH3 PH4 PH5))
      ((LAMBDA(MSG) (SEND FM PHILS-TR MSG))
       (MESSAGE (QUOTE (P1 P2 P3 P4 P5)) PH1 PH2 PH3 PH4 PH5)))
      (NEW PHILOSOPHER FM 1 EAT-TR THINK-TR)
      (NEW PHILOSOPHER FM 2 EAT-TR THINK-TR)
      (NEW PHILOSOPHER FM 3 EAT-TR THINK-TR)
      (NEW PHILOSOPHER FM 4 EAT-TR THINK-TR)
      (NEW PHILOSOPHER FM 5 EAT-TR THINK-TR)))
    (NEW FORK-MONITOR EAT-TR THINK-TR PHILS-TR)))
  (TRANSACTION)
  (TRANSACTION)
  (TRANSACTION)))

PROCESS  0, EVALUATED TO TRUE

PHILOSOPHER:
  2
EATING!!
PHILOSOPHER:
  5
EATING!!
PHILOSOPHER:
  5
THINKING!!
PHILOSOPHER:
  4
EATING!!
PHILOSOPHER:
  2
THINKING!!
PHILOSOPHER:
  1
EATING!!
PHILOSOPHER:
  4
THINKING!!
PHILOSOPHER:
  3
EATING!!
PHILOSOPHER:
  1
THINKING!!
PHILOSOPHER:
  5
EATING!!
PHILOSOPHER:
  3
THINKING!!
PHILOSOPHER:
  2
EATING!!
SIMULA Implementation

SPLITS is based on a SIMULA implementation of the CODA interpreter [8]. The code is included in Appendix B.

CODA is an iterative, continuation based interpreter for SCHEME. A continuation is made up of "registers" collected as a "snapshot" of the system environment at the time of the continuation's creation. The continuation is stored as a continuation link, or c-link register.

CODA implements only a very "pure" subset of SCHEME. DEFINE [15] is not available to the programmer, and the property-list is not explicitly modifiable by CODA SCHEME expressions. Implementing SCHEME in this manner alleviates the "levels" problem found in other dialects of this language, incurred by being able to extend the interpreter.

Each register contains some record indicating how further processing is to proceed upon popping a c-link off the c-link. The current expression, or #EXP# register, is the current expression being evaluated. The #EVL# register and the #UNL# register are the evaluated and unevaluated pieces of the current expression, respectively. The #ENV# register points to the environment in which #EXP# is to be evaluated. The program counter, or #PC# register, points to a section of code that, when executed, proceeds with the evaluation of #EXP#. Every c-link has a register member (#CLINK#) which is also a c-link -- this is how the continuations are maintained. An accumulator (BOX) register accumulates a value to be returned. It is not part of a c-link.

In the algorithmic description of CODA below, the code labelled by "L1" evaluates most simple SCHEME expressions. If a more complicated expression is encountered (ie. LAMBDA or LABELS), then some initial processing is accomplished by L1, but further processing is deferred to the code labelled "L2". The latter code is able to take the pre-processed expression and evaluate it. The procedure RESTORE pops the c-link. BETALINK is a procedure that transforms the environment into the proper form for use by LABELS-created mutually recursive procedures.
PROCEDURE EVAL(X,N);
BEGIN
  PROCEDURE RESTORE;
  IF NOT(NULL(#CLINK#)) THEN
  BEGIN
    #EXP#:=#CLINK#.EXPR;
    #ENV#:=#CLINK#.ENVIR;
    #EVL#:=#CLINK#.EVLST;
    #UNL#:=#CLINK#.UNLST;
    #PC#:=#CLINK#.PCNTR;
    #CLINK#:=#CLINK#.LINK
  END
  ELSE ERROR(PROCESS-RAN-OUT,#EXP#);
  FUNCTION BETALINK(LABLIST,ENVTEMP);
  BEGIN
    TEMP:=ENVIR();
    ENVTEMP:=#ENV#;
    WHILE LABLIST NE NIL DO
    BEGIN
      BETAPT:=BETA(ONE(LABLIST),LAMBDAXEXP,TEMP);
      ENVTEMP:=ENVIR(ONE(LABLIST),IDENT,BETAPT,ENVTEMP);
      LABLIST:=CDR(LABLIST)
    END;
    TEMP.IDENT:=ENVTEMP.IDENT;
    TEMP.VAL:=ENVTEMP.VAL;
    TEMP.ENVLINK:=ENVTEMP.ENVLINK;
    BETALINK:=TEMP;
  END;
  GO TO X1;
L1: IF ATOM(#EXP#) THEN
    BEGIN
      IF (#EXP# EQ NIL) OR (#EXP# EQ T) OR
      NUMBERP(#EXP#) OR PRIMOP(#EXP#) THEN BOX:=#EXP#
      ELSE IF ASSOC(#EXP#, #ENV#)
          THEN BOX:=VALUE(#EXP#, #ENV#)
      ELSE
          BEGIN
            ERROR(ILLEGAL-ATOM, #EXP#);
            BOX:=NIL
          END;
          RESTORE
    ELSE
      BEGIN
        CASE #EXP# OF
        "QUOTE"
          BEGIN
            BOX:=#EXP#.STRING;
            RESTORE
          END;
        "IF"
          BEGIN
            #CLINK#: = FRAME(#EXP#, #ENV#, #EVL#, 
              #UNL#, M1, #CLINK#);
            #EXP#: = #EXP#.PREDICATE;
            #PC#: = L1
          END;
        "CATCH"
          BEGIN
            #ENV#: = ENVIR(#EXP#, IDENT, #CLINK#, #ENV#);
            #EXP#: = #EXP#.FUNCALL;
            #PC#: = L1
          END;
        "LABELS"
          BEGIN
            #ENV#: = BETALINK(#EXP#.DEFS, #ENV#);
            #EXP#: = #EXP#.FUNCALL;
            #PC#: = L1
          END;
        OTHERWISE
          BEGIN
            #EVL#: = NIL;
            #UNL#: = #EXP#;
            #PC#: = L2
          END
      END
      END;
GO TO #PC#;
L2: IF (NOT(NULL(#UNL#))) THEN
    BEGIN
        #CLINK#:=FRAME(#EXP#, #ENV#, #EVL#, #UNL#, M2, #CLINK#);
        #EXP#:=ONE(#UNL#);
        #PC#:=L1
    END
ELSE IF PRIMOP(ONE(#EVL#)) THEN
    BEGIN
        BOX:=APPLY(ONE(#EVL#), CDR(#EVL#));
        RESTORE
    END
ELSE
    CASE ONE(#EVL#) OF
        "BETA"
        BEGIN
            #ENV#:=PAIRLIS(#EVL#.LAMBDAEXP.PRAMS,
                          CDR(#EVL#), #EVL#.ENVIR);
            #PC#:=L1
        END
        "DELTA"
        BEGIN
            #CLINK#:=#EVL#.LINK;
            BOX:=TWO(#EVL#);
            RESTORE
        END
OTHERWISE
        ERROR(BAD-FUNCTION-EVARGLIST, #EXP#)
    END
END;
GO TO #PC#;
M1: #EXP#: = IF NOT(NULL(BOX)) THEN THREE(#EXP#)
    ELSE FOUR(#EXP#);
    PC := L1;
    GO TO #PC#;
M2: #EVL#: := SNOOC(#EVL#, BOX);
    #UNL#: := CDR(#UNL#);
    PC := L2;
    GO TO #PC#;
X1: #CLINK#: := FRAME(NIL, NIL, NIL, NIL, El, NIL);
    #ENV#: := N;
    #EXP#: := X;
    #PC#: := L1;
    #EVL#: := NIL;
    #UNL#: := NIL;
    GO TO #PC#;
E1: PRINT BOX
END.

Our implementation was developed through several stages. We began by implementing the CODA interpreter in SIMULA. A major effort in the testing of this interpreter was providing it with already lexically scanned input which was intentionally done by hand. The next phase was a simple modification (because of the structure afforded by SIMULA) to convert the interpreter to accommodate quasi-parallel processing. The third phase of our system development was to introduce the PLITS distributed processing paradigm. PLITS was chosen rather than Hoare's CSP model [10] and Brinch Hansen's distributed Processes (DP) model [3] because of the implications of its synchronization mechanism and its environment-like messages. In pure PLITS, synchronization enters the system when a module attempts to receive a message; if none is available,
it must wait until one is. However, a module can send a message with no synchronization involvement. In CSP, a process must wait upon issuing a send command until the destination process acknowledges receipt. In Brinch Hansen's DP model, a process calling another process must wait not only until the call is received, but until the called process completes the requested action. Both of these imply a stronger coupling than that which we find consistent with the notion of "distributed computing". It should be noted, however, that any of these distributed processing paradigms can be modelled in any of the others (using buffer processes, etc.).

The last stage of development was designing and implementing a lexical scanner for our system. It is our opinion that developing the system in stages such as these contributed much to the overall coherence of our implementation. We were readily able to envision the needed facilities and consistent means of implementing them when examining these developments separately.

The lexical scanner takes input of SCHEME-like expressions, and parses them to generate class objects linked in the appropriate structure to correspond with the SPLITS expressions. It should be noted that while the scanner does accept SCHEME-like code as input, we
assume a certain amount of "pre-scanning". For instance, we would not expect anyone to actually write programs using the expanded lambdas for sequential execution as we have in our examples -- these expansions would be the result of the "pre-scanner" taking an expression like BLOCK ( (expl) ... (expN)) (see [15]), for example, in an ALGOL-like syntax, and re-writing it in the form acceptable to the current scanner. The reader will probably recognize several instances in our examples where some such convenient primitive as BLOCK would have been useful.

The scanner is implemented using the SIMULA text manipulation primitives. The system originating GENESIS expression is read as a single string and then parsed from the top down, much as an expression is interpreted in a LISP interpreter. Scanning then became a matter of taking each element of an expression and creating the appropriate internal data object for our SIMULA interpreter with the appropriate links.

Extensive use was made of the CLASS construct [2,6] in our implementation. We found that a certain elegance could be achieved by defining class objects to correspond to the various types of expressions in the SPLITS system. The implementation in SIMULA defines each SPLITS expression-type as a class, enabling us to define within each its own specific operations. For
example, the evaluator for each expression is included as a generic \textit{virtual [2]} procedure within the corresponding class definition. By implementing the interpreter in this manner, we were able to automatically invoke these specific operations with a single call to a higher level \textit{virtual} procedure. Similar structuring was used to define \textit{virtual} procedures \textsc{apply}, \textsc{print}, and \textsc{create} (a parse-related procedure).

Several standard data structures were employed in our \textsc{simula} implementation. A sparse matrix information structure \cite{12} was adapted for use as the message queue of each module. The entire matrix structure is called a \textit{community}. The row and column head nodes are called \textit{fromheads} and \textit{tranheads} (for transaction-head) respectively. Each matrix element is known as a \textit{communique}; and elements of a communique \textit{(messages)} are known as \textit{communiques} (pronounced communi-kay). The organization of this structure follows rather obviously. Fromheads exist for each module from which a message has been queued; and tranheads exist for each transaction code of messages that have been queued. Linked to the appropriate heads are the communiques of communiques from a particular module pertaining to a particular transaction. This data structure gives the respective communiques the desired \textit{FIFO} properties.
As modules are created they are added to a pool from which processes are randomly selected for execution (random number of iterations of the pc-loop [8]). This pool is implemented using our own version of the SIMULA SIMSET primitives [1, 2] Each module is defined as a class instance of type community concatenated [2] with a class instance of type "process". Our "SIMSET" is a class prefixed [2] to the former class.
Conclusion

We found that CODA interpreter for lambda calculus provided an elegant and convenient base from which to build a distributed processing system. The PLITS formalism, with its environment-like messages, provided a consistent extension of SCHEME. SIMULA was found to facilitate a quite elegant and modular interpreter implementation.
Bibliography


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

Appendix A

Syntax of SPLITS Primitives

expression ::= od communication-expression cd |
             od structuring-expression cd |
             od primitive cd |
             od function cd |
             list | atom

communication-expression ::= genesis-exp | extract-exp |
                           message-exp | slot-exp |
                           new-exp | send-exp |
                           receive-exp | pending-exp |
                           incommunicadop-exp |
                           incommunicado-exp |
                           suicide-exp | clearq-exp |
                           extantp-exp | presentp-exp |
                           absentp-exp

structuring-expression ::= labels-exp | catch-exp | if-exp |
                         block-exp | lambda-exp
primitive ::= lessp-exp | cons-exp | car-exp | cdr-exp |
              plus-exp | times-exp | divide-exp |
              greaterp-exp | null-exp | eq-exp | quote-exp |
              same-exp | atomp-exp | aset-exp | and-exp |
              or-exp | print-exp | not-exp | minus-exp

function ::= identifier arglist

list ::= od {expression} cd

atom ::= letter {character} delimiter | number

genesis-exp ::= GENESIS od proclist cd init-code

extract-exp ::= EXTRACT identifier identifier

message-exp ::= MESSAGE od {identifier} cd {expression}

slot-exp ::= SLOT identifier expression identifier

new-exp ::= NEW identifier args

send-exp ::= SEND identifier transaction identifier

receive-exp ::= RECEIVE identifier transaction

pending-exp ::= PENDING identifier transaction

incommunicadop-exp ::= INCOMMUNICADOP identifier

incommunicado-exp ::= INCOMMUNICADO
suicide-exp ::= SUICIDE

clearq-exp ::= CLEARQ

extantp-exp ::= EXTANTP identifier

presentp-exp ::= PRESENTP identifier identifier

transaction-exp ::= TRANSACTION

labels-exp ::= LABELS od funlist cd od function cd

catch-exp ::= CATCH identifier expression

if-exp ::= IF expression expression expression

lambda-exp ::= LAMBDA formal-parameters expression

cons-exp ::= CONS expression expression

car-exp ::= CAR expression

cdr-exp ::= CDR expression

plus-exp ::= PLUS expression expression

minus-exp ::= MINUS expression expression

greaterp-exp ::= GREATERP expression expression

null ::= NULL expression

eq-exp ::= EQ expression expression

quote-exp ::= QUOTE expression
same-exp ::= SAME expression expression
atomp-exp ::= ATOMP expression
aset-exp ::= SET expression expression
and-exp ::= AND expression expression
or-exp ::= OR expression expression
print-exp ::= PRINT expression
not-exp ::= NOT expression
proclist ::= od identifier
            od LAMBDA arglist od labels-exp cd cd cd
            [ od identifier od lambda-exp cd cd ]
initcode ::= expression
transaction ::= identifier | NIL
arglist ::= od {identifier} cd
args ::= {expression}
formal-parameters ::= od {identifier} cd
funlist ::= od identifier lambda-exp cd
            [ od identifier lambda-exp cd ]
identifier ::= letter {character} delimiter
letter ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
         'G' | 'H' | 'I' | 'J' | 'K' | 'L' |
         'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
         'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
         'Y' | 'Z'

character ::= letter | digit | '+' | '*' | '$' | ',' |
            '|' | '!' | '@' | '&' | '-' | '=' | ':' |
            '.' | ';' | ':' | '?' | ';' | '/'

number ::= digit [digit] delimiter |
         '-' digit [digit] delimiter |
         '+' digit [digit] delimiter

digit ::= '0' | '1' | '2' | '3' | '4' |
        '5' | '6' | '7' | '8' | '9'

delimiter ::= od | cd | '.' | ',' | <cr> | ','

od ::= '(' | '[' | '<'

cd ::= ')' | ']' | '>

NOTE: [] stand for zero or more occurrences of the enclosed item.
Appendix B

SPLITS Source:

begin
class FORM;
virtual: ref(FORM) procedure CREATE;
procedure PRINT;
procedure EVAL;
procedure EVLIST;
begin
    procedure PRINT;
    begin
        outtext("UH OH NO MATCH");
    end***PRINT***;
    procedure EVLIST;
    begin
        outimage;
        outtext("ERROR -- BAD FUNCTION DD");
        outimage;
    end***EVLIST***;
end***FORM***;
FORM class LIST(HEAD, REST);
ref(FORM) HEAD;
ref(LIST) REST;
begin
    ref(LIST) procedure ADDONEND(ITEM);
    ref (FORM) ITEM;
    begin
        ref(LIST) PTR, TEMP;
        TEMP := this LIST;
        PTR := new LIST(TEMP.HEAD, none);
        ADDONEND := PTR;
        TEMP := TEMP.REST;
        while TEMP /= none do
            begin
                PTR.REST := new LIST(TEMP.HEAD, none);
                PTR := PTR.REST;
                TEMP := TEMP.REST;
            end***WHILE***;
        PTR.REST := new LIST(ITEM, none);
    end***ADDONEND***;
end***EVAL***;
procedure EVAL;
begin
    EVL$ := none;
    UNL$ := this LIST;
    PC$ := new DOEVLIS;
end***EVAL***;
procedure EVLIS;
begin
    CLINK$ := new CLINK (EXP$, ENV$, EVL$, this LIST, new
    BLOCK2, CLINK$);
    EXP$ := HEAD;
    PC$ := new DOEVAL;
end***EVLIS***;
procedure PRINT;
begin
    outtext("(");
    this LIST.PRINT1;
    outtext(")");
end;
procedure PRINT1;
begin
    HEAD.PRINT;
    if REST /= none then
        begin
            outtext(" ");
            REST.PRINT1;
        end;
    end;
end***LIST***;
FORM class ATOM;
begin
  procedure EVAL;
  begin
    ACC$ := if EXP$ is LITERAL
    then ENV$.ASSOC(EXP$ qua LITERAL)
    else EXP$;
    CLINK$.RESTORE;
  end;
end***ATOM***;

ATOM class CONSTANT;
begin
  ref (FORM) procedure CREATE(DUMMY);
  text DUMMY;
  begin
    CREATE := this CONSTANT;
    end***CREATE***;
end***CONSTANT***;

CONSTANT class NUMBER (VAL);
integer VAL;
begin
  procedure PRINT;
  begin
    outint(VAL,8);
    end***PRINT***;
end***NUMBER***;
CONSTANT class TRUE#;
begin
  procedure PRINT;
  begin
    outtext("TRUE ");
  end;
end;

CONSTANT class NIL;
begin
  ref (FORM) HEAD, REST;
  procedure PRINT;
  begin
    outtext("NIL ");
  end;
  HEAD := this NIL;
  REST := this NIL;
end;
ATOM class PRIMOP;
virtual: ref (FORM) procedure APPLY;
begin
  ref (FORM) procedure CREATE (ARGS);
text ARGS;
begin
  CREATE := this PRIMOP;
end;
procedure EVLIS;
begin
  ACC$ := APPLY (EVL$.REST);
  CLINK$.RESTORE;
end***EVLIS***;
end***PRIMOP***;

PRIMOP class EQX;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
  APPLY:-- if ARGS.HEAD qua NUMBER.VAL =
         ARGS.REST.HEAD qua NUMBER.VAL
        then TRUE$;
        else NIL$;
end***APPLY***;
procedure PRINT;
begin
  outtext("EQ ");
end***PRINT***;
end***EQX***;

PRIMOP class SAMEX;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
  APPLY:-- if ARGS.HEAD qua LITERAL.SAME(ARGS.REST.HEAD)
         then TRUE$;
         else NIL$;
end***APPLY***;
procedure PRINT;
begin
  outtext("SAME ");
end***PRINT***;
end***SAMEX***;
PRIMOP class NOTX;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    APPLY:=-if (ARGS.HEAD == none) or (ARGS.HEAD == NIL$)
      then TRUE$
      else NIL$
    end***APPLY***;
    procedure PRINT;
    begin
      outtext("NOT ");
    end***PRINT***;
  end***NOTX***;
end

PRIMOP class ORX;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    boolean DONE;
    ref (LIST) TEMP;
    DONE:=false;
    TEMP:-ARGS;
    while TEMP /= none and not DONE do
      begin
        if TEMP.HEAD /= NIL$ and TEMP.HEAD /= none then
          begin
            DONE:=true;
            APPLY:=-TRUE$
          end
        else TEMP:=-TEMP.REST
      end***WHILE***;
    if not DONE then APPLY:-NIL$
end***APPLY***;
  procedure PRINT;
  begin
    outtext("OR ")
  end***PRINT***;
end***ORX***;
PRIMOP class ANDX;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
  boolean DONE;
  ref (LIST) TEMP;
  TEMP:=ARGS;
  DONE:=false;
while TEMP /= none and not DONE do
begin
  if TEMP.HEAD == NIL$ or TEMP.HEAD == none then
begin
    DONE:=true;
    APPLY:=-NIL$
  end
else TEMP:=-TEMP.REST
end***WHILE***;
if ARGS /= none and not DONE then APPLY:=-TRUE$
else
if ARGS /= none then APPLY:=-NIL$
end***APPLY***;
procedure PRINT;
begin
  outtext("AND ")
end***PRINT***;
end***ANDX***;

PRIMOP class PLUS;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
  APPLY:=-new NUMBER((ARGS.HEAD qua NUMBER.VAL)+(ARGS.REST.HEAD qua NUMBER.VAL))
end***APPLY***;
procedure PRINT;
begin
  outtext("PLUS ")
end***PRINT***;
end***PLUS***;
PRIMOP class MINUS;
begin
    ref (CONSTANT) procedure APPLY(ARGS);
    ref (LIST) ARGS;
    begin
        APPLY:=-new NUMBER((ARGS.HEAD qua NUMBER.VAL)-(ARGS.REST.HEAD qua NUMBER.VAL))
    end***APPLY***;
    procedure PRINT;
    begin
        outtext("MINUS ")
    end***PRINT***;
end***MINUS***;

PRIMOP class CAR;
begin
    ref (FORM) procedure APPLY(ARGS);
    ref (LIST) ARGS;
    begin
        APPLY :- ARG5.HEAD qua LIST.HEAD;
    end***APPLY***;
    procedure PRINT;
    begin
        outtext("CAR ");
    end***PRINT***;
end***CAR***;

PRIMOP class CDR;
begin
    ref (FORM) procedure APPLY(ARGS);
    ref (LIST) ARGS;
    begin
        APPLY :- ARG5.HEAD qua LIST.REST;
    end***APPLY***;
    procedure PRINT;
    begin
        outtext("CDR ");
    end***PRINT***;
end***CDR***;
PRIMOP class CONS;
begin
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
    if ARGS.REST.HEAD in NIL then
      APPLY := new LIST(ARGS.HEAD, none)
    else
      APPLY := new LIST(ARGS.HEAD, ARGS.REST.HEAD);
end***APPLY***;
procedure PRINT;
begin
  outtext("CONS ");
end***PRINT***;
end***CONS***;

PRIMOP class GREATERP;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
  APPLY := if (ARGS.HEAD qua NUMBER.VAL) >
(ARGS.REST.HEAD qua
  NUMBER.VAL)
    then TRUE$
  else NIL$;
end***APPLY***;
procedure PRINT;
begin
  outtext("GREATERP ");
end***PRINT***;
end***GREATERP***;
PRIMOP class ASSIGN;
begin
  ref(CONSTANT) procedure APPLY(ARGS);
  ref(LIST) ARGS;
  begin
    APPLY := NIL$;
    if ARGS.REST.REST == none then
      ENV$.ASSIGN(ARGS.HEAD qua LITERAL,ARGS.REST.HEAD)
    else
      ARGS.REST.REST.HEAD qua ENVIR.
      ASSIGN(ARGS.HEAD qua LITERAL,ARGS.REST.HEAD);
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("SET ");
  end;
end***ASSIGN***;

PRIMOP class PRINTVAL;
begin
  ref(CONSTANT) procedure APPLY(ARGS);
  ref(LIST) ARGS;
  begin
    APPLY := NIL$;
    ARGS.HEAD.PRINT;
    OUTIMAGE;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("PRINT");
  end;
end***PRINTVAL***;

PRIMOP class ATOMP;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    APPLY: -if (ARGS.HEAD ==none) or (ARGS.HEAD is ATOM)
      then TRUE$
    else NIL$;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("ATOMP ");
  end***PRINT***;
end***ATOMP***;
PRIMOP class MESSAGE;
begin
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
    ref (ENVIR) TEMP;
    if ARGs.HEAD in NIL then
      APPLY :- new ENVIR(none,none,none)
    else
      begin
        TEMP:-new ENVIR(ARGs.HEAD qua LIST.HEAD,
                         ARGs.REST.HEAD,
                         new ENVIR(none,none,none));
        APPLY :- TEMP.PAIRLIS(ARGs.HEAD qua
                               LIST.REST,ARGS.REST.REST);
      end;
  end***APPLY***;
procedure PRINT;
begin
  outtext("MESSAGE ");
end;
end***MESSAGE***;

PRIMOP class SLOT;
begin
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
  APPLY:-new
    ENVIR(ARGS.HEAD,ARGS.REST.HEAD,ARGS.REST.REST.HEAD)
end***APPLY***;
procedure PRINT;
begin
  outtext("SLOT")
end***PRINT***;
end***SLOT***;
PRIMOP class NU;
begin
  procedure PROCERR;
  begin
    outimage;
    outtext(" ILLEGAL ARGs NEW PROCESS ");
    outimage;
  end
  end***PROCERR***;
  ref (FORM) procedure APPLY(ARGs);
  ref (LIST) ARGs;
  begin
    ref (PROCESS) PROC;
    inspect ARGs.HEAD
    when BETA do
    begin
      if LAMBDAEXP.BODY is LABELS then
      begin
        PROC := new PROCESS(PROCID, NIL$, LAMBDAEXP.BODY,
        ENV.PAILRIS(LAMBDAEXP.PRAMS,
        ARGs.REST),
        none, none, new DOEVAL,
        new CLINK(none, none, none,
        none, none, none));

        APPLY := PROC;
        PROC.INTO(READYPool);
        PROCID := PROCID + 1;
      end
      else PROCERR;
    end
    otherwise PROCERR;
  end
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("NEW ");
  end
  end***PRINT***;
  end***NU***;
PRIMOP class UVAL;
begin
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    ref (ENVIR) TENV;
    TENV := ARGS.HEAD qua ENVIR;
    APPLY := TENV.ASSOC(ARGS.HEAD qua LITERAL);
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("EXTRACT");
  end;
end***UVAL***;

PRIMOP class TRANSACTION;
begin
  integer TR;
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    TR := TR+1;
    APPLY := new NUMBER(TR);
  end**APPLY***;
  procedure PRINT;
  begin
    outtext("TRANSACTION");
  end***PRINT***;
end***TRANSACTION***;

PRIMOP class CLEARQ;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    ARGS.HEAD qua COMMUNITY.CLEARCOMMUNITY;
    APPLY:-NIL$
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("CLEARQ ");
  end***PRINT***;
end***CLEARQ***;
PRIMOP class SELFDESTRUCT;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    ARGS.HEAD qua COMMUNITY.SUICIDE;
    APPLY:=-NIL$;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("SUICIDE ");
  end***PRINT***;
end***SELFDESTRUCT***;

PRIMOP class ISOLATE;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    ARGS.HEAD qua COMMUNITY.MAKEINCOMMUN;
    APPLY:=-NIL$;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("INCOMMUNICADO ");
  end***PRINT***;
end***ISOLATE***;

PRIMOP class ISOLATED;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    APPLY:=-ARGS.HEAD qua COMMUNITY.INCOMMUN;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("INCOMMUNICADO ");
  end***PRINT***;
end***ISOLATED***;
PRIMOP class UNISOLATE;
begin
  ref (CONSTANT) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    ARGS.HEAD qua COMMUNITY.COMMUNICATE;
    APPLY:-TRUE$;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("COMMUNICATE ");
  end***PRINT***;
end***UNISOLATE***;

PRIMOP class PRESENT;
begin
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
  begin
    ref (FORM) PTR;
    PTR := ARGS.REST.HEAD qua ENVIR.FIND(ARGS.HEAD);
    APPLY := if PTR == none then NIL$ else TRUE$;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("PRESENT ");
  end***PRINT***;
end***PRESENT***;
PRIMOP class COMMUNE;
begin
end ***COMMUNE***;

COMMUNE class PENDING;
begin
ref(FORM) procedure CREATE (ARGS);
text ARGS;
begin
ref(LIST) X;
X := PARSE(ARGS);
if X.HEAD == NIL$ then
begin
if X.REST.HEAD == NIL$ then
CREATE := new LIST(new PENDING, none)
else
CREATE := new LIST(new PENDINGA, X.REST)
end
else
if X.REST.HEAD == NIL$ then
CREATE := new LIST(new PENDINGF,
new LIST (X.HEAD, none))
else
CREATE := new LIST(new PENDINGAF ,X);
end ***CREATE***;
procedure PRINT;
begin
outtext("PENDING ");
end;
end ***PENDING***;

PENDING class PENDINGAF;
begin
ref(FORM) procedure APPLY(ARGS);
ref(LIST) ARGS;
begin
APPLY := if RUNNING.
PENDINGAF(ARGS.REST.HEAD,ARGS.HEAD)=/=none
THEN
TRUE$
else
NIL$
end ***APPLY***;
procedure PRINT;
begin
outtext("PENDING ");
end;
end ***PENDINGAF***;
PENDING class PENDINGF;
begin
  ref(FORM) procedure APPLY(ARGS);
  ref(LIST) ARGS;
begin
  APPLY := if RUNNING.PENDINGF(ARGS.HEAD) /= none then
            TRUE$
            else
            NIL$
end
end
end

PENDING class PENDINGA;
begin
  ref(FORM) procedure APPLY(ARGS);
  ref(LIST) ARGS;
begin
  APPLY := if RUNNING.PENDINGA(ARGS.HEAD) /= none then
            TRUE$
            else
            NIL$
end
end
end

PENDING class PENDINGG;
begin
  ref(FORM) procedure APPLY(ARGS);
  ref(LIST) ARGS;
begin
  APPLY := if RUNNING.PENDINGG /= none then
            TRUE$
            else
            NIL$
end
end
end

procedure PRINT;
begin
  outtext("PENDING ");
end;
end
end

end
end
end
COMMUNE class RECEIVE;
begin
ref (FORM) procedure CREATE (ARGS);
text ARGS;
begin
ref (LIST) X;
X := PARSE (ARGS);
if X.HEAD == NIL$ then
begin
if X.REST.HEAD == NIL$ then
CREATE := new LIST(new RECEIVE, none)
else CREATE := new LIST (new RECEIVEA, X.REST)
end
else if X.REST.HEAD == NIL$ then
CREATE := new LIST (new RECEIVED,
new LIST (X.HEAD, none))
else CREATE := new LIST (new RECEIVAF, X);
end ***CREATE***;
ref (FORM) procedure APPLY(ARGS);
ref (LIST) ARGS;
begin
ref (FROMHD) PTR;
PTR := RUNNING.PENDING;
APPLY := if PTR == none then WAIT
else PTR.RIGHTMOST.BOTERM.PICKFIRST;
end***APPLY***;
procedure PRINT;
begin
outtext("RECEIVE ");
end;
end***RECEIVE***;

COMMUNE class RECEIVED;
begin
ref (FORM) procedure APPLY(ARGS);
ref (LIST) ARGS;
begin
ref (FROMHD) PTR;
PTR := RUNNING.PENDINGP(ARGS.HEAD);
APPLY := if PTR == none then WAIT
else PTR.RIGHTMOST.PICKFIRST;
end***APPLY***;
procedure PRINT;
begin
outtext("RECEIVE ");
end***PRINT***;
end***RECEIVEE***;
COMMUNE class RECEIVEA;
begin
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
    ref (TRANHD) PTR;
    PTR := RUNNING.PENDINGA(ARGS.HEAD);
    APPLY := if PTR == none then WAIT
                else PTR.BOTTOM.PICKFIRST;
end***APPLY***;
procedure PRINT;
begin
  outtext("RECEIVE");
end***PRINT***;
end***RECEIVEA***;

COMMUNE class RECEIVEAF;
begin
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARGS;
begin
    ref (COMMUNIQUEUE) PTR;
    PTR := RUNNING.PENDINGA(ARGS.REST.HEAD qua NUMBER,ARGS.HEAD qua PROCESS);
    APPLY := if PTR == none then WAIT
               else PTR.PICKFIRST;
end***APPLY***;
procedure PRINT;
begin
  outtext("RECEIVE");
end***PRINT***;
end***RECEIVEAF***;
COMMUNE class SEND;
begin
  ref (FORM) procedure CREATE (ARGS);
  text ARG5;
  begin
    ref (LIST) X;
    X := PARSE (ARGS);
    if X.REST qua LIST.HEAD == NIL$ then
      X.REST qua LIST.HEAD := new NUMBER(0);
      CREATE := new LIST(new SEND, X);
    end ***CREATE***;
  ref (FORM) procedure APPLY(ARGS);
  ref (LIST) ARG5;
  begin
    if ARG5.HEAD qua COMMUNITY.INCOMMUN == TRUE$ then
      APPLY := - WAIT
    else begin
      ref (ENVIR) MES;
      ref (FROMHD) FPTR;
      ref (TRANHD) TPTR;
      ref (COMMUNIQUEUE) QPTR;
      ref (FORM) ABOUT;
      APPLY := TRUE$;
      ABOUT := ARG5.REST.HEAD;
      MES := ARG5.REST.REST.HEAD;
      MES := new ENVIR(FROM$, RUNNING, new
        ENVIR(About$, ABOUT, MES));
      QPTR := ARG5.HEAD qua
        COMMUNITY.PENDINGA(ABOUT, RUNNING);
      if QPTR /= none then
        QPTR.INSERT(MES)
      else begin
        FPTR := ARG5.HEAD qua
          COMMUNITY.PENDINGR(RUNNING);
        TPTR := ARG5.HEAD qua COMMUNITY.PENDINGA(ABOUT);
      end
    end
  end
end
if FPTR /= none then begin
  if TPTR /= none then ARGS.HEAD qua COMMUNITY.CREATEQ(RUNNING, ABOUT, MES, FPTR, TPTR)
  else ARGS.HEAD qua COMMUNITY.CREATEA(RUNNING, ABOUT, MES, FPTR);
end else
else if TPTR /= none then ARGS.HEAD qua COMMUNITY.CREATEF(RUNNING, ABOUT, MES, TPTR)
else ARGS.HEAD qua COMMUNITY.CREATEAF(RUNNING, ABOUT, MES);
end;
end;
end***APPLY***;
procedure PRINT;
begin
  outtext("SEND");
end***PRINT***;
end***SEND***;

PRIMOP class EXTANTP;
begin
  ref(FORM) procedure APPLY(ARGs);
  ref(LIST) ARGS;
  begin
    ref(ENVIR) TEMP;
    TEMP ::= ENV$.ASSOC(ARGS.HEAD qua LITERAL);
    APPLY ::= if TEMP == none then NIL$
    else
    if (not TEMP.APVAL is PROCESS) then NIL$
    else TEMP.APVAL qua PROCESS.EXTANT;
  end***APPLY***;
  procedure PRINT;
  begin
    outtext("EXTANTP ");
  end***PRINT***;
end***EXTANT***;
ATOM class LITERAL (STRING);
text STRING;
begin
  boolean procedure SAME (ID);
  ref (LITERAL) ID;
  begin
    SAME := STRING eq ID.STRING
  end***SAME***;
  procedure PRINT;
  begin
    outtext(STRING);
  end;
end***LITERAL***;

FORM class IF#(PREDICATE, THENPART, ELSEPART);
ref (FORM) PREDICATE, THENPART, ELSEPART;
begin
  ref (FORM) procedure CREATE (ARGS);
  text ARGS;
  begin
    ref(FORM) X;
    X := PARSE (ARGS);
    CREATE := new IF# (X qua LIST.HEAD, X qua LIST.REST qua
    LIST.HEAD, X qua
    LIST.REST qua LIST.REST qua LIST.HEAD);
  end ***CREATE***;
  procedure EVAL;
  begin
    CLINK$ := new CLINK (EXP$, ENV$, EVL$, UNL$, new
    BLOCK1, CLINK$);
    EXP$ := PREDICATE;
    PC$ := new DOEVAL;
  end***EVAL***;
  procedure PRINT;
  begin
    outtext("IF ");
    PREDICATE.PRINT;
    outimage;
    outtext("THEN ");
    outimage;
    THENPART.PRINT;
    outimage;
    outtext("ELSE ");
    outimage;
    ELSEPART.PRINT;
    outimage;
  end;
end***IF***;
FORM class QUOTE (ITEM);
ref (FORM) ITEM;
begin
ref (FORM) procedure CREATE (ARGS);
text ARGS;
begin
CREATE := new QUOTE (PARSE(ARGS) qua LIST.HEAD);
end ***CREATE****;
procedure EVAL;
begin
ACC$ := ITEM;
CLINK$.RESTORE;
end***EVAL***;
procedure PRINT;
begin
outtext("[QUOTE ");
ITEM.PRINT;
outtext("] ");
end;
end***QUOTE***;
FORM class LABELS (DEFS, FUNCALL);
ref (LIST) DEFS;
ref (FORM) FUNCALL;
begin
  ref (FORM) procedure CREATE (STRING);
text STRING;
begin
  ref (LIST) X;
  X:=-PARSE(STRING);
  CREATE:=-new LABELS (X.HEAD,X.REST.HEAD);
end ***CREATE***;
procedure EVAL;
begin
  ref (BETA) BETAPT;
  ref(LIST)LABLIST;
  ref(ENVIR) ENV, ENVTMP;
  ENV := new ENVIR(none,none,none);
  LABLIST := DEFS;
  ENVTMP := ENV$;
  while LABLIST /= none do
  begin
    BETAPT := new BETA (LABLIST.HEAD qua LIST.REST.HEAD qua
    LAMBDPA,ENV);
    ENVTMP := new ENVIR (LABLIST.HEAD qua LIST.HEAD qua
    LITERAL,BETAPT,ENVTMP);
    LABLIST := LABLIST.REST;
  end***WHILE***;
  ENVTMP. IDENT := ENVTMP. IDENT;
  ENVTMP. APVAL := ENVTMP. APVAL;
  ENVTMP. ENV := ENVTMP. ENV;
  ENV$ := ENV;
  EXP$ := FUNCALL;
  PC$ := new DOEVAL;
end***EVAL***;
procedure PRINT;
begin
  outtext("[LABELS ");
  outimage;
  DEFS.PRINT;
  outimage;
  FUNCALL.PRINT;
  outtext("]");
  outimage;
end;
end***LABELS***;
FORM class CATCH (IDENT, FUNCALL);
ref (LITERAL) IDENT;
ref (FORM) FUNCALL;
begin
  procedure EVAL;
  begin
    ENV$ := new ENVIR (IDENT,new DELTA(CLINK$),ENV$);
    EXP$ := FUNCALL;
    PC$ := new DOEVAL;
  end***EVAL***;
  procedure PRINT;
  begin
    outtext("[CATCH ");
    IDENT.PRINT;
    outimage;
    FUNCALL.PRINT;
    outtext("]");
    outimage;
  end;
end***CATCH***;

FORM class LAMBDA (PARAMS, BODY);
ref (FORM) PARAMS;
ref (FORM) BODY;
begin
  ref (FORM) procedure CREATE (ARGS);
  text ARGS;
  begin
    ref (LIST) X;
    X := PARSE (ARGS);
    CREATE := new LAMBDA (X.HEAD,X.REST.HEAD);
  end ***CREATE***;
  procedure EVAL;
  begin
    ACC$ := new BETA (this LAMBDA,ENV$);
    CLINK$.RESTORE;
  end***EVAL***;
  procedure PRINT;
  begin
    outtext("[LAMBDA ");
    if PARAMS =/= none then
      PARAMS.PRINT
    else
      outtext("NIL");
      outimage;
      BODY.PRINT;
      outtext("]");
    end;
  end***LAMBDA***;
FORM class BETA (LAMBDAEXP, ENV);
ref (ENVIR) ENV;
ref (LAMBDAD) LAMBDAEXP;
begin
  procedure EVLIS;
  begin
    ENV$ := ENV.PAIRLIS(LAMBDAEXP.PRAMS, EVL$. REST);
    EXP$ := LAMBDAEXP.BODY;
    PC$ := new DOEVAL;
  end***EVLIS***;
  procedure PRINT;
  begin
    outtext("[BETA ");
    outimage;
    LAMBDAEXP.PRINT;
    if ENV /= none then
      ENV.PRINT
    else
      outtext("**NONE**");
      outtext(""");
      outimage;
  end***PRINT***;
end***BETA***;

FORM class DELTA (LINK);
ref (CLINK) LINK;
begin
  procedure EVLIS;
  begin
    CLINK$ := LINK;
    ACC$ := EVL$. REST.HEAD;
    CLINK$. RESTORE;
  end***EVLIS***;
  procedure PRINT;
  begin
    outtext("(DELTA ");
    LINK.PRINT;
    outtext(""");
  end***PRINT***;
end***DELTA***;
FORM class ENVIR (IDENT, APVAL, ENV);
ref (LITERAL) IDENT;
ref (FORM) APVAL;
ref (ENVIR) ENV;
begin
ref (ENVIR) procedure PAIRLIST (FPARALIST, APARALIST);
ref (FORM) APARALIST, FPARALIST;
begin
ref (ENVIR) ENV;
ENV :- this ENVIR;
while FPARALIST =/= NIL$ and FPARALIST =/= none do
begin
ENV :- new ENVIR (FPARALIST qua LIST.HEAD, APARALIST qua LIST.HEAD, ENV);
APARALIST :- APARALIST qua LIST.REST;
FPARALIST :- FPARALIST qua LIST.REST;
end**WHILE**;
PAILRIS :- ENV;
end**PAIRLIST**;
ref (ENVIR) procedure FIND (ID);
ref (LITERAL) ID;
begin
ref (ENVIR) TENV;
TENV :- this ENVIR;
FIND :- none;
while (if TENV.IDENT =/= none then (not (ID.SAME (TENV.IDENT))) else false) do
TENV :- TENV.ENV;
if (TENV.IDENT =/= none) then FIND :- TENV;
end**FIND**;
ref (FORM) procedure ASSOC (ID);
ref (LITERAL) ID;
begin
ref (ENVIR) TMP;
TMP :- this ENVIR.FIND (ID);
if TMP =/= none then
ASSOC :- TMP.APVAL
else begin
outtext (" UNBOUND VARIABLE ");
ID.PRINT;
outimage;
end;
end**ASSOC**;
procedure ASSIGN(FPARAM,APARAM);
ref(LITERAL) FPARAM;
ref(FORM) APARAM;
begin
  ref(ENVI R) TENV;
  TENV :- this ENVI R;
  while
    (if TENV.IDENT =/= none
      then not TENV.IDENT.SAME(FPARAM)
      else false)
  do TENV :- TENV.ENV;
  if (if TENV.IDENT =/= none then TENV.IDENT.SAME(FPARAM)
    else false) then TENV.APVAL :- APARAM
  else
    begin
      TENV.IDENT :- FPARAM;
      TENV.APVAL :- APARAM;
      TENV.ENV :- new ENVI R(none,none,none);
    end;
end***ASSIGN***;
procedure PRINT;
begin
  ref (ENVI R) TEMP;
  outtext ("[");
  TEMP :- this ENVI R;
  outtext ("(");
  IDENT.PRINT;
  APVAL.PRINT;
  outtext (")");
  outimage;
  if TEMP.ENV =/= none then
    begin
      TEMP :- TEMP.ENV;
      TEMP.PRINT;
    end;
  outtext ("]");
end***PRINT***;
end***ENVI R***;
class CLINK(exp, ENV, EVL, UNL, PCNTR, LINK);
ref(FORM) exp;
ref(ENVIR) ENV;
ref(LIST) EVL, UNL;
ref(PC) PCNTR;
ref(CLINK) LINK;
begin
  procedure RESTORE;
  begin
    EXP$ := exp;
    ENV$ := ENV;
    EVL$ := EVL;
    UNL$ := UNL;
    PC$ := PCNTR;
    CLINK$ := LINK
  end***RESTORE***;
  procedure PRINT;
  begin
    outtext ("[CLINK]");
    if exp /= none then
      exp.PRINT
    else
      outtext("**NONE**");
      outimage;
      if ENV /= none then
        ENV.PRINT
      else
        outtext("**NONE**");
        outimage;
        if EVL /= none then
          EVL.PRINT
        else
          outtext("**NONE**");
          outimage;
          if UNL /= none then
            UNL.PRINT
          else
            outtext("**NONE**");
            outimage;
            if PCNTR /= none then
              PCNTR.PRINT
            else
              outtext("**NONE**");
              outimage;
              if LINK /= none then
                LINK.PRINT
              else
                outtext("**NONE**");
                outimage;
                end***PRINT***;
  end***CLINK***;
class PC;
virtual: procedure PRINT;
begin
end***PC***;

PC class BLOCK1;
begin
  procedure PRINT;
  begin
    outtext(" ==>BLOCK1 ");
    outimage;
  end;
  detach;
  if ACC$ =/= NIL$ then
    EXP$: - EXP$ qua IF#.THENPART
  else
    EXP$: - EXP$ qua IF#.ELSEPART;
    PC$: - new DOEVAL
end***BLOCK1***;

PC class DOEVAL;
begin
  procedure PRINT;
  begin
    outtext(" ==>DOEVAL==> ");
    outimage;
  end;
  detach;
  EXP$.EVAL;
end***DOEVAL***;
PC class BLOCK2;
begin
  procedure PRINT;
  begin
    outtext("==BLOCK2==>");
    outimage;
  end;
  detach;
  EVL$ := if EVL$ /= none
  then EVL$.ADDONEND(ACC$)
  else new LIST(ACC$, none);
  UNL$ := UNL$.REST;
  PC$ := new DOEVLIS
end***BLOCK2***;

PC class DOEVLIS;
begin
  procedure PRINT;
  begin
    outtext("==DOEVLIS==>");
    outimage;
  end;
  detach;
  if UNL$ /= none
  then UNL$.EVLIS
  else EVL$.HEAD.EVLIS;
end***DOEVLIS***;
ref (CONSTANT) procedure WAIT;
begin
  CLINK$ := new CLINK(EXP$, ENV$, EVL$,
                      UNL$, new DOEVLIS, CLINK$);
  WAIT := NIL$;
end***WAIT***;

FORM class LINKAGE;
begin
  ref(LINKAGE) SUCC, PRED;
  SUCC := this LINKAGE;
  PRED := this LINKAGE;
end***LINKAGE***;

LINKAGE class LINK;
begin
  procedure OUT;
  begin
    SUCC.PRED := PRED;
    PRED.SUCC := SUCC;
    SUCC := this LINKAGE;
    PRED := this LINKAGE;
  end***OUT***;
  procedure INTO(H);
  ref(LINKAGE) H;
  begin
    OUT;
    SUCC := H;
    PRED := H.PRED;
    H.PRED.SUCC := this LINKAGE;
    H.PRED := this LINKAGE;
  end***PRECEDE***;
end***LINK***;

LINKAGE class HEAD;
begin
  ref(LINK) procedure FIRST;
  FIRST := if SUCC in LINK then SUCC else none;
  integer procedure CARDINAL;
  begin
    integer I;
    ref(LINK) X;
    X := FIRST;
    while X=/=none do
      begin
        X := if X.SUCC in LINK then X.SUCC else none;
        I := I+1;
      end;
    CARDINAL := I;
  end***CARDINAL***;
end***HEAD***;
FORM class COMMUNIQUE( FROM, ABOUT, UP, DOWN, LEFT, RIGHT, QUEUE);
ref(PROCESS) FROM;
ref(NUMBER) ABOUT;
ref(COMMUNIQUE) UP, DOWN, RIGHT, LEFT;
ref(LIST) QUEUE;
begin
procedures INSERT(MESSAGE);
ref(ENVIR) MESSAGE;
begin
  QUEUE := QUEUE. ADDONEND(MESSAGE);
end
procedures REMOVEQ;
begin
  if LEFT /= none then LEFT.RIGHT := RIGHT;
  if RIGHT /= none then RIGHT.LEFT := LEFT;
  if UP /= none then UP.DOWN := DOWN;
  if DOWN /= none then DOWN.UP := UP;
  if UP is TRANHD and UP.DOWN == none then
    UP qua TRANHD.REMOVE;
  if LEFT is FROMHD and LEFT.RIGHT == none then
    LEFT qua FROMHD.REMOVE;
end
procedures PICKFIRST;
begin
  PICKFIRST := QUEUE.HEAD;
  QUEUE := QUEUE.REST;
  if QUEUE == none then REMOVEQ;
end
procedures RIGHTMOST;
BEGIN
  RIGHTMOST := if RIGHT == none then this COMMUNIQUE
else
  RIGHT.RIGHTMOST;
end
procedures BOTTOM;
BEGIN
  BOTTOM := if DOWN == none then this COMMUNIQUE
else
  DOWN.BELOW;
end;
COMMUNIQUE class TRANHD;
begin
  procedure REMOVETR;
  begin
    if LEFT /= none then LEFT.RIGHT := RIGHT
    else RUNNING.FIRSTTRANHD := RIGHT;
    if RIGHT /= none then RIGHT.LEFT := LEFT;
  end***REMOVETR***;
end***TRANHD***;

COMMUNIQUE class FROMHD;
begin
  procedure REMOVEFR;
  begin
    if UP /= none then UP.DOWN := DOWN
    else RUNNING.FIRSTFROMHD := DOWN;
    if DOWN /= none then DOWN.UP := UP;
  end***REMOVEFR***;
end***FROMHD***;
LINK   class COMMUNITY;
begin
  ref(CONSTANT) EXTANT, INCOMMUN;
  ref(FORM) FIRSTTRANHD, FIRSTFROMHD;
  procedure CREATEAP(FROM, ABOUT, MESSAGE);
  ref (PROCESS) FROM;
  ref (NUMBER) ABOUT;
  ref (ENVIR) MESSAGE;
begin
    ref (FROMHD) FH;
    ref (TRANHD) TH;
    ref (COMMUNIQUEUE) CQ;
    FH := new 
      FROMHD(FROM, none, none, FIRSTFROMHD, none, none, none);
    TH := new 
      TRANHD(none, ABOUT, none, none, none, FIRSTTRANHD, none);
    CQ := new COMMUNIQUEUE(FROM, ABOUT, TH, none, FH, 
      none, new LIST(MESSAGE, none));
    FH.RIGHT := CQ;
    TH.DOWN := CQ;
    FIRSTFROMHD := FH;
    FIRSTTRANHD := TH;
end***CREATEAP***;
procedure CREATEQ(FROM, ABOUT, MESSAGE, FRPRT, TRPRT);
ref (PROCESS) FROM;
ref (NUMBER) ABOUT;
ref (FROMHD) FRPRT;
ref (TRANHD) TRPRT;
ref (ENVIR) MESSAGE;
begin
  ref (COMMUNIQUEUE) CQ;
  CQ := new 
    COMMUNIQUEUE(FROM, ABOUT, TRPRT, TRPRT.DOWN, FRPRT, 
      FRPRT.RIGHT, new LIST(MESSAGE, none));
  TRPRT.DOWN := CQ;
  FRPRT.RIGHT := CQ;
  CQ.RIGHT.LEFT := CQ;
  CQ.DOWN.UP := CQ;
end***CREATEQ***;
procedure CREATEA(FROM, ABOUT, MESSAGE, FRPRTR);
ref (NUMBER) ABOUT;
ref (PROCESS) FROM;
ref (ENVIR) MESSAGE;
ref (FROMHD) FRPRTR;
begin
  ref (TRANHD) TH;
  ref (COMMUNIQUE) CQ;
  TH := new
  TRANHD none,ABOUT none,none,none,none,FIRSTTRANHD none;
  FIRSTTRANHD := TH;
  TH.RIGHT.LEFT := TH;
  CQ := new
  COMMUNIQUE FROM,ABOUT,TH none,FRPRTR,FRPRTR.RIGHT,
  new LIST MESSAGE none);
  FRPRTR := CQ;
  TH.DOWN := CQ;
  CQ.RIGHT.LEFT := CQ;
end***CREATEA***;
procedure CREATEx(FROM, ABOUT, MESSAGE, TRPRTR);
ref (PROCESS) FROM;
ref (NUMBER) ABOUT;
ref (ENVIR) MESSAGE;
ref (TRANHD) TRPRTR;
begin
  ref (FROMHD) FH;
  ref (COMMUNIQUE) CQ;
  FH := new
  FROMHD FROM none,none,none,FIRSTFROMHD none,none,none,none;
  FIRSTFROMHD := FH;
  FH.DOWN.UP := FH;
  CQ := new
  COMMUNIQUE FROM,ABOUT,TRPRTR,TRPRTR.DOWN,FH none,new
  LIST MESSAGE none);
  FH.RIGHT := CQ;
  TRPRTR.DOWN := CQ;
  CQ.DOWN.UP := CQ;
end***CREATEEx***;
ref (COMMUNIQUE) procedure PENDINGAF(About, FROM);
ref (NUMBER) ABOUT;
ref (PROCESS) FROM;
begin
  ref (FROMHD) FH;
  ref (COMMUNIQUE) CQ;
  FH := FIRSTFROMHD;
  while (if FH == none then false else FH.FROM /= FROM)
  do
    FH := FH.DOWN;
    CQ := FH;
    while (if CQ == none then false else
      (if CQ.ABOUT /= none then
        CQ.ABOUT qua NUMBER.VAL ne ABOUT qua NUMBER.VAL
      else
        true))
  do
    CQ := CQ.RIGHT;
    PENDINGAF := CQ;
end***PENDINGAF***;

ref (TRANHD) procedure PENDINGA(About);
ref (NUMBER) ABOUT;
begin
  ref (TRANHD) TH;
  TH := FIRSTTRANHD;
  while (if TH == none then false else
    (if TH.ABOUT /= none then
      TH.ABOUT qua NUMBER.VAL ne
    ABOUT qua NUMBER.VAL else true))
  do
    TH := TH.RIGHT;
    PENDINGA := TH;
end***PENDINGA***;
ref (FROMHD) procedure PENDING(FROM);
ref (PROCESS) FROM;
begin
  ref (FROMHD) FH;
  FH := FIRSTFROMHD;
  while (if FH == none then false else FH.FROM /= FROM)
    do
      FH := FH.DOWN;
      PENDING := FH;
  end***PENDING***;

ref (FROMHD) procedure PENDING;
begin
  PENDING := FIRSTFROMHD;
end***PENDING***;

procedure MAKEINCOMMUN;
begin
  INCOMMUN := TRUE$;
end***MAKEINCOMMUN***;

procedure COMMUNICATE;
begin
  INCOMMUN := NIL$;
end***COMMUNED***;

procedure CLEARCOMMUNITY;
begin
  FIRSTTRANHD := none;
  FIRSTFROMHD := none;
end***CLEARCOMMUNITY***;

procedure SUICIDE;
begin
  this COMMUNITY.OUT;
  EXTANT := NIL$;
end***SUICIDE***;
EXTANT := TRUE$;
INCOMMUN := NIL$;
end***COMMUNITY***;
HEAD class JOBQUE(exp,ENV);
ref(LIST) exp;
ref(ENVIR) ENV;
begin
  ref(PROCESS) procedure SELECT;
  begin
    ref(PROCESS) CURRENT;
    integer RANDNUM,I;
    RANDNUM := randint(1, this JOBQUE.CARDINAL, U);
    CURRENT := FIRST qua PROCESS;
    for I := 2 step 1 until RANDNUM do
      CURRENT := CURRENT.SUCO qua PROCESS;
    SELECT := CURRENT;
  end***SELECT***;
ref(LIST) CREATEJOBQUE;
ref(PROCESS) TPROCESS;
CREATEJOBQUE := exp;
PROCID := Ø;
while CREATEJOBQUE /= none do
begin
  TPROCESS := new PROCESS(PROCID, NILS, CREATEJOBQUE.HEAD, ENV, none, none, new DOEVAL,
                          new CLINK(none, none, none, none, none, none, none));
  TPROCESS. INTO(this JOBQUE);
  PROCID := PROCID + 1;
  CREATEJOBQUE := CREATEJOBQUE.REST;
end***WHILE***;
end***JOBQUE***;
COMMUNITY class
PROCESS(PNAME, HLDACC, HLDEXP, HLDENV, HLDEVL, HLDUNL, HLDPCC, HLDLINK);
integer PNAME;
ref(FORM) HLDACC;
ref(FORM) HLDEXP;
ref(ENVIR) HLDENV;
ref(LIST) HLDEVL;
ref(LIST) HLDUNL;
ref(PC) HLDPCC;
ref(CLINK) HLDLINK;
begin
procedure RENEw;
begin
    ACC$: = HLDACC;
    EXP$: = HLDEXP;
    ENV$: = HLDENV;
    EVL$: = HLDEVL;
    UNL$: = HLDUNL;
    PC$: = HLDPCC;
    CLINK$: = HLDLINK;
end***RENEW***;
procedure SUSPEND;
begin
    HLDACC := ACC$;
    HLDEXP := EXP$;
    HLDENV := ENV$;
    HLDEVL := EVL$;
    HLDUNL := UNL$;
    HLDPCC := PC$;
    HLDLINK := CLINK$;
end***SUSPEND***;
procedure PRINT;
begin
    outimage;
    outtext("PROCESS # ");
    outint(PNAME,3);
    outtext(" EVALUATED TO ");
    ACC$.PRINT;
    outimage;
end;
end***PROCESS***;
text procedure CDRSTRING(STRING);
  text STRING;
begin
  character TEMP;
  integer PLACE;
  PLACE := 1;
  STRING.setpos(2);
  TEMP := STRING.getchar;
  while TEMP = ' ' do
    begin
      TEMP := STRING.getchar;
      PLACE := PLACE + 1;
    end;
  CDRSTRING := blanks(1000);
  CDRSTRING := if TEMP eq '(' or TEMP eq '<' or TEMP eq '['
    then CSLIST(STRING.sub(PLACE,(STRING.length-PLACE)+1))
    else if TEMP eq ')' or TEMP eq '>' or TEMP eq ']
    then notext
    else CSATOM(STRING.sub(PLACE,STRING.length-PLACE+1));
  STRING.setpos(1);
end***CDRSTRING***;


text procedure CSATOM(STRING);
  text STRING;
begin
  text TEMP;
  character CURCHAR;
  STRING.setpos(2);
  CURCHAR := STRING.getchar;
  while CURCHAR ne ' ' and CURCHAR ne '.'
      and CURCHAR ne ')' and CURCHAR ne '>'
      and CURCHAR ne ']' and CURCHAR ne '('.
      and CURCHAR ne '[' and CURCHAR ne '<'
    do
    begin
      CURCHAR := STRING.getchar;
    end;
  if CURCHAR ne ')' and CURCHAR ne '>' and CURCHAR ne ']'
    then while CURCHAR eq ' ' or CURCHAR eq ':' or CURCHAR eq ',
        do CURCHAR := STRING.getchar;
  TEMP := copy
    (STRING.sub(STRING.pos-2,STRING.length-STRING.pos+3));
  TEMP.setpos(1);
  TEMP.putchar('(');
  CSATOM := TEMP;
end***CSATOM***;
text procedure CSLIST(STRING);
text STRING;
begin
  text TEMP;
  integer PARENCOUNT;
  character CURCHAR;
  STRING.setpos(3);
  PARENCOUNT := 1;
  while PARENCOUNT ne 0 do
    begin
      CURCHAR := STRING.getchar;
      PARENCOUNT :=
      if CURCHAR eq '(' or CURCHAR eq '<' or CURCHAR eq '['
        then PARENCOUNT + 1
      else if CURCHAR eq ')') or CURCHAR eq '>'
        then PARENCOUNT - 1
      else PARENCOUNT;
    end***WHILE***;
    CURCHAR := STRING.getchar;
    while CURCHAR = ',' or CURCHAR = ',' or CURCHAR = '.'
      do CURCHAR := STRING.getchar;
    TEMP :=
      copy(STRING.sub(STRING.pos-2,STRING.length-STRING.pos+3));
    TEMP.setpos(1);
    TEMP.putchar('(');
    CSLIST := TEMP;
  end***CSLIST***;
end


text procedure GRABFIRST(STRING);
text STRING;
begin
  integer PLACE;
  character STORE;
  text TEMP;
  STRING.setpos(2);
  PLACE := 2;
  STORE := STRING.getchar;
  while STORE = ' ' do
    begin
      STORE := STRING.getchar;
      PLACE := PLACE+1;
    end;
  TEMP :=
    copy (STRING.sub(PLACE-1,STRING.length-PLACE+2));
  TEMP.setpos(1);
  TEMP.putchar('(');
  GRABFIRST := if STORE = '(' or STORE = '<' or STORE = '['
    then GFLIST(TEMP)
    else if STORE = ')' or STORE = '>' or STORE = ']' then
      notext
    else GFATOM(TEMP);
  end ***GRABFIRST***;
text procedure GFATOM(STRING);
text STRING;
begin
    character STORE;
    STRING.setpos(2);
    STORE := STRING.getchar;
    while STORE ne ' ' and STORE ne ',' and STORE ne '!' and STORE ne ':'
       and STORE ne ')'; and STORE ne '>' and STORE ne ']';
       and STORE ne '[' and STORE ne '(' and STORE ne '<'
do STORE := STRING.getchar;
    GFATOM := copy(STRING.sub(2,STRING.pos-3));
end ***GFATOM***;


text procedure GFLIST(STRING);
text STRING;
begin
    integer PARENCOUNT;
    character CURCHAR;
    STRING.setpos(3);
    PARENCOUNT := 1;
    while PARENCOUNT ne 0 do
        begin
            CURCHAR := STRING.getchar;
            PARENCOUNT :=
            if CURCHAR = '(' or CURCHAR = '<' or CURCHAR = '['
                then PARENCOUNT + 1
            else if CURCHAR = ')' or CURCHAR = '>' or CURCHAR = ']
               then PARENCOUNT - 1
            else PARENCOUNT;
        end ***WHILE***;
    GFLIST := copy(STRING.sub(2,STRING.pos-2));
end ***GFLIST***;
boolean procedure NUMBERP(STRING);
text STRING;
begin
  character TEMP;
  STRING.setpos(1);
  TEMP := STRING.getchar;
  if TEMP = '-' then
    NUMBERP := NUMBERP(STRING.sub(2, STRING.length))
  else
    NUMBERP := TEMP ge '0' and TEMP le '9';
end ***NUMBERP***;

boolean procedure LISTP(STRING);
text STRING;
begin
  character TEMP;
  STRING.setpos(1);
  TEMP := STRING.getchar;
  LISTP := TEMP eq '(' or TEMP eq '<' or TEMP eq '[';
end ***LISTP***;
procedure MAKESCANPAIRS;
begin
  ref(LIST) IDS, VALS;
  IDS := new LIST(new LITERAL(copy("GENESIS")), none);
  IDS := new LIST(new LITERAL(copy("EXTRACT")), IDS);
  IDS := new LIST(new LITERAL(copy("MESSAGE")), IDS);
  IDS := new LIST(new LITERAL(copy("SLOT")), IDS);
  IDS := new LIST(new LITERAL(copy("TRANSACTION")), IDS);
  IDS := new LIST(new LITERAL(copy("NEW")), IDS);
  IDS := new LIST(new LITERAL(copy("SEND")), IDS);
  IDS := new LIST(new LITERAL(copy("RECEIVE")), IDS);
  IDS := new LIST(new LITERAL(copy("PENDING")), IDS);
  IDS := new LIST(new LITERAL(copy("INCOMMUNICATION")), IDS);
  IDS := new LIST(new LITERAL(copy("INCOMMUNICATION")), IDS);
  IDS := new LIST(new LITERAL(copy("SUICIDE")), IDS);
  IDS := new LIST(new LITERAL(copy("CLEARQ")), IDS);
  IDS := new LIST(new LITERAL(copy("EXTANTP")), IDS);
  IDS := new LIST(new LITERAL(copy("PRESENTP")), IDS);
  IDS := new LIST(new LITERAL(copy("LABELS")), IDS);
  IDS := new LIST(new LITERAL(copy("CATCH")), IDS);
  IDS := new LIST(new LITERAL(copy("IF")), IDS);
  IDS := new LIST(new LITERAL(copy("IF")), IDS);
  IDS := new LIST(new LITERAL(copy("LAMBDA")), IDS);
  IDS := new LIST(new LITERAL(copy("CONS")), IDS);
  IDS := new LIST(new LITERAL(copy("CAR")), IDS);
  IDS := new LIST(new LITERAL(copy("CDR")), IDS);
  IDS := new LIST(new LITERAL(copy("PLUS")), IDS);
  IDS := new LIST(new LITERAL(copy("MINUS")), IDS);
  IDS := new LIST(new LITERAL(copy("GREATERP")), IDS);
  IDS := new LIST(new LITERAL(copy("NULL")), IDS);
  IDS := new LIST(new LITERAL(copy("EQ")), IDS);
  IDS := new LIST(new LITERAL(copy("QUOTE")), IDS);
  IDS := new LIST(new LITERAL(copy("SAME")), IDS);
  IDS := new LIST(new LITERAL(copy("ATOMP")), IDS);
  IDS := new LIST(new LITERAL(copy("SET")), IDS);
  IDS := new LIST(new LITERAL(copy("AND")), IDS);
  IDS := new LIST(new LITERAL(copy("OR")), IDS);
  IDS := new LIST(new LITERAL(copy("PRINT")), IDS);
  IDS := new LIST(new LITERAL(copy("NOT")), IDS);
  IDS := new LIST(new LITERAL(copy("COMMUNICATE")), IDS);
  IDS := new LIST(new LITERAL(copy("TRUE")), IDS);
  IDS := new LIST(new LITERAL(copy("NIL")), IDS);
  IDS := new LIST(new LITERAL(copy("ASSIGN")), IDS);
VALS := new LIST(new LABELS((none, none), none));
VALS := new LIST(new UVAL, VALS);
VALS := new LIST(new MESSAGE, VALS);
VALS := new LIST(new SLOT, VALS);
VALS := new LIST(new TRANSACTION, VALS);
VALS := new LIST(new NU, VALS);
VALS := new LIST(new SEND, VALS);
VALS := new LIST(new RECEIVE, VALS);
VALS := new LIST(new PENDING, VALS);
VALS := new LIST(new ISOLATED, VALS);
VALS := new LIST(new ISOLATE, VALS);
VALS := new LIST(new SELFDESTRUCT, VALS);
VALS := new LIST(new CLEARQ, VALS);
VALS := new LIST(new EXPTANTP, VALS);
VALS := new LIST(new PRESENT, VALS);
VALS := new LIST(new LABELS((none, none), VALS));
VALS := new LIST(new CATCH((none, none), VALS));
VALS := new LIST(new IF#((none, none, none), VALS));
VALS := new LIST(new LAMBDA((none, none), VALS));
VALS := new LIST(new CONS, VALS);
VALS := new LIST(new CAR, VALS);
VALS := new LIST(new CDR, VALS);
VALS := new LIST(new PLUS, VALS);
VALS := new LIST(new MINUS, VALS);
VALS := new LIST(new GREATERP, VALS);
VALS := new LIST(new NOTX, VALS);
VALS := new LIST(new EQX, VALS);
VALS := new LIST(new QUOTE((none), VALS));
VALS := new LIST(new SAMEX, VALS);
VALS := new LIST(new ATOMP, VALS);
VALS := new LIST(new ASSIGN, VALS);
VALS := new LIST(new ANDX, VALS);
VALS := new LIST(new ORX, VALS);
VALS := new LIST(new PRINTVAL, VALS);
VALS := new LIST(new NOTX, VALS);
VALS := new LIST(new UNISOLATE, VALS);
VALS := new LIST(TRUE$, VALS);
VALS := new LIST(NIL$, VALS);
VALS := new LIST(new ASSIGN, VALS);
ALIST := new ENVIR((none, none, none));
ALIST := ALIST.PAIRLIS((IDS, VALS));
end ***MAKE-SCAN-PAIRS***;
ref (FORM) procedure SCANASSOC(ATOM, PAIRS);
text ATOM;
ref (ENVIR) PAIRS;
begin
  ref (ENVIR) TMP;
  TMP := PAIRS.FIND(new LITERAL(ATOM));
  SCANASSOC := if TMP /= none
               then TMP.APVAL
               else none;
end ***SCANASSOC***;

ref (FORM) procedure PARSE (STRING);
text STRING;
begin
  text LOCAL, X;
  ref (FORM) Z;
  ref (LIST) W;
  LOCAL := copy(STRING);
  W := new LIST (none, none);
  while LOCAL ne notext do
    begin
      X := GRABFIRST(LOCAL);
      if X /= notext then
        begin
          if LISTP(X) then W := W.ADDONEND(PARSE(X))
          else
            begin
              Z := SCANASSOC(X, ALIST);
              if Z /= none then
                begin
                  W := W.ADDONEND(Z.CREATE(CDRSTRING(LOCAL)));
                  if (Z in COMMUNE) or (not(Z in PRINOP)) and
                     (not(Z in CONSTANT))
                    then LOCAL := copy("()")
                    end
                  else W := if NUMBERP(X) then W.ADDONEND(new
                                           NUMBER(X.getint))
                           else W.ADDONEND(new LITERAL(X));
                end;
            end;
      end;
      LOCAL := CDRSTRING(LOCAL);
end ***WHILE***;
if (Z is QUOTE) or (Z is IF#) or (Z is LAMBDA) or (Z is
LABELS) or
(Z is CATCH) or (Z in COMMUNE) then
  PARSE := W.REST.HEAD
else
  PARSE := W.REST;
end ***PARSE***;
text procedure READF;
begin
  ref (infile) INF;
text READSTRING, BUF, image;
integer PLACE, IP;
INF := new infile("INF DSK:*")
BUF := blanks(80);
INF.open(BUF);
READSTRING := blanks(10000);
image := BUF;
PLACE := 1;
while not INF.endfile do begin
  INF.inimage;
  IP := 1;
  image.setpos(1);
  while image.getchar = ' ' do IP := IP + 1;
  READSTRING.sub(PLACE, image.sub(IP,
                   image.strip.length-IP+1).length) :=
    image.sub(IP, image.strip.length-IP+1);
  PLACE := PLACE + image.strip.length - IP + 3;
end***WHILE***;
READF := copy(READSTRING.strip);
INF.close;
end***READF***;

ref (FORM) procedure SCANNER;
begin
  text STRING;
  STRING := READF;
  SCANNER := new LIST(PARSE(STRING), none);
end ***SCANNER***;
procedure DRIVER(exp,ENV);
ref (LIST) exp;
ref (ENVIR) ENV;
begin
  exp.PRINT;
  outimage;
  READYPOOL := new JOBOQUE(exp,ENV);
  while READYPOOL.CARDINAL > 0 do
  begin
    INTEGER RANDNUM;
    RANDNUM:=RANDINT(1,55,U);
    RUNNING := READYPOOL.SELECT;
    RUNNING.RENEW;
    while RANDNUM > 1 AND PC$ /= none do
    begin
      RESUME(PC$);
      RANDNUM:=RANDNUM-1
    end;
    if PC$ == none then
    begin
      RUNNING.OUT;
      RUNNING.PRINT;
    end;
    RUNNING.SUSPEND;
    end***WHILE***;
  end;
end;
ref (ENVIR) ALIST;
ref (FORM) ACC$, EXP$;
ref (ENVIR) ENV$;
ref (LIST) EVL$, UNL$;
ref (PC) PC$;
ref (CLINK) CLINK$;
integer PROCID;
ref (LITERAL) FROM$, ABOUT$;
ref (JOBQUE) READYPOOL;
ref (PROCESS) RUNNING;
ref (NIL) NIL$;
ref (TRUE#) TRUE$;
integer U;
TRUE$ := new TRUE$;
NIL$ := new NIL$;
FROM$ := new LITERAL(copy("FROM");
ABOUT$ := new LITERAL(copy("ABOUT");
outtext(" SPLITS V2.1 RELEASE 1.0");
outimage;
outimage;
MAKESCANPAIRS;
while true do
   DRIVER(SCANNER, new ENVIR(none, none, none));
end;