A PAL Program of the Blackboard Evaluator

Arthur Evans, Jr.

May 16, 1968

Attached is a listing of a PAL program that simulates the left-hand blackboard evaluator of 6.231. It includes the following:

- the applicative subset
- tuples
- assignment
- recursion of single functions
- simultaneous definition
- labels, in PAL's complete generality
- val and res

It does not include

- simultaneous assignment
- simultaneous recursion
- the operator jj

Adding simultaneous assignment is easy; adding jj is not too hard; and adding simultaneous recursion is much harder.

The attached listing, with pages numbered 1 to 21, contains five files, like this.

AA	page 1
BB	6
T7	8
CC	12
/LB02/	18

They are organized as follows:

AA PAL

This file contains definitions of the data structures needed, along with routines to operate on them. The order, roughly, is

- debugging
- control and stack
- environment
- memory
- codes used in control and stack
- cycle counter

dump

initialization routines

In general, the actual structure of the various data bases is known only to the routines in AA.

BB PAL

This file is best ignored. It contains a routine for printing values produced by the interpreter, and it contains various debug things.

T7 PAL

This file contains sample input to the evaluator. The comment shows the PAL program being simulated, and the rest of the file contains the definition of a tuple, D, which is the control structure.

CC PAL

This file does the work. After the definition of the function Apply-Closure, the rest of the file is imperative. The actual evaluator is on pages 13 to the middle of 16. Each label BRn processes a single control item, as indicated on page 3.

The code on page 16 labelled MAIN accepts a directive from the console and then carries it out. The possible directives are at the bottom of page 16.

/LB02/ PAL

This library file is used instead of /LB00/. It is shown here for completeness, but is not part of the evaluator.

```
//  Simulate the blackboard machine executing PAL.

//  Arthur Evans, Jr.  24 April 68
//  This file was last modified on 05/12/68 at 10:57 by Evans.


//  Some definitions for debugging.

def ERRORFXIT = nil  //  updated later to MAIN
and OLDjj = nil  //  hold jj evaluated in Report

def Report t =  //  error detected by the simulator
     Write 'ERROR:  ';  Write t;  Write '*n';
     OLDjj  :=  jj;  //  save 'j' for calling DDebug
     goto ERRORFXIT


//  Define the stack and the control, and the relevant selectors.

def S = nil          // the stack
and C = nil          // the control

//  The stack and the control have the same format:  Each is a
//  3-tuple, whose third element is a 3-tuple, etc.  The first
//  element of the tuple is the type of the item, and the second
//  element is relevant data (or nil).

//  The following three functions are selectors for the stack
//  and the control:

def CODE ST = ST 1  // select code
and DATA ST = ST 2  // select data
and LINK ST = ST 3  // select link


//  Routines for the stack and control.  OLDTOP is for debugging.

def OLDTOP = nil

def Pop ST =         // pop C or S and return old top element
     Null ST  ->  Report 'S or C unexpectedly empty'  !  dummy;
     OLDTOP  :=  CODE ST, DATA ST;
     ST  :=  LINK ST;  //  pop the thing
     $ OLDTOP

and Push (x, ST) =  //  push x into C or S
     ST  :=  x 1, x 2, $ ST

def Stack x =        //  push x into S
     Push(x, S)
```

```
//  Definitions for the ENVIRONMENT.

def E = nil            //  the environment
and CurE = 0           //  current environment level
and LastE = 0          //  last environment level used
and ENVstack = nil     //  stack of environment layers

//  An environment level is a 3-tuple, like this:
//        1  name (as a string)
//        2  value
//        3  index in E of next level to look in

def LookupinE Name =  //  look up Name in current environment
     f CurE
     where rec f x =
          x = 0   ->  Report('Can*'t find ', Name, ' in environment.')
        ! E x 1 = Name  ->  E x 2
        ! f(E x 3)

def EnterE(Name, Store, Link)  =
     E  :=  E aug (Name, $ Store, $ Link);
     LastE  :=  LastE + 1;
     ENVstack  :=  $ CurE, $ ENVstack;  //  remember current environment
     CurE  :=  LastE

and PopENV ()  =  //  pop the environment stack
     CurE, ENVstack  :=  ENVstack

and InitE ()  =     //  initialize the environment to empty
     E, CurE, LastE, ENVstack  :=  nil, 0, 0, nil

def RES = 'RES'  //  identifier for VAL and RES



// Definitions for the MEMORY.

def GUESSmark = 'GUESS'   //  mark for guesses, for Y
and GUESScount = 0  //  make all guess distinct

def M = 0, (GUESSmark, 0), nil  // the memory
and LastM = 0          //  the last cell used in the memory

def Store v  =       //  the routine for curly S
     LastM  :=  LastM + 1;
     M  :=  $ LastM, $ v, $ M;
     $ LastM

and Contents c  =   //  the routine for curly C
     let rec f x  =  c = x 1  ->  x 2  !  f(x 3)
     in
     let t  =  f M  //  look up c in the memory
     in
     CODE t = GUESSmark  //  did we find a guess?
```

```
    ->    Report 'GUESS found in memory'
    !     t  //  all OK, so return the value found

def Update(c, v)   = //  update cell c with value v
    M  :=  $c, $v, $M

and InitM ()   =  //  initialize the memory
    M, LastM, GUESScount  :=  (0, (GUESSmark, 0), nil), 0, 0

def GUESS ()  =  //  return the next guess
    GUESScount  :=  GUESScount + 1;
    GUESSmark, $ GUESScount



//  Codes used in the CONTROL of the machine:

//  name                  code        data

def mENV               =  1  //  n   (index in E)
and mVAR               =  2  //  print name, as a string
and mCON               =  3  //  value
and mBASIC             =  4  //  value
and mLAMDA             =  5  //  (body, (bv1, bv2, ...) )
and mGAMMA             =  6
and mBETA              =  7
and mDELTA             =  8  //  n, the subscript in D
and mTAU               =  9  //  n, the order of the tuple to make
and mCurlyS            = 10
and mCurlyC            = 11
and mUPDATE            = 12
and mSEMICOLON         = 13
and mRVGAMMA           = 14  //  n, number of operands (1 or 2)
and mY                 = 15
and mEQUALCOLON        = 16  //  reverse update, for recursion
and mBIGDELTA          = 17  //  (body, (L1,D1), (L2,D2), ...)
and mGC                = 18  //  (body, dump)
and mVAL               = 19
and mRES               = 20


//  Codes used in the STACK of the machine.

def ENVmark            =  '  ENV'  //  n, the index in E
and LVmark             =  '   LV'  //  n, memory location
and RVmark             =  '   RV'  //  value
and BASICmark          =  'BASIC'  //  value
and LAMDAmark          =  'LAMDA'  //  (body, (bv1, bv2, ...), env)
and TUPLEmark          =  'TUPLE'  //  n, memory location
and DUMMYmark          =  'DUMMY'
and PPmark             =  '   PP'  //  (n, line)   (n is index in D)

def PALcycles   =   0


//  Stuff to count cycles of the evaluator.
```

```
def IntCYCLES  =  0  //  total count of interpretation cycles
and MaxCYCLES = 100  //  max cycles permitted per evaluation
and StartCYCLES = 0  //  cycle count at beginning of an execution
and CurCYCLES  =  0  //  cycles since last execution of InitCY

def CountCYCLES ()  =  //  count interpretation cycles
     IntCYCLES  :=  IntCYCLES + 1;
     IntCYCLES  >  CurCYCLES + MaxCYCLES
     ->   Report 'Interpretation cycle limit exceeded.'
     !    dummy

def InitCY ()  =  //  initialize stuff for cycles
     StartCYCLES  :=  IntCYCLES;
     CurCYCLES  :=  IntCYCLES

and MoreCY ()  =
     CurCYCLES  :=  IntCYCLES

def LastDELTA  =  0  //  last delta encountered, for debugging


//  Definitions for the DUMP.

def Dump = nil  //  The dump, itself.
and DumpCount = 1  //  The next dump layer to be assigned.

//  The dump is a k-tuple, where k is the number of layers currently
//  in use.  Each layer is a 4-tuple:
//        C, S, ENVstack, CurE

def AddDump ()  =  //  add a layer to the dump
     Dump  :=  Dump aug ($C, $S, $ENVstack, $CurE);
     DumpCount  :=  DumpCount + 1

and RestoreDump n  =  //  restore things as of dump layer n
     let Du = Dump n
     in
     C  :=  Du 1;
     S  :=  Du 2;
     ENVstack  :=  Du 3;
     CurE  :=  Du 4

def InitDu ()  =  //  initialize the dump
     Dump := nil;
     DumpCount  :=  1


//  Initialization routines for stack, control and everything.

def InitC ()  =  //  initialize the control
     C  :=  nDELTA, 1, nil;
     LastDELTA  :=  C

and InitS ()  =  S := nil
```

```
def InitALL ()   =
     InitC nil;  //  control
     InitS nil;  //  the stack
     InitE nil;  //  the environment
     InitM nil;  //  the memory
     InitDu nil;  //  the dump
     InitCY nil;  //  the cycle counter
     CLDjj  :=  nil  //  jj as in Report

def EMPTY   = ''  // the empty string
and NEWLINE = '*n'
```

```
//  Part 2 of the Blackboard Evaluator:  Debug and print routines.

//  Last modified on 05/11/68 at 21:17 by Evans.

def
     rec PE v  =
          CODE v  =  TUPLEmark
          -> ( let t = Contents(DATA v)
               in
               Write '(';
               P(1, Order t);
               Write ')'
               where rec P(i, n) =
                    n = 0   ->  Write nil
                  ! i > n   ->  dummy
                  ! (   PE(Contents(t i));  P(i+1, n) )
           )
          !   Write (' ', DATA v, ' ')
     within
     IntPrint x  =
          Write NEWLINE;
          PE( Contents(DATA x) );
          Write '*n*n'


def BasicPrint x =  //  the routine called by code
     IntPrint x;
     LVmark, 0


def OLDloop = nil  //  This item is updated later to $ Loop.

def T, TA, TB, TC  =  nil, nil, nil, nil  //  some temps
and SA, SB  =  nil, nil   //  some more

def OLDSYSTEMERROR = $ SYSTEMERROR

def OLDSYS ()  =  //  set SYSTEMERROR as it originally was
     SYSTEMERROR  :=  OLDSYSTEMERROR

def SYSTEMERRORVALUE = nil  //  value returned by SYSTEMERROR

def MYSYSTEMERROR t  =  //  my version of SYSTEMERROR
     let j = jj  //  to call BDebug
     in
     let loopP = $ SYSTEMERROR
     in
     SYSTEMERRORVALUE  :=  t;
     SYSTEMERROR := (ll x.x);
     BDebug j;
```

```
      SYSTEMERROR := IOERR;
      $ SYSTEMERRORVALUE

def MYSYS ()   =
      SYSTEMERROR  :=  MYSYSTEMERROR

def Test = nil  //  updated later to a little test function


def Branch = nil  //  Updated later to a label tuple.
```

```
//   Generalized labels.
//   Last modified on 05/11/68 at 15:14 by Evans.

//   The PAL program:

//           let a, k, d = -1, nil, nil
//           in
//           let f x =
//                1:     a := a + 1;
//                       Print a;
//                m:       x < 0   ->   (k := d)
//                       ! x = 0   ->   (k := 1)
//                       ! (x := x - 3)
//           in
//
//           d := n;  f a;  goto k;
//           n: a

//   Operators:

def ADD x y  =   x + y
and SUB x y  =   x - y
and NEG x    =     - x
and LES x y  =   x < y
and EQU x y  =   x = y

def NIL  = nil

//   The control structure:

def D =

//   D  1:  GAMMA (LAMDA 2 (a k d)) S T3 S RVGAMMA1 NEG 1 S nil S nil

(mGAMMA, NIL,
(mLAMDA, (2, ('a', 'k', 'd')),
(mCurlyS, NIL,
(mTAU, 3,
(mCurlyS, NIL,
(mRVGAMMA, 1,
(mBASIC, NEG,
(mCON, 1,
(mCurlyS, NIL,
(mCON, NIL,
(mCurlyS, NIL,
(mCON, NIL,
 nil ))))))))))))
,

//   D  2:  GAMMA (LAMDA 3 f) S (LAMDA 8 x)
```

```
(mGAMMA, NIL,
(mLAMDA, (3, 't'),
(mCurlyS, NIL,
(mLAMDA, (8, 'x'),
 nil ))))
,

//  D  3:  (BIGDELTA 4 (n 7))

(mBIGDELTA, (4, ('n', 7)),
 nil )
,

//  D  4:  D5 ; := d C n

(mDELTA, 5,
(mSEMICOLON, NIL,
(mUPDATE, NIL,
(mVAR, 'd',
(mCurlyC, NIL,
(mVAR, 'n',
 nil ))))))
,

//  D  5:  D6 ; GAMMA C f a

(mDELTA, 6,
(mSEMICOLON, NIL,
(mGAMMA, NIL,
(mCurlyC, NIL,
(mVAR, 'f',
(mVAR, 'a',
 nil ))))))
,

//  D  6:  D7 ; GO C k

(mDELTA, 7,
(mSEMICOLON, NIL,
(mGO, NIL,
(mCurlyC, NIL,
(mVAR, 'k',
nil )))))
,

//  D  7:  a

(mVAR, 'a',
 nil )
,

//  D  8:  ( BIGDELTA 9 (1 9) (m 11) )

(mBIGDELTA, (9, ('1', 9), ('m', 11)),
 nil )
```

,

```
// D 9: D10 ; := a RVGAMMA + C a 1

(mDELTA, 10,
(mSEMICOLON, NIL,
(mUPDATE, NIL,
(mVAR, 'a',
(mRVGAMMA, 2,
(mBASIC, ADD,
(mCurlyC, NIL,
(mVAR, 'a',
(mCON, 1,
 nil )))))))))
,

// D 10: D11 ; PRINT a

(mDELTA, 11,
(mSEMICOLON, NIL,
(mGAMMA, NIL,
(mBASIC, BasicPrint,
(mVAR, 'a',
 nil )))))
,

// D 11: D12 D13 BETA RVGAMMA LES C x 0

(mDELTA, 12,
(mDELTA, 13,
(mBETA, NIL,
(mRVGAMMA, 2,
(mBASIC, LES,
(mCurlyC, NIL,
(mVAR, 'x',
(mCON, 0,
 nil ))))))))
,

// D 12: := k C d

(mUPDATE, NIL,
(mVAR, 'k',
(mCurlyC, NIL,
(mVAR, 'd',
 nil ))))
,

// D 13: D14 D15 BETA RVGAMMA EQU C x 0

(mDELTA, 14,
(mDELTA, 15,
(mBETA, NIL,
(mRVGAMMA, 2,
(mBASIC, EQU,
(mCurlyC, NIL,
```

```
(mVAR, 'x',
(mCON, 0,
 nil ))))))))
,

//  D 14:  := k C l

(mUPDATE, NIL,
(mVAR, 'k',
(mCurlyC, NIL,
(mVAR, 'l',
 nil ))))
,

//  D 15:  := x RVGAMMA - C x 3

(mUPDATE, NIL,
(mVAR, 'x',
(mRVGAMMA, 2,
(mBASIC, SUB,
(mCurlyC, NIL,
(mVAR, 'x',
(mCON, 3,
 nil )))))))
```

```
//  Part 3 of the Blackboard evaluator.

//  This section was last modified on 05/12/68 at 11:33 by Evans.

let ApplyClosure(Rator, Rand)  =  //  apply a lamda-closure
     let Epart  = ↓ Rator 3  //  environment in which closure formed
     in
     Apply(Rator 2, DATA Rand)  //  call the routine to do the work
     where rec Apply(F, X)  =  //  apply F to X
          Istuple F  //  check for multiple bv-part
          -> ( let y = Contents X  //  the tuple rand
               in
               CODE y = TUPLEmark
               -> ( let n = Order F  //  rator
                    and d = Contents(DATA y)  //  rand
                    in
                    n = Order d
                    -> ( Ap 1  //  apply tuple
                         where rec Ap k =
                              k > n  ->  dummy
                              ! (Apply(F k, d k);  Ap(k+1))
                       )
                    !  Report 'Conformality failure'
                  )
               !  Report 'Conformality failure.'
             )
          !  //  single bv-part
             (
               EnterE ( F, X, Epart );
               Stack( ENVmark, $CurE );
               Push( (mENV, $CurE), C );
               Epart := CurE
             )
in

//  ****************************************************************

//                    EXECUTION STARTS HERE!


Test :=
   ( f
     where f n =
          Loop := T1;
          goto CDLoop;
      T1: n  :=  n - 1;
          n > 0  ->  goto CDLoop  !
          Loop  :=  CDLoop;
          S
```

```
   );

CODEcop := Lcop;

Branch  :=  BR1,  BR2,  BR3,  BR4,  BR5,  BR6,  BR7,  BR8,  BR9,  BR10,
            BR11, BR12, BR13, BR14, BR15, BR16, BR17, BR18, BR19, BR20;

ERROREXIT  :=  MAIN;   //  exit from the routine Report
MYSYS nil;  //  establish my SYSTEMERROR

goto MAIN;


//  Here is the MAIN PROCESSING LOOP of the evaluator.

Loop:       Null C  ->  goto Done  !  dummy;   //  All done, so quit.
            CountCYCLES nil;  //  count interpretation cycles
            T  :=  Pop C;  //  get the next control item
            goto Branch(CODE T);  //  branch on its code

BR1:        //  environment marker
            TA  :=  Pop S;  //  the useful value in the stack
            TE  :=  Pop S;  //  the environment marker in the stack
            //  Now we have a validity check.
            (CODE TE = ENVmark)  //  TE must be an environment marker
            & (DATA TE = DATA T)  //  it must be same env as in control
            & (DATA TE = CurE  )  //  and same env as current
            & (CODE TA = LVmark)  //  value must be an LV
            ->  Stack(LVmark, DATA TA)  //  all OK
            !   Report 'Improper environment marker.';
            PopENV nil;  //  set CurE
            goto Loop;

BR2:        //  variable
            Stack(LVmark, LookupinE(DATA T));
            goto Loop;

BR3:        //  constant
            Stack(RVmark, DATA T);
            goto Loop;

BR4:        //  basic operator
            Stack (BASICmark, DATA T);
            goto Loop;

BR5:        //  lamda expression
            Stack( LAMDAmark, DATA T aug $ CurE );
            goto Loop;

BR6:        //  gamma
            T   :=  Pop S;  //  the rator
            TA  :=  Pop S;  //  the rand
            CODE T = LAMDAmark  //  check the type of the rator
            -> ( //  apply a lamda closure
                 ApplyClosure(DATA T, TA);  //  apply the closure
                 Push( (mDELTA, DATA T 1), C)
```

```
     )
   !
   CODE T = TUPLEmark
   -> ( //  apply a tuple
        TA := Contents(DATA TA);   //  rv of rand
        not( (CODE TA = RVmark)&(Isinteger (DATA TA) ))
             -> Report 'Tuple mis-applied.' !
        TB  :=  (Contents(DATA T)) (DATA TA);
        Stack(LVmark, ↓ TB)
     )
   !
   CODE T  =  BASICmark  //  apply a basic
   ->   Stack( (DATA T)↓TA)
   !  Report 'Improper MATCR for GAMMA.';
   goto Loop;

BR7:     //  beta -- conditional branch
         TA   :=  Pop S;  //  this is to be a truth value, so check
         not( (CODE TA = RVmark) & (Isboolean(DATA TA)) )
              ->  Report 'Non-boolean argument to conditional'  !
         TB   :=  Pop C;  //  the 'false' exit
         TC   :=  Pop C;  //  the 'true' exit
         Push( (DATA TA -> TC ! TB), C);
         goto Loop;

BR8:     //  delta
         LastDELTA   :=   DATA T;  //  item to load into control
         TA   :=  D LastDELTA;
         L8A:  //  copy the control into C
         Push( (TA 1, TA 2), C);
         TA   :=  TA 3;
         goto  Null TA -> Loop ! L8A;

BR9:     //  tau -- make a k-tuple
         TA, TB   :=  nil, DATA T;
         L9A: TB > C
              -> ( TA   :=   TA aug DATA(Pop S);
                   TB   :=   TB - 1;
                   goto L9A
                 )
               !
         Stack(TUPLEmark, Store TA);
         goto Loop;

BR10:    //  curly S
         Stack(LVmark, Store(Pop S));
         goto Loop;

BR11:    //  curly C
         Stack(Contents(DATA(Pop S)));
         goto Loop;

BR12:    //  update (:=)
         TA   :=  Pop S;  //  the left side
         TB   :=  Pop S;  //  the right side
         Update(DATA TA, TB);
```

```
          Stack(LVmark, C);
          goto Loop;

BR13:     //  semi-colon
          TA := Pop S;
          (CODE TA = LVmark) & (DATA TA = 0)
          -> goto Loop
          !  Report 'Improper sequence element.';

BR14:     //  RVGAMMA
          TA  :=  DATA(Pop S);  // the rator
          TA  :=  TA ( DATA(Pop S) );  //  apply to first rator
          DATA T = 2  ->  (TA  :=  TA(DATA(Pop S))) ! dummy;
          Stack(RVmark, ↓ TA);
          goto Loop;

BR15:     //  Y  --  make a closure recursive
          TA  :=  Pop S;  //  the closure to which we are applying Y
          Push ( (mEQUALCOLON, nil), C );
          Push ( (mCurlyC, nil), C );
          Push ( (mGAMMA, nil), C );
          TB  :=  Store(GUESS nil);
          Stack(LVmark, ↓ TB);
          Stack(LVmark, ↓ TB);
          Stack TA;
          goto Loop;

BR16:     //  equalcolon (=:)
          TA  :=  Pop S;  //  value to be stored
          TB  :=  Pop S;  //  the place to put it
          Update(DATA TB, TA);  //  update the memory
          Stack TA;  //  leave an answer
          goto Loop;

BR17:     //  BIGDELTA  --  define labels
          //  DATA has the form (n, (L1,D1), (L2,D2), ...)
          TA  :=  DATA T;
          TB  :=  Order TA;  //  number of labels to define (plus 1)
          L17A:     TB < 2  -> goto L17B  !  dummy;  //  out if done
                    TC  :=  Store(BPmark, (TA TB 2, ↓ DumpCount) );
                    EnterE( TA TB 1, TC, CurE);
                    Stack(ENVmark, ↓ CurE);
                    Push( (mENV, ↓ CurE), C );
                    TB  :=  TB - 1;
                    goto L17A;
          L17B:     AddDump nil;
                    Push( (mDELTA, TA 1), C );
                    goto Loop;

BR18:     //  goto (body, dump)
          TA  :=  Pop S;  //  place to go
          CODE TA = BPmark  //  check validity of target
          ->  dummy  //  it's OK
          !   Report 'Improper rand for goto.';
          RestoreDump(DATA TA 2);
          Push( (mDELTA, DATA TA 1), C);
```

```
          goto Loop;

ER19:     //  val
          TA := Pop S;  //  the item whose value we want
          TB := Store (BDmark, $ DumpCount);  //  a dump marker
          EnterE(RES, TB, CurE);  //  put RES into the environment
          Stack(ENVmark, &CurE);
          Push( (mENV, $ CurE), C );
          AddDump nil;
          Push( (mDELTA, DATA TA 1), C );
          goto Loop;

ER20:     //  res
          TA := Contents( DATA(Pop S) );  //  the dump
          TB := Pop S;  //  the result to return
          RestoreDump( DATA TA );
          Stack TB;
          goto Loop;


//  This ends the Blackboard Evaluator.


//  ****************************************************************

//  Here is the console reading loop...


MAIN:     SA  :=  CycleCount PALcycles;
          PALcycles  :=  CycleCount 0;
          Write ( 'R: ', SA, '*n*n');

          SA  :=  EMPTY;
M1:       SB  :=  Readch nil;
            SB = NEWLINE ->  goto M2
          ! SB =  '*s'   ->  goto M1
          ! dummy;
          SA  :=  SA %Conc SB;
          goto M1;

M2:         SA = EMPTY  ->  goto M1  !  dummy;
            Write 'W*n';
            SA = 'eval'   ->  goto START
          ! SA = 's'      ->  Intprint S
          ! SA = 'debug'  ->  Debug nil
          ! SA = 'jdebug' ->  BDebug OLDjj
          ! SA = 'more'   ->  (MoreCY nil;  goto Loop)
          ! SA = 'mysys'  ->  MYSYS nil
          ! SA = 'oldsys' ->  OLDSYS nil
          ! SA = 'cycle'  ->  Write(CycleCount 0, ' cycles.*n')
          ! SA = 'oloop'  ->  (Loop  :=  OLDLoop)
          ! SA = 'test'   ->  Test 1
          ! SA = 'quit'   ->  goto Quit
          ! Write '?*n';

          goto MAIN;
```

```
START:      //  Begin here to do an evaluation.
            InitALL nil;  //  Initialize C, S, E, M, and other things.
            goto Loop;

Done:       //  come here at the end of the evaluation
            write (IntCYCLES - StartCYCLES, ' interpretation cycles.*n');
            goto MAIN;

Quit:       write 'Good Bye.'
```

```
//  Art Evans' private library of PAL functions.

//  Last modified on 05/12/68 at 17:11 by Evans.

def Write x  =  //  write a tuple, without commas and parens at top
     Istuple x
     ->    W(1, Order x)
     !     Print x
     where rec W(i, n)   =
            n = 0   ->  Print nil
          ! i > n   ->  dummy
          ! (  Print(x i);  W(i+1, n)   )


def FINISH  =  //  call FINISH()  to terminate execution
     jj ( ll().nil )


def         //  convert a character to an integer
            Zero = 48  //  ASCII value for '0'
            within
     CtoI x  =  StoI x  +  Zero


def         //  DEBUG - written by Barkalow, typed by Evans.
            Chkind x  =  //  check the kind of x
                 let y = CtoI x  //  convert character to numeric
                 in
                   y>47 & y<58  ->  y-48  //  digit
                 ! y < 33   ->  10  //  control character (sp, tab, nl)
                 ! y = 59   ->  11  //  semicolon
                 ! y = 44   ->  11  //  comma
                 ! y = 40   ->  11  //  left parenthesis
                 ! y = 41   ->  11  //  right parenthesis
                 ! y = 46   ->  11  //  period
                 ! y = 39   ->  12  //  quote
                 ! -1               //  default
            within

            Ch = nil  //  the last scanned character
            and
            Symb, Val  =  nil, nil  //  last token scanned
            within

            Nextsymb()  =  //  read next symbol
             (   let N, Kind  =  0, 0  //  two temps
                 in

                 SpaceLOOP:  //  loop to here until a non-space
                 Kind  :=  Chkind Ch;  //  check kind of character
```

```
            Kind = 10  //  space, tab or newline
            ->    (Ch := Readch nil;  goto SpaceLOOP)  //  skip spaces
            !
            Kind = 11  //  ; , ( ) .
            ->    (Symb := Ch;  Ch := Readch nil)
            !
            Kind = 12  //  begin a quotation
            -> ( Symb, Val  :=  'C', '';
                 CHARLOOP:  //  the loop to scan a string
                 Ch  :=  Readch nil;  //  read next character
                   Ch = '*''  //  done with string
                 -> ( Ch := Readch nil;  goto RETURN)
                 ! Ch = '**'  //  process a special
                 -> ( Ch  :=  Readch nil;  //  read next
                      Ch  :=   Ch='t' -> '*t'
                             ! Ch='n' -> '*n'
                             ! Ch='s' -> '*s'
                             ! Ch='b' -> '*b'
                             ! Ch  //  default
                    )
                 !  dummy;  //  a character has been read
                 Val  :=  Val aConc Ch;  //  build up string
                 goto CHARLOOP
               )
            !  //  it must be an identifier
            (    Symb, Val  :=  'C', '';
                 IDLOOP:  //  build up a name or constant
                 Kind < 10
                 -> ( Val  :=  Val %Conc Ch;
                      Kind < 0
                      ->    (Symb  :=  'V')
                      !     (N  :=  10*N + Kind);
                      Ch  :=  Readch nil;
                      Kind  :=  Chkind Ch;
                      goto IDLOOP
                    )
                 !
                 Symb  =  'C'
                 ->    (Val  := N)
                 !
                 RETURN:  dummy
            )
        )
      within

   BDebug j  =  //  finally, the function being defined...
   let Lookup S  =  //  the function that looks up identifiers
         S = 'nil'  ->  nil  !  LookupinJ(S, j)

   in
   let rec  //  Rbasic, Rterm and Rexp are mutually recursive
   (    Rbasic ()  =  //  read an item
         val(let A =
                      Symb = 'V'  ->  Lookup Val
                    ! Symb = 'C'  ->  £ Val
                    ! Symb = '('  ->  (Nextsymb nil; Rexp nil)
```

```
                    ! res nil
                in
                Nextsymb nil;
                res A
              )
       and Rterm f   =
           Symb = 'V'   |  Symb = 'C'   |  Symb = '('
           ->    Rterm( f  (Rbasic nil) )
           !     f
       and Rexp  ()   =  // read an expression
           let A = nil
           in
           REXPLOOP:  //   ?
           A   :=   A aug Rterm(Rbasic nil);
           Symb = ','
           ->    (Nextsymb nil;  goto REXPLOOP)
           !
           Order A = 1
           ->    A 1  //   don't return a 1-tuple
           !     A
      )
      in

      //  Here execution begins for debug.
      Write 'Debug entered.*n*n';
      Ch   :=   Readch nil;  //   start things off
      LOOP:  //   debug's main loop
      Nextsymb nil;
      Write(Rexp nil, '*n*n');
      Symb = '.'
      ->    Write 'Program re-entered.*n'
      !     goto LOOP


def Debug ()   =  //   call debug, with no preparation
      BDebug jj


def UPDATE(x, y)   =  //   update x with y
      x   :=   y;
      x

and GOTO x   =  //   goto
      goto x


def ADebug j   =  //   call debug, preparing SYSTEMERROR first
      let rorrE = $ SYSTEMERROR   //   save a copy
      in
      SYSTEMERROR   :=   (ll x.x);  //   identity function
      BDebug j;
      SYSTEMERROR   :=   rorrE

def         //   funny business to update SYSTEMERROR
            f x   =  // the value SYSTEMERROR will have is f
                 let j = jj
```

```
                in
                Er 'System ';
                ADebug j;
                x  //  return value called with
            within
        BDebug  =
            SYSTEMERROR  :=  f;  //  update SYSTEMERROR
            EDebug  //  this definition is otherwise mandatory


    def CycleCount n  =  //  number of execution cycles
        LookupNo - n

    def LookupNo  =  //  protect it from tampering
        $ LookupNo

    def DoOver  =  //  go to DoOver to restart execution
        ( ll(). L: L )
```