```
//  Preliminary definitions for the evaluator.

//  Last modified on 04/19/69 at 18:47 by Evans.

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

//  Selectors and constructors for the stack and control.

def t(x, y) = x  // Top of stack or control.

and r(x, y) = y  // Rest of stack or control.

def r2 x = r(r x)  // Rest of (rest of (stack or control)).

and r3 x = r(r(r x))  // Rest of (rest of rest).

and 2d x = t(r x)  // Second element of stack or control.

def Push(x, s) = x, s  // Put new item onto stack or control.

def rec Prefix(x, y) =  // Put control x at top of control y.
        Null x -> y
      | Null y -> x
      | Push(t x, Prefix(r x, y) )


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

//  Selectors, predicates and constructors for lambda-expressions
//  and lambda-closures.

def LAMBDA = '_lambda'  // Tag for lambda-expressions and closures.

def Cons_lambda_exp(bV, Body)  =  //  Construct a lambda-expression.
     LAMBDA,  bV, Body

and Cons_closure(L_exp, Env)   =  //  Construct a lambda-closure.
     L_exp aug Env

def bV x = x 2       //  Select bv-part of a lambda-exp or closure.
and Body x = x 3     //  Select body part...
and Env x = x 4      //  Select environment part...

def       Test(x, n) =
                Istuple x
                -> Order x   eq  n
                   -> x 1  eq LAMBDA
                   |  false
                |  false
          within

     Is_lambda_exp x = Test(x, 3)
and  Is_closure x = Test(x, 4)
```

```
//  * * * * * * * * * * * * * * * * * * * * * * *

//  Tagger and tag-checkers for structures.

def Tag n s  =  s aug n  //  Tag structure s with tag n.

and Is_tag s n  =  //  Does structure s have tag n?
     Istuple s  ->  n eq s(Order s)  |  false

and Get_tag s  =  s(Order s)  //  Return the tag of s.

and Sons s  =  Order s - 1  //  Return number of sons of s.
```

```
//  Definitions and predicates for the right-hand evaluator.

//  Last modified on 04/26/69 at 12:49 by Evans.


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

//  Items and predicates for control structure and stack.

def GAMMA       = '_gamma'
and BETA        = '_beta'
and CONSTANT    = '_constant'
and BASIC       = '_basic'
and VARIABLE    = '_variable'
and AUG         = '_aug'
and TUPLE       = '_tuple'     // Used only in stack.
and ETA         = '_eta'       // Used in stack for recursion.

def       Test(x, y) =
             Istuple x
             -> Order x eq 2
                -> Isstring(x 1)
                   -> x 1 eq y
                   |  false
                |  false
             |  false
     within
     Is_constant x   = Test(x, CONSTANT)
 and Is_variable x   = Test(x, VARIABLE)
 and Is_eta x        = Test(x, ETA)
 and Is_basic x      = Test(x, BASIC)

 and Is_tuple x =
        Test(x, TUPLE)  ->  true  // Is it a constructed tuple?
     |  Test(x, CONSTANT)  ->  Null(x 2)  // Is it nil?
     |  false  // Neither.

 and Is_identifier x =  // Is x constant, variable or basic?
     Test(x, CONSTANT) or Test(x, VARIABLE) or Test(x, BASIC)


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

//  Call for YSTAR is produced in MakeControl for rec-defs.

def YSTAR  =
     VARIABLE, 'ystar'

and NIL  =
     CONSTANT, nil

def Is_YSTAR x =
     Isstring x -> x eq 'ystar' | false
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * *

//  Tags for abstract syntax tree.

def TEST      = '_test'      // test ... ifso ... ifnot ...
and ARROW     = '_arrow'     // ... -> ... | ...
and AP        = '_ap'        // functional application
and FN        = '_fn'        // lambda
and EQUAL     = '_equal'     // definition
and WITHIN    = '_within'
and REC       = '_rec'
and FF        = '_ff'        // function form definition
and AND       = '_and'       // 'and' definition
and COMMA     = '_comma'     // tuple maker
and LET       = '_let'
and WHERE     = '_where'
and BINOP     = '_binop'
and UNOP      = '_unop'
and PERCENT   = '_percent'


//  Taggers for tags in abstract syntax tree.

def TEST_ x y z      = Tag TEST (x, y, z)
and ARROW_ x y z     = Tag ARROW (x, y, z)
and AP_ x y          = Tag AP (x, y)
and FN_ x y          = Tag FN (x, y)
and LET_ x y         = Tag LET (x, y)
and WHERE_ x y       = Tag WHERE (x, y)
and EQUAL_ x y       = Tag EQUAL (x, y)
and WITHIN_ x y      = Tag WITHIN (x, y)
and REC_ x           = Tag REC (nil aug x)
and FF_ x y          = Tag FF (x, y)
and AUG_ x y         = Tag AUG (x, y)
and BINOP_ x y z     = Tag BINOP (x, y, z)
and UNOP_ x y        = Tag UNOP (x, y)
and PERCENT_ x y z   = Tag PERCENT (x, y, z)

//  AND_ and COMMA_ would have to be n-ary taggers, and hence
//  are not provided.


//  Taggers for standardized syntax tree.

def GAMMA_ x y       = Tag GAMMA (x, y)
and BETA_ x y z      = Tag BETA (x, y, z)
and LAMBDA_ x y      = Tag LAMBDA (x, y)


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

//  Some useful functions for Transform.

def Value_of x  =  // Evaluate a control element, to put it on stack.
    x
```

```
and Val_of x  =  // De-tag a stack element, to get its value.
    x 2

def Apply x y  =
    let t = (Val_of x) (Val_of y)
    in
    (Isfunction t -> BASIC | CONSTANT), t

and Aug x y  =  //  Augment x with y.
    Is_tuple x  ->  (TUPLE, Val_of x aug y)
    |  Error 'first argument of Aug not a tuple'


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

//                          E N V I R O N M E N T

//  The following function is used in applying a lambda-closure.
//  The names on the (possibly structured) bv-part 'Names' are
//  added to the environment 'Env', associated with the corres-
//  ponding part of 'Values'.  The new environment is returned as
//  the value of the function.

def rec Decompose(Names, Values, Env)  =
    test  Is_variable Names  // Is it a single variable?
    ifso  (Names 2, Values, Env)  // Yes, so add it to environment.
    ifnot  // Check conformality.
    test  Is_tuple Values
    ifnot Error 'conformality failure'  // Tuple applied to scalar.
    ifso
    test  Order Names eq Order (Val_of Values)
    ifnot Error 'conformality failure.'  // Differing tuple lengths.
    ifso  // Process a multiple-bv part.
      (  Q 1 Env
         where rec Q n e =
                 n > Order Names  ->  e
              |  Q (n+1) ( Decompose(Names n, (Val_of Values) n, e) )
      )



//  Define primitive environment, and provide function to look
//  up variables in the environment.

def PE =  // The primitive environment.
    'ystar', 'ystar',  // for recursion
    nil

and Lookup(Var, Env) =  // Look up a variable in the environment.
    let V = Var 2  // The variable identifier.
    in
    L Env  // Start looking in Env.
    where rec L e =
            Null e  ->  Error 'variable not found in environment'
```

```
| V eq e 1  ->  e 2  // Found.
| L (e 3)  // Keep looking.
```

```
//                                 MakeControl

//  The function MakeControl accepts as input an abstract syntax-
//  tree representation of a program, and produces as output a
//  control structure suitable for input to the evaluator.  It is
//  called by Interpret.

//  Last modified on 04/19/69 at 17:43 by Evans.

//  The structure of the definitions which follows is like this:

//  def
//            rec D x =  standardize definitions
//            within

//       rec ST x =  standardize a syntax tree
//

//  def rec FF =  flatten standardized syntax tree

//  def MakeControl P = FF( ST P, nil )


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


def XF_dbg = false  // Control debug printing from these routines.
//
```

```
def
     rec D x =  // Standardize a definition.
          let Type = Is_tag x
          in
             Type EQUAL  ->  x  // Already OK.
          |  Type WITHIN
               -> ( let u, v = D(x 1), D(x 2)
                    in
                    EQUAL_ (v 1) ( GAMMA_ (LAMBDA_ (u 1) (v 2)) (u 2) )
                  )
          |  Type REC
               -> ( let w = D(x 1)
                    in
                    EQUAL_ (w 1) ( GAMMA_ YSTAR (LAMBDA_ (w 1) (w 2)) )
                  )
          |  Type FF
               -> ( EQUAL_ (x 1 1) (Q (Order(x 1)) (x 2))
                    where rec Q k t =
                         k < 2  ->  t
                         |  Q (k-1) (LAMBDA_ (x 1 k) t)
                  )
          |  Type AND
               -> ( EQUAL_ L (Tag COMMA R)
                    where L, R = Q 1 nil nil
                    where rec Q k s t =
                         k > Sons x  ->  (s, t)
                         |  ( let w = D(x k)
                              in
                              Q (k+1) (s aug w 1) (t aug w 2)
                            )
                  )
          |  Error 'improper node found in D'
     within

rec ST x  =  // Standardize abstract syntax tree.
  if XF_dbg do Write( Get_tag x, '*n' );
  (
     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type BETA or Type TEST or Type ARROW
          ->  BETA_ (ST(x 1)) (ST(x 2)) (ST(x 3))
     |  Type LAMBDA or Type FN
          ->  LAMBDA_ (x 1) (ST(x 2))
     |  Type COMMA
          -> ( Q 1 NIL
               where rec Q k t =
                    k > Sons x  ->  t
                  |  Q (k+1) ( AUG_ t (ST(x k)) )
             )
     |  Type PERCENT
          -> GAMMA_ (x 2) ( AUG_ (AUG_ NIL (ST(x 1))) (ST(x 3)) )
     |  Type LET
          -> ( let w = D(x 1)  // Standardize the definition.
               in
```

```
               GAMMA_ ( LAMBDA_ (w 1) (ST(x 2)) ) (ST (w 2))
             )
      |  Type WHERE
           -> ( let w = D(x 2)  // Standardize the definition.
               in
               GAMMA_ ( LAMBDA_ (w 1) (ST(x 1)) ) (ST (w 2))
             )
      |  Type AP   ->  GAMMA_ (ST(x 1)) (ST(x 2))
      |  Type BINOP
           ->  GAMMA_ ( GAMMA_ (BASIC, x 3) (ST(x 1)) ) (ST(x 2))
      |  Type UNOP
           ->  GAMMA_ (BASIC, x 2) (ST(x 1))
      |  Sons x eq 1 -> Tag (Get_tag x) ( nil aug ST(x 1) )
      |  Sons x eq 2 -> Tag (Get_tag x) ( ST(x 1), ST(x 2) )
      |  Error 'improper node found in ST'
   )
//
```

```
//  The function FF flattens a standardized tree into a
//  control structure.

def rec FF(x, c)  =  // Flatten standardized tree x onto control c.
  if XF_dbg do Write( Get_tag x, '*n' );
  (
     let Type = Is_tag x
     in
        Is_identifier x -> (x, c)
     |  Type LAMBDA
          -> ( let Body = FF( x 2, nil )
               in
               Cons_lambda_exp(x 1, Body), c
             )
     |  Type BETA
          -> ( let TA = FF(x 2, nil)  // True arm.
               and FA = FF(x 3, nil)  // False arm.
               in
               FF( x 1, (BETA, (TA, (FA, c))) )
             )
     |  Sons x eq 2 -> FF( x 2, FF( x 1, (Get_tag x, c) ) )
     |  Sons x eq 1 -> FF( x 1, (Get_tag x, c) )
     |  Error 'improper node found in FF'
  )

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

def MakeControl P  =  // The routine that does all the work.
     FF ( ST P, nil )
```

```
// State transformations for the right-hand evaluator.
// Last modified on 04/26/69 at 12:54 by Evans.

def Sub_prob_exit(C, S, E, D) =
     D 1, Push(t S, D 2), D 3, D 4

and Eval_constant(C, S, E, D) =
     r C, Push(Value_of(t C), S), E, D

and Eval_basic(C, S, E, D) =
     r C, Push(Value_of(t C), S), E, D

and Eval_variable(C, S, E, D) =
     r C, Push( Lookup(t C, E), S ), E, D

and Eval_lambda_exp(C, S, E, D) =
     let New_S = Cons_closure(t C, E)
     in
     r C, Push(New_S, S), E, D

and Eval_conditional(C, S, E, D) =
     let New_C =  Val_of(t S) -> t(r C) | t(r2 C)
     in
     Prefix(New_C, r3 C), r S, E, D

and Extend_tuple(C, S, F, D) =
     let New_S = Aug (t S) (2d S)
     in
     r C, Push(New_S, r2 S), E, D

and Apply_closure(C, S, E, D) =
     let New_E = Decompose ( bV(t S), 2d S, Env(t S) )
     and New_D = r C, r2 S, E, D
     in
     Body(t S), nil, New_E, New_D

and Apply_basic(C, S, E, D) =
     let New_S = Apply (t S) (2d S)
     in
     r C, Push(New_S, r2 S), E, D

and Apply_tuple(C, S, E, D) =
     let New_S = Apply (t S) (2d S)
     in
     r C, Push(New_S, r2 S), E, D

and Apply_Y(C, S, E, D) =
     let t = ETA, 2d S
     in
     let New_S = Push(2d S,  Push(t, r2 S) )
     in
     C, New_S, E, D

and Apply_eta(C, S, E, D) =
     Push(GAMMA, C), Push(t S 2, S), E, D
```

```
//  Main program.  Transform does one step in the evaluation,
//  and Interpret is the driver for it.

//  Last modified on 04/26/69 at 13:55 by Evans.

def Transform(C, S, E, D)  =  //  Do a single step.
     let A = C, S, E, D
     in
     test  Null C  // Is the control empty?
     ifso  Sub_prob_exit A  //  Yes, so terminate a subproblem.
     ifnot  //  No, so branch on top item of control.
       (  let x = t C  // Top of control.
          in
             Is_constant x     -> Eval_constant A
          |  Is_basic x        -> Eval_basic A
          |  Is_variable x     -> Eval_variable A
          |  Is_lambda_exp x   -> Eval_lambda_exp A
          |  x eq BETA         -> Eval_conditional A
          |  x eq AUG          -> Extend_tuple A
          |  x eq GAMMA
               -> ( let r = t S  // The rator.
                    in
                       Is_closure r  ->  Apply_closure A
                    |  Is_basic r    ->  Apply_basic A
                    |  Is_tuple r    ->  Apply_tuple A
                    |  Is_YSTAR r    ->  Apply_Y A
                    |  Is_eta r      ->  Apply_eta A
                    |  Error 'improper rator'
                  )
          |  Error 'bad control'
       )


def Interpret Program  =
     let Control_structure = MakeControl Program
     in
     Evaluate(Control_structure, nil, PE, nil)
     where rec Evaluate(C, S, E, D) =
          test  Null C & Null D  // Are we done?
          ifso  t S  //  Yes, so return...
          ifnot Evaluate(Transform(C, S, E, D))
```

```
//  Data set 1 for the right-hand evaluator.
//  Last modified on 04/19/69 at 12:12 by Evans.

//  The PAL program being represented is:

//         let f t = t + 1 in f 2

def Data =
   LET_
   (  FF_
      ( f, t )
      (  PLUS_ t 1_  )
   )
   (  AP_ f 2_  )
     where
       (       f    = VARIABLE, 'f'
          and t    = VARIABLE, 't'
          and 1_   = CONSTANT, 1
          and 2_   = CONSTANT, 2

          and PLUS_ x y = BINOP_ x y (fn x y.x+y)
       )
```

```
//  Data set #2 for the right-hand evaluator.
//  Last modified on 04/19/69 at 15:48 by Evans.

//  The PAL program:

//          let f x = x + (x > 0  ->  1  |  -1)
//          in f 2  *  f(-3)


def Data =
   LET_
   (  FF_
      ( f, x )
      (  PLUS_
         x
         (  ARROW_
            (  GTR_ x 0_  )
            1_
            (  NEG_ 1_  )
         )
      )
   )
   (  MULT_
      (  AP_ f 2_   )
      (  AP_ f (NEG_ 3_)   )
   )
   where
    (       0_  = CONSTANT, 0
      and 1_  = CONSTANT, 1
      and 2_  = CONSTANT, 2
      and 3_  = CONSTANT, 3

      and f   = VARIABLE, 'f'
      and x   = VARIABLE, 'x'

      and PLUS_ x y = BINOP_ x y (fn x y.x+y)
      and MULT_ x y = BINOP_ x y (fn x y.x*y)
      and GTR_  x y = BINOP_ x y (fn x y.x>y)
      and NEG_  x   = UNOP_  x   (fn x  . -x)
    )
```

```
//  Data set #3 for the right-hand evaluator.
//  Last modified on 04/19/69 at 16:13 by Evans.

//  The PAL program:

//         let f (x, y, z) w  =  x + y + z - w
//         in  f(1, 2, 3) 5


def Data =
   LET_
   (  FF_
      (  f, (x, y, z), w)
      (  SUB_
         (  ADD_
            (  ADD_ x y  )
            z
         )
         w
      )
   )
   (  AP_
      (  AP_
         f
         (  COMMA_3 1_ 2_ 3_  )
      )
      5_
   )
   where
    (      1_  =  CONSTANT, 1
       and 2_  =  CONSTANT, 2
       and 3_  =  CONSTANT, 3
       and 5_  =  CONSTANT, 5

       and f   =  VARIABLE, 'f'
       and x   =  VARIABLE, 'x'
       and y   =  VARIABLE, 'y'
       and z   =  VARIABLE, 'z'
       and w   =  VARIABLE, 'w'

       and ADD_ x y = BINOP_ x y (fn x y. x + y)
       and SUB_ x y = BINOP_ x y (fn x y. x - y)

       and COMMA_3 x y z  =  Tag COMMA (x, y, z)
    )
```

```
//  Data set #4 for the right-hand evaluator.
//  Last modified on 04/19/69 at 17:07 by Evans.

//  The PAL program:

//          let f x = x+1 and g x = x-2 and h x = x-3
//          in f(g(h 5))


def Data =
   LET_
   (  AND_3
      (  FF_ (f, x) (ADD_ x 1_)   )
      (  FF_ (g, x) (SUB_ x 2_)   )
          (  FF_ (h, x) (SUB_ x 3_)   )
   )
   (  AP_
      f
      (  AP_ g (AP_ h 5_)   )
   )

   where
    (       1_    =  CONSTANT, 1
      and 2_    =  CONSTANT, 2
      and 3_    =  CONSTANT, 3
      and 5_    =  CONSTANT, 5

      and f     =  VARIAB-LE, 'f'
      and g     =  VARIABLE, 'g'
      and h     =  VARIABLE, 'h'
      and x     =  VARIABLE, 'x'

      and ADD_ x y  =  BINOP_ x y (fn x y. x + y)
      and SUB_ x y  =  BINOP_ x y (fn x y. x - y)

      and AND_3 x y z  =  Tag AND (x, y, z)
    )
```

```
//  Data set #5 for the right-hand evaluator.
//  Last modified on 04/19/69 at 18:13 by Evans.

//  The PAL program:

//          let f x = x - 1
//          and
//            (        w = 3
//                     within
//               g x = x + w
//               and
//               h x = x - w
//            )
//          in
//          f  (g (h 2) )


def Data =
   LET_
   (  AND_2
      (  FF_ (f, x) (SUB_ x 1_)   )
      (  WITHIN_
         (  EQUAL_ w 3_   )
         (  AND_2
            (  FF_ (g, x) (ADD_ x w)   )
            (  FF_ (h, x) (SUB_ x w)   )
         )
      )
   )
   (  AP_ f (AP_ g (AP_ h 2_))   )

   where
    (      f     =  VARIABLE, 'f'
      and g     =  VARIABLE, 'g'
      and h     =  VARIABLE, 'h'
      and x     =  VARIABLE, 'x'
      and w     =  VARIABLE, 'w'

      and 1_    =  CONSTANT,  1
      and 2_    =  CONSTANT,  2
      and 3_    =  CONSTANT,  3

      and ADD_ x y  =  BINOP_ x y (fn x y. x + y)
      and SUB_ x y  =  BINOP_ x y (fn x y. x - y)

      and AND_2 x y  =  Tag AND (x, y)
    )
```

```
//  Test recursion in the right-hand evaluator.

//  Last modified on 04/26/69 at 14:17 by Evans.

//  The PAL program:

//          let rec f n  =  Less2 n  ->  1  |  n*f(Pred n)
//          in
//          f 3

//  Less2 and Pred are primitive functions, like this:
//          Less2 x  =  x < 2
//          Pred x  =  x - 1


def Data_N = CONSTANT, 3

def Data =
   LET_
   (  REC_
      (  FF_
         (  f, n  )
         (  ARROW_
            (  Less2 n  )
            1_
            (  MULT_
               n
               (  AP_ f (Pred n)  )
            )
         )
      )
   )
   (  AP_ f Data_N  )

   where
    (      f     =  VARIABLE, 'f'
      and n     =  VARIABLE, 'n'

      and 1_    =  CONSTANT,  1
      and 2_    =  CONSTANT,  2

      and Less2 x     =  UNOP_  x   (fn   x.  x < 2)
      and Pred  x     =  UNOP_  x   (fn   x.  x - 1)

      and MULT_ x y  =  BINOP_ x y (fn x y.  x * y)
    )
```

```
//  Definitions and predicates for the jumping evaluator.

//  Last modified on 04/27/69 at 14:11 by Evans.


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

//  Items and predicates for control structure and stack.

def GAMMA        = '_gamma'
and BETA         = '_beta'
and DELTA        = '_delta'
and CONSTANT     = '_constant'
and BASIC        = '_basic'
and VARIABLE     = '_variable'
and ADDRESS      = '_address'   // Used only in stack.
and ASSIGN       = '_assign'    // :=
and GOTO         = '_goto'
and DOLLAR       = '_dollar'
and AUG          = '_aug'
and TUPLE        = '_tuple'     //  Used only in the stack.
and ALPHA        = '_alpha'
and LABEL        = '_label'     // Used only in stack.
and ETA          = '_eta'       // Used in stack for recursion.

def       Test(x, y) =
             Istuple x
             -> Order x eq 2
                -> Isstring(x 1)
                   -> x 1 eq y
                   |  false
                |  false
             |  false
     within
     Is_constant x  = Test(x, CONSTANT)
 and Is_variable x  = Test(x, VARIABLE)
 and Is_address x   = Test(x, ADDRESS)
 and Is_label x     = Test(x, LABEL)
 and Is_eta x       = Test(x, ETA)
 and Is_basic x     = Test(x, BASIC)

 and Is_tuple x =
        Test(x, TUPLE)  ->  true  // Is it a constructed tuple?
     |  Test(x, CONSTANT)  ->  Null(x  2)  // Is it nil?
     |  false  // Neither.

 and Is_identifier x =  // Is x constant, variable or basic?
     Test(x, CONSTANT) or Test(x, VARIABLE) or Test(x, BASIC)


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

//  Call for YSTAR is produced in MakeControl for rec-defs.
//  PI, RHO, 1_ and NIL are used for "valof" and "res".
```

```
def YSTAR  =
     VARIABLE, 'ystar'

and PI  =          // Used in desugaring 'valof' and 'res'.
     VARIABLE, 'pi'

and RHO  =  // Used in desugaring 'valof' and 'res'.
     VARIABLE, 'rho'

and 1_  =  // The constant '1'.
     CONSTANT, 1

and NIL  =
     CONSTANT, nil

and DUMMY  =
     CONSTANT, '_dummy'

def Is_YSTAR x =
     Isstring x -> x eq 'ystar' | false


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

//  Tags for abstract syntax tree.

def TEST      = '_test'      // test ... ifso ... ifnot ...
and ARROW     = '_arrow'     // ... -> ... | ...
and IF        = '_if'        // if ... do ...
and AP        = '_ap'        // functional application
and FN        = '_fn'        // lambda
and EQUAL     = '_equal'     // definition
and WITHIN    = '_within'
and REC       = '_rec'
and FF        = '_ff'        // function form definition
and AND       = '_and'       // 'and' definition
and COMMA     = '_comma'     // tuple maker
and LET       = '_let'
and WHERE     = '_where'
and COLON     = '_colon'
and VALOF     = '_valof'
and RES       = '_res'
and WHILE     = '_while'
and BINOP     = '_binop'
and UNOP      = '_unop'
and PERCENT   = '_percent'


//  Taggers for tags in abstract syntax tree.

def TEST_ x y z      = Tag TEST (x, y, z)
and ARROW_ x y z     = Tag ARROW (x, y, z)
and AP_ x y          = Tag AP (x, y)
and FN_ x y          = Tag FN (x, y)
and LET_ x y         = Tag LET (x, y)
```

```
and WHERE_ x y        = Tag WHERE (x, y)
and EQUAL_ x y        = Tag EQUAL (x, y)
and WITHIN_ x y       = Tag WITHIN (x, y)
and REC_ x            = Tag REC (nil aug x)
and FF_ x y           = Tag FF (x, y)
and AUG_ x y          = Tag AUG (x, y)
and ASSIGN_ x y       = Tag ASSIGN (x, y)
and ALPHA_ x y        = Tag ALPHA (x, y)
and DOLLAR_ x         = Tag DOLLAR (nil aug x)
and GOTO_ x           = Tag GOTO (nil aug x)
and COLON_ x y        = Tag COLON (x, y)
and VALOF_ x          = Tag VALOF (nil aug x)
and RES_ x            = Tag RES (nil aug x)
and WHILE_ x y        = Tag WHILE (x, y)
and BINOP_ x y z      = Tag BINOP (x, y, z)
and UNOP_ x y         = Tag UNOP (x, y)
and PERCENT_ x y z    = Tag PERCENT (x, y, z)

//  AND_ and COMMA_ would have to be n-ary taggers, and hence
//  are not provided.


//  Taggers for standardized syntax tree.

def GAMMA_ x y        = Tag GAMMA (x, y)
and BETA_ x y z       = Tag BETA (x, y, z)
and LAMBDA_ x y       = Tag LAMBDA (x, y)

and DELTA_ x y        = Null  x -> y | Tag DELTA (x, y)


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

//  Some useful functions for transform.
//%
//%def Value_of x  =  // Evaluate a control element, to put it on stack.
//%     x
//%
//%and Val_of x  =  // De-tag a stack element, to get its value.
//%     x 2
//%
//%def Apply x y  =
//%     let t = (Val_of x) (Val_of y)
//%     in
//%     (Isfunction t -> BASIC | CONSTANT), t
//%
//%and Aug x y  =  //  Augment x with y.
//%     Is_tuple x  ->  (TUPLE, Val_of x aug y)
//%     |  Error 'first argument of aug not a tuple'


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

//  Define the five components of the evaluator.  E and M are used
//  as global variables in the following functions.
```

```
def C, S, E, D, M = nil, nil, nil, nil, nil


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

//                      M E M O R Y

// A memory is a 2-tuple, whose first component is that integer
// which is the last address used (initially zero), and whose
// second component is a Mem, like this:

//       A Mem is either empty (nil)
//       or it is a 3-tuple, whose components are
//          an address,
//          a contents,
//          a Mem.

def
    Extend Value  =  // Find a new cell to hold Value.
         M 1 := M 1 + 1;
         M 2 := $(M 1), Value, $(M 2);
         (ADDRESS, $(M 1))

 and
    Update(Cell, Value)  =
         M 2:= $(Cell 2), Value, $(M 2)

 and
    Contents Cell  =
         let c = Cell 2
         in
         Look (M 2)
         where rec Look m =
                 Null m  ->  Error 'address not in memory'
              |  m 1 eq c  ->  m 2
              |  Look (m 3)

 and
    Initialize_memory ()  =
         M := 0, nil


// Two useful functions used by the evaluator.

// Return argument if not an address, and contents otherwise.

def Rval x =
    Is_address x  ->  Contents x  |  x

// Return argument if an address, and new cell containing it
// otherwise.

and Lval x =
    Is_address x  ->  x  |  Extend x
```

```
//  * * * * * * * * * * * * * * * * * * * * * * * * * *

//                    E N V I R O N M E N T

//  An environment is either empty (nil), or a 3-tuple:
//        Name, Value, Environment

//  The primitive environment:

//%def Initialize_env () =
//%     let t = Extend 'ystar'
//%     in
//%     E := 'ystar', t 2, nil
//%
//%
//  The function to look up a variable in the environment:
//%
//%def Lookup Var =
//%     let V = Var 2  // The identifier itself.
//%     in
//%     L E  // Start looking in the environment.
//%     where rec L e =
//%             Null e  ->  Error 'variable not found in environment'
//%           | V eq e 1  ->  e 2  // Found.
//%           | L(e 3)  // Keep looking.
//%
//%
//  The following function is used in applying a lambda-closure.
//  The names on the (possibly structured) bv-part 'Names' are
//  added to the environment 'Env', associated with the corres-
//  ponding part of 'Values'.  The new environment is returned as
//  the value of the function.
//%
//%def rec Decompose(Names, Values, Env)  =
//%     test  Is_variable Names  // Is it a single variable?
//%     ifso  (Names 2, Values, Env)  // Yes, so add it to environment.
//%     ifnot  // Check conformality.
//%     test  Is_tuple Values
//%     ifnot Error 'conformality failure'  // Tuple applied to scalar.
//%     ifso
//%     test  Order Names eq Order (Val_of Values)
//%     ifnot Error 'conformality failure.'  // Differing tuple lengths.
//%     ifso  // Process a multiple-bv part.
//%       (  Q 1 Env
//%          where rec Q n e =
//%                  n > Order Names  ->  e
//%                | Q (n+1) ( Decompose(Names n, (Val_of Values) n, e) 
//%       )


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

//  This function combines two lists of labels.  It is used in
```

```
//  XFORM in putting together a DELTA node.

def Combine(x, y)  =
     Q 1 x
     where rec Q k s =
          k > Order y  ->  s  |  Q (k+1) (s aug y k)
```

```
//                              MakeControl

//  The function MakeControl accepts as input an abstract syntax-
//  tree representation of a program, and produces as output a
//  control structure suitable for input to the evaluator.  It is
//  called by Interpret.

//  Last modified on 05/19/69 at 17:59 by Evans.

//  The structure of the definitions which follows is like this:

//  def
//            rec D x =  standardize definitions
//            within

//       rec ST x =  standardize a syntax tree
//

//  def rec LL x =  process labels

//  def rec FF =  flatten standardized syntax tree

//  def MakeControl P = FF( LL(ST P), nil )


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


def XF_dbg = false  // Control debug printing from these routines.
//
```

```
def
     rec D x =  // Standardize a definition.
          let Type = Is_tag x
          in
             Type EQUAL  ->  x  // Already OK.
          |  Type WITHIN
               -> ( let u, v = D(x 1), D(x 2)
                    in
                    EQUAL_ (v 1) ( GAMMA_ (LAMBDA_ (u 1) (v 2)) (u 2) )
                  )
          |  Type REC
               -> ( let w = D(x 1)
                    in
                    EQUAL_ (w 1) ( GAMMA_ YSTAR (LAMBDA_ (w 1) (w 2)) )
                  )
          |  Type FF
               -> ( EQUAL_ (x 1 1) (Q (Order(x 1)) (x 2))
                    where rec Q k t =
                         k < 2  ->  t
                         |  Q (k-1) (LAMBDA_ (x 1 k) t)
                  )
          |  Type AND
               -> ( EQUAL_ L (Tag COMMA R)
                    where rec L, R = Q 1 nil nil
                    where rec Q k s t =
                         k > Sons x  ->  (s, t)
                         |  ( let w = D(x k)
                              in
                              Q (k+1) (s aug w 1) (t aug w 2)
                            )
                  )
          |  Error 'improper node found in D'
     within

rec ST x  =  // Standardize abstract syntax tree.
  if XF_dbg do Write( Get_tag x, '*n' );
  (
     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type BETA or Type TEST or Type ARROW
          ->  BETA_ (ST(x 1)) (ST(x 2)) (ST(x 3))
     |  Type IF
          ->  BETA_ (ST(x 1)) (ST(x 2)) DUMMY
     |  Type LAMBDA or Type FN
          ->  LAMBDA_ (x 1) (ST(x 2))
     |  Type COMMA
          -> ( Q 1 NIL
               where rec Q k t =
                    k > Sons x  ->  t
                  |  Q (k+1) ( AUG_ t (ST(x k)) )
             )
     |  Type PERCENT
          -> GAMMA_ (x 2) ( AUG_ (AUG_ NIL (ST(x 1))) (ST(x 3)) )
     |  Type COLON
```

```
             -> ( let w = ST(x 2)
                  in
                     Is_tag w COLON   ->  COLON_ (w 1 aug x 1) (w 2)
                  |  COLON_ (nil aug x 1) w
                )
        |  Type LET
             -> ( let w = D(x 1)  // Standardize the definition.
                  in
                  GAMMA_ ( LAMBDA_ (w 1) (ST(x 2)) ) (ST (w 2))
                )
        |  Type WHERE
             -> ( let w = D(x 2)  // Standardize the definition.
                  in
                  GAMMA_ ( LAMBDA_ (w 1) (ST(x 1)) ) (ST (w 2))
                )
        |  Type VALOF
             -> ( let w = GAMMA_ PI 1_
                  in
                  let v = COLON_ (nil aug RHO) w  // RHO: w
                  in
                  let u = ASSIGN_ PI (AUG_ NIL (ST (x 1)))
                  in
                  GAMMA_ (LAMBDA_ PI (ALPHA_ u v)) NIL
                )
        |  Type RES
             -> ( let w = ASSIGN_ PI (AUG_ NIL (ST (x 1)))
                  in
                  ALPHA_ w (GOTO_ RHO)
                )
        |  Type AP  ->  GAMMA_ (ST(x 1)) (ST(x 2))
        |  Type BINOP
             ->  GAMMA_ ( GAMMA_ (BASIC, x 3) (ST(x 1)) ) (ST(x 2))
        |  Type UNOP
             ->  GAMMA_ (BASIC, x 2) (ST(x 1))
        |  Sons x eq 1 -> Tag (Get_tag x) ( nil aug ST(x 1) )
        |  Sons x eq 2 -> Tag (Get_tag x) ( ST(x 1), ST(x 2) )
        |  Error 'improper node found in ST'
    )
//
```

```
//  The function LL processes labels, bringing each label as far
//  up the tree as possible.  The effect is that each label is
//  declared by a DELTA node as soon as its scope is entered.

def
            Proc_labels x =
                 Is_tag x DELTA
                 ->  (x 1, x 2)
                 |   (nil, x)
            within

            Combine_labels(u, v)  =
                 let U = Proc_labels u
                 and V = Proc_labels v
                 in
                 Combine(U 1, V 1), (U 2, V 2)
            within

rec LL x =
  if XF_dbg do Write( Get_tag x, '*n' );
  (
     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type ALPHA
          -> ( let s, w = Combine_labels( LL(x 1), LL(x 2) )
               in
               DELTA_ s ( ALPHA_ (w 1) (w 2) )
             )
     |  Type BETA
          -> ( let s, w = Combine_labels( LL(x 2), LL(x 3) )
               in
               DELTA_ s ( BETA_ (LL(x 1)) (w 1) (w 2) )
             )
     |  Type WHILE
          -> ( let s, w = Proc_labels( LL(x 2) )
               in
               DELTA_ s (WHILE_ (LL(x 1)) w)
             )
     |  Type COLON
          -> ( let L, z = Proc_labels(LL(x 2))
               in
               let w = Lval z
               in
               DELTA_ (Q 1 nil) w
               where rec Q k t =
                    k > Order(x 1)  ->  t  |  t aug x 1 k aug w
             )
     |  Type LAMBDA  ->  LAMBDA_ (x 1) (LL(x 2))
     |  Sons x eq 1  ->  Tag (Get_tag x) ( nil aug LL(x 1) )
     |  Sons x eq 2  ->  Tag (Get_tag x) ( LL(x 1), LL(x 2) )
     |  Error 'improper node in LL'
  )
//
```

```
// The function FF flattens a standardized tree into a
// control structure.

def rec FF(x, c) = // Flatten standardized tree x onto control c.
  if XF_dbg do Write( Get_tag x, '*n' );
  (
     let Type = Is_tag x
     in
        Is_identifier x -> (x, c)
     |  Is_address x -> ( Update(x, FF(Contents x, nil)); (x, c) )
     |  Type LAMBDA
          -> ( let Body = Lval(  FF( x 2, nil ) )
               in
               Cons_lambda_exp(x 1, Body), c
             )
     |  Type BETA
          -> ( let TA = Lval( FF(x 2, c) )  // True arm.
               and FA = Lval( FF(x 3, c) )  // False arm.
               in
               FF( x 1, (BETA, (TA, FA)) )
             )
     |  Type ALPHA
          -> ( let Rest = Lval( FF(x 2, c) )
               in
               FF(x 1, (ALPHA, Rest))
             )
     |  Type DELTA
          ->   ( DELTA, (x 1, (FF(x 2, nil), c)) )
     |  Type WHILE
          -> ( let w = Lval NIL
               in
               let TA = FF(x 2, (ALPHA, w))
               and FA = DUMMY, c
               in
               Update ( w, FF(x 1, (BETA, (TA, FA))) );
               (w, nil)
             )
     |  Sons x eq 2 -> FF( x 2, FF( x 1, (Get_tag x, c) ) )
     |  Sons x eq 1 -> FF( x 1, (Get_tag x, c) )
     |  Error 'Improper node found in FF'
  )

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


def MakeControl P  =  // The routine that does all the work.
     FF ( LL( ST P ), nil )
```